Formal Modelling of Information System Evolution using B

Pascal André
LS2N - Nantes Université - École Centrale Nantes - CNRS
Nantes, France *pascal.andre@ls2n.fr*

Henri Habrias
LS2N - Nantes Université - École Centrale Nantes - CNRS
Nantes, France *henri.habrias@univ-nantes.fr*

Abstract

In this paper, we explore the problem of predictable evolution of Information System (IS). In this context, a part of the Information System evolution is modelled through phases and interphases in order to specify evolution rules. A formal model is provided using the B language to check the rule effectiveness.
Keywords: IS Modelling, Predictable System Evolution, Phases, Formal Methods, B

1. Introduction

Software evolution usually focuses on the maintenance (improvement) of a software system along its lifecycle. Usually, it is perceived a continuous change from a lesser, simpler, or worse state to a higher or better state [7]. When you consider the client point of view, only stable changes (or states) are visible: there are the delivered software releases. Thus the evolution becomes time-discrete and can be represented by a state machine (or any equivalent dynamic model), where the *states* (steps) of the software lifecycle are different (sometimes concurrent) releases of the "same" system and the *transitions* between states are software developments and installations. Since cycles are merely unwhished, software evolution is rather a sequence of evolving states rather than a real (lifecycle) graph (even a tree). The transitions are not predicted tasks but rather depend on first system verification and validation (tests, current usage, bugs) and, second the requirements evolution (adding new functions, modifying existing parts...). The evolution management is to control these transitions to keep the system consistent.

In this paper, we work on a restricted frame : software systems are information systems (IS), whom evolution is already known. Each step (state) of the evolution is called a **phase**. In each phase, the system behaves in a specific way, which can be different from another phase. The transition from one phase to another is a phase called **interphase** whom rules are temporary. The evolution can be **cyclic** since periods, *e.g.* like the four seasons in agriculture, can be a reason for evolving software. We call this general frame the **predictable evolution of information systems**. This is a different view of the IS evolution paradigm mentioned in [1], which targets non cyclic IS, as well as evolution oriented approaches such as [14], [9].

There are many and many formalisms, methods and tools for modelling information systems. Some are semi-formal (UML, E-R,...) other are formal (algebraic, state-based), some define several aspects or views (static, dynamic, functional). Here, we choose a formal state-based approach, the B notation, where a system is specified by a (mathematical) state and operations on that state. The formal (mathematical and logical) notation allow formal checking and proof of properties. B also benefits from automated toolkits for writing, proving and refining specifications. More specifically, we use a standard B language [2], [10] instead of the more recent Event-B [4] because we do not discover events during the refinement. Shortly, a standard B machine models a system through an invariant property and presents a set of operations that query or modify this state. Each operation is to be proved against the invariant. The theorem to

prove is: the conjunction of the invariant and the precondition implies that the substitution preserves the invariant. It is clear that the proof relates to the invariant, which must fix the system's properties, preserved all along the system's life.

The paper is organized as follows. The general frame on predictable evolution of information systems is introduced in section 2. Then we overview several approaches for modelling such an evolution in section 3. The rest of the paper is devoted to the use of a formal method to implement phases and interphases. The formal B notation is overviewed in Section Aof the appendix we made available online[1]. Formal methods are gainful not only to specify information systems but also to verify the conformance of the system evolution. The matter is illustrated in Section 4 on a simple example of university scholarship management.

2. Modelling System Evolution

In this section, we show that phases are necessary to model evolving aspects of the system. We also explore the relation between phases: the system transitions.

2.1. Information System Modelling

We assume to be at a modelling level and not at an implementation level. Reasoning on models before implementing them has been proved useful for a long time. The recent trends in software engineering enforce this idea, since they provide central role to models in business system development. This is the *credo* of the OMG, promoting the Model Driven Approach [13], [11]. Moreover, formal models are necessary to get any confidence in the model semantics, to prove properties and to automate model transformation. This explains the use of a formal notation, the B language, in our work.

Three-dimension Models A system model is a combination of three aspects: static, functional and dynamic. The *static* part (static model in UML [8]) describes the invariant properties of the system. In a model-based approach, the static model specifies the system structure by a set of typed variables (types can be classes in object-oriented approaches as in UML) and constraints on these variables. The *functional* part describes the system computations. In a model based approach, it describes changes on the variables of the system structure through operations. The *dynamic* part describes how the system evolves. In UML, the dynamic behaviour is defined in each (dynamic) class, as a statechart, thus the dynamic behaviour of the whole system is never specified explicitly. In other languages, like the standard B, the dynamic behaviour is assumed to be sequential application of operations, conditioned by their precondition.

What happens in practice? Since many years, the static model is the main model for system modelling, even for UML models. The functional part, we mean more than the operation profiles, is often delayed to implementation ; except for formal methods which provide an abstract semantics (axioms, pre/post conditions...). In any cases, defining a system dynamics is merely neglected in most modelling languages: it is either implicit (sequential) or distributed (local behaviour of processes or actors).

Toward a macro dynamic modelling The "great" static model states, at a macro-level, what is invariant along the whole system lifecycle. The current dynamic models, when exist, describe micro-evolution (*e.g.* object evolution). There is then some target mismatch in modelling: we need both finer invariants (for example, something which must be true for a period of time) and coarser dynamic changes (for example, the change of a composite part which is not only the union of the changes of its components).

Let us take the example of a commercial activity. During the sales season, the commercial

[1] `https://uncloud.univ-nantes.fr/index.php/s/S4ZXm9mQNBZMg6k`

rules change: the prices are lower (10, 30, 50 percent), the customer cannot get back the sold articles, there is no discount, etc. During the inventory, a part of (or all) the stock is unavailable for selling. In clothing trade, they are seasonable collections. All these examples illustrate various periods for the commercial activity.

In order to handle these situations, especially when the target system is some complex process, we propose to model macro-evolutions by phases and interphases.

2.2. Phases and Related Paradigms

In front of a complex system, the specifier tries to decompose it into smaller parts: the subsystems. The system is then a **composition** of the subsystems, the composition rules define the relation between the subsystems and the system (collaboration, delegation, control...). This is a (well-known?) horizontal composition. The system is the union of the subsystems.

We also need a vertical relation between subsystems to handle the fact that the system scope may vary along the time. We mean that only a part of the system has some meaning at a given time. This part is called a phase. A **phase**, in its own, is defined as a static model. In other words, one phase is the static model of its system at a given time. Thus the static model of a system (modelled through phases) is the slicing of its phases.

Be careful, phases do not confuse views or aspects which are overlapping horizontal slices, while phases are vertical slices (Figure 1). Phases are close to states in a statechart, when considering one object as a system (UML), or **dynamic interface**. More generally, phases denote successive appearances of the system, similar to the lunar phases or physically distinctive forms of a substance, such as the solid, liquid, and gaseous states of ordinary matter[2]. In Biology, a phase[3] is a characteristic form, appearance, or stage of development that occurs in a cycle.

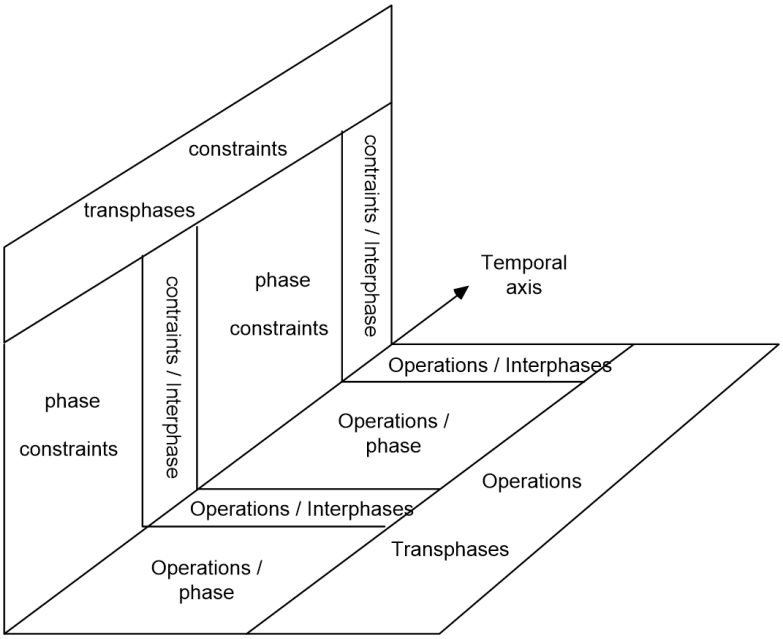

Fig. 1. Phases and Interphase along the Temporal Axis

Relations between Phases There are several sorts of relation between phases : a temporal relation (a phase precedes another phase), a composition relation (a phase includes -maybe with modalities- another phase), a taxonomic relation (a phase is a "subtype" of another phase). The temporal relation is to be the interphase concept (see section 2.3). The structural relation will be based on B machine structural relations. The taxonomy (or classification) is reduced to a simple inclusion: the **transphase** is the intersection of the phases, that is the smallest invariant of the system static model (Figure 1); it is implemented by a B machine inclusion.

[2]http://en.wikipedia.org/wiki/Phase
[3]http://dictionary.reference.com

2.3. Phases and Interphases

What can happen between two successive phases? Usually, in database systems, the transformation between two stable states of the system is called a **transaction**. A transaction takes a database that conforms to the static model of the system, and provides an updated database that still conforms to the (same) static model of the system. Both database states conform to the SAME invariant. This is not exactly what we want, since the invariant may differ in the two phases.

Again, from Biology, one can find that an interphase is the stage of a cell between two successive mitotic or meiotic divisions (the phases). At first sight, this definition is clearly what we need. When we study it more precisely, an *"Interphase is a phase of the cell cycle, defined only by the absence of cell division. Cells during interphase may or may not be growing.[...]"*[4] In this definition, an interphase is a phase, and the problem remains: what can happen between two successive phases or interphases?

Let us adopt the following meaning. In its general semantics an **interphase** occurs between two successive phases. An interphase can be implemented as an atomic transformation (a transition in the state machines) or a phase (when it is somewhat complex). In this last case, a new concept is to be introduced to permit phase transfer. In any cases, an interphase should be a short period of time.

Figure 1 illustrates the phasing of an information system. The sytem model evolves along a temporal axis (dynamics). The evolution includes a (static) invariant part and a (functional) operation part. The interphase semantics is quite loose, allowing static and functional descriptions. The piece of model, which is common to all phases, is called the **transphase**. Carefully note, that the same phase may occur several times (cycles are allowed), as illustrated in Figure 2.

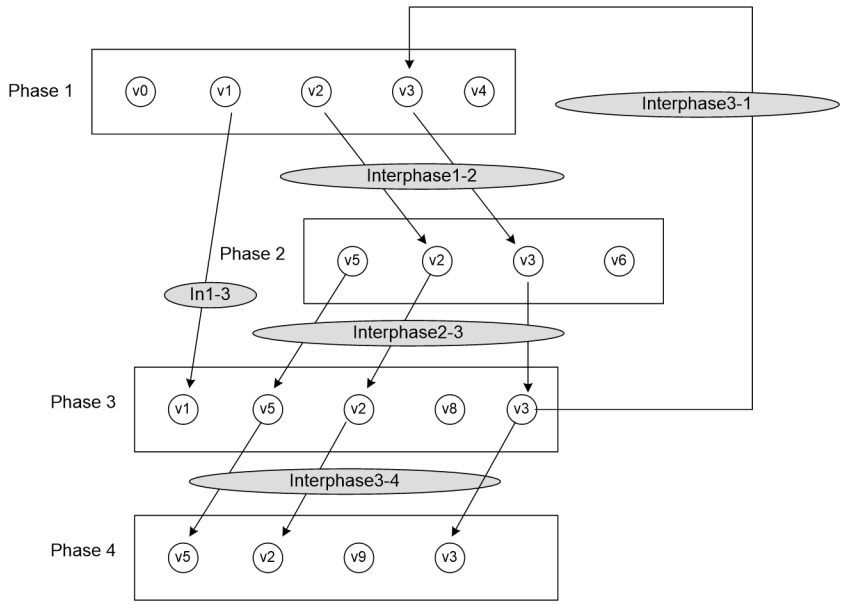

Fig. 2. Evolution of Phase Variables

Figure 2 shows the evolution of the system variables when phases change: (i) some variables exist in one phase only (`v0`, `v6`), (ii) some variable appear from a phase (`v5`), (iii) some variable disappear after a phase (`v1`). We assume that variables always have the same type and meaning, but the constraints may vary. Interphases and phases can : (i) appear in circles: phase `3` go back to phase `1`, (ii) be alternatives: phase `3` to `1` OR phase `3` to `4`.

[4] http://en.wikipedia.org/wiki/Interphase

In summary, it is a kind of state-transition system. We assume a "continuity" constraint: if a variable is needed later, it must be stored in the intermediate phases with a constant phase dynamic constraint (section 2.4). For example, interphase `In1-3` is not allowed and the `v1` variable should be represented in phase `2`.

2.4. More on Phases and Interphases

Phases are stable states of the system lifecycle, each defined by a particular invariant. In a phasing system, one may want to specify interphase policies or phase dynamic constraints.

Interphase Policies The transition can be revolutionary (delete all, create new), conservative (keep all what preserve the "new" invariant) or interactive (ask the user for the conflicting elements). Interphases can have parameters.

Phase Dynamic Constraints Sometimes, we want to prevent any modification on a variable or to set some relation between the value of the variables at the "beginning" of the phase and the value at the end, or between phases.

3. Implementing Phase and Interphase

In this section we overview various implementations of phases and interphases. It is illustrated with the B notation[5].

3.1. Implementing a System Model

A system model is specified by a B machine. Its general syntax is the following (only a part of B clauses is shown). The `VARIABLES` and `VARIABLES` clauses describe the static part. The `OPERATIONS` clause describes the functional part. The mandatory `INITIALISATION` clause enforces the existence of at least one "correct" state for the machine (a model).

Remember that in a B machine, each operation (querying or updating the state) is to be proved to preserve the invariant (property). This implies that *if a property cannot be expressed in the invariant, no proof will ensure the preservation of the property.* This is sometimes forgotten in practice. Nevertheless a precondition can be stronger than the invariant.

MACHINE
  Name
VARIABLES
  x
INVARIANT
  $x \in Type$
INITIALISATION
  $x := expr$
OPERATIONS
$Op1(pp, ss) =$
PRE
  predicate
THEN
  substitution
END
END

3.2. Implementing the Phases

Phases are a sort of dynamic (or evolving) interfaces. There are numerous ways to implement dynamic interfaces: aspect programming, state machines, multiple inheritance (polymorphism), guarded preconditions... An important constraint is: the system is the same for all phases. In an object model, it means that the system identity never changes whatever phase it is in. Aspect programming and multiple inheritance are quite different since in our model there is one aspect only at a given moment. By guarded preconditions, we mean that the specifier determines the

[5]The operator syntax is provided in Section Aof the web appendix[1].

precondition of operations by a predicate on some variables of the system state. It does not map to our context since we require clearly identified steps: the phases. Last, state machines can be taken as a starting point, because phase and state are similar concepts, but transition and interphase can be quite different. In the following, we borrow the state machine analogy for implementing phases and interphases.

A phase is identified by a name (or a number to simplify the phase ordering and comparison). A phase characterised by variables, an invariant property, and operations (just like a B machine). The implementation is done along two main axis: static/dynamic separation, centralized/distributed control. Like the state machines, the phases can be implemented within the system model or apart. The state machine can be implemented within one centralized machine or distributed over several machines, as illustrated by Table 1.

Table 1. State distribution

B machines	Centralised	Distributed
integrated dynamics	1 machine only	1 for the system + 1 per phase
separate dynamics	1 for the system + 1 for all phases	1 for the system + 1 per phase

In order to improve the specification readability, evolutivity and maintainability we rather choose the separate and distributed version. So we need at least one B machine for the system model and one for the phasing system (separate), one for each phase (distributed). For sake of simplicity, the common part shared by each phase *i.e.*, the transphase, is stored as the system model. We currently do not take into account the phases constraints of section 2.4.

3.3. Implementing the Interphases

Interphases can be transitions (atomic transformations) or phases (durable transformations). If an interphase is a phase, then it must define a specific invariant according to the interphase policy (section 2.4). Defining the invariant between two stable states of the system (the phases) is quite difficult, because the rules are fuzzy. For example, what is the price of an article between the normal sell and the low cost sell? Does it depend on the vendor?

We choose an intermediate way that preserves the definition of stable states for the system (Figure 3). A phase is divided in three subphases: the phase preparation (init), the phase progress (run) and the phase termination (close). Each subphases must conform to the phase invariant. An interphase is then an atomic transition from a phase termination to a phase preparation. The atomicity is ensured by the fact that an interphase is implemented as an operation. By default, the variables appearing in phase `n` are equal to those of phase `n-1`.

In B, the precondition of an operation ensures that the (system) state after the operation preserves the system invariant whenever the (system) state before the operation conforms to the system invariant. Usually in a B specification the invariant *before* and the invariant *after* are the same. In our implementation, an interphase is a special operation whose precondition applies to the source phase invariant and whose postcondition (i.e. the state after substitution) applies to the target phase invariant.

4. Application

In this section, we illustrate our approach on a well-known case study; the annual academic schedule at university. After a short presentation of the case study in section 4.1, we specify the system in B in section 4.2 and reason about it in section 4.3.

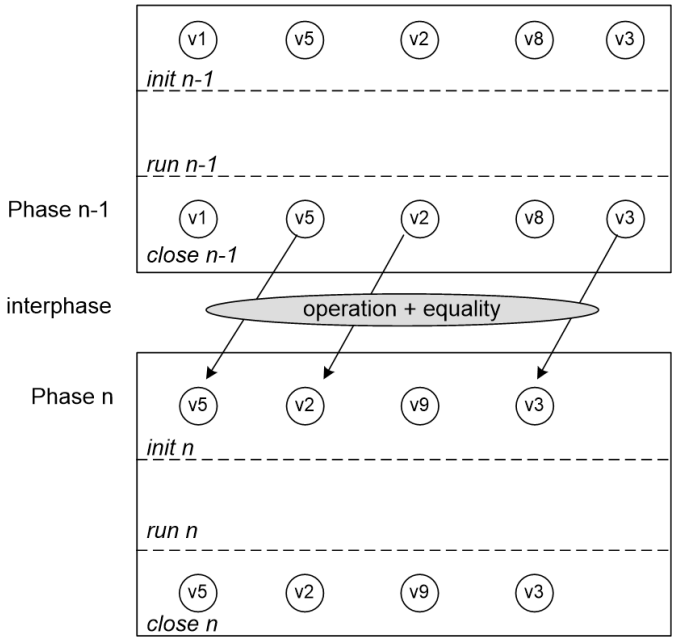

Fig. 3. Splitting phases into subphases

4.1. The Case Study

In our (strongly simplified) school, we distinguish two phases in the university year. In the first one, we use the file of the students of the preceding year and the file of the open courses of the preceding year. New students can register and new courses can be added and old courses suppressed. Students register to no more than one open course. At phase 2, we keep only the students registered to a course and the courses having at least a student registered in this course and, of course, we keep which student is registered to what course. During this phase, students can succeed. In this case, we register who succeed to what course. A student cannot succeed to a course where he is not registered in. At the end of phase 2, we update the history file of the results. We do not keep then the registration to courses. But we keep the students of the year and the courses of the year. We use these files at the beginning of the new academic year.

4.2. A Specification of the Case Study in B

According to the principles edicted at the end of section 3.2, the specification is made of two parts: the system model (including the transphase) and the phasing model (including a machine for all phases). Figure 4 highlights the structure of the B specification.

The SEES clause allows read access to the variables of the seen machine ; the sets and constants are visible. The INCLUDES clause allows an write access to the variables of the included machine ; the sets and constants are visible. The INCLUDES clause is transitive. The EXTENDS clause is an INCLUDES clause where the operations of the included machine are promoted as operations of the extending machine.

In the following, the variables have short names to gain some place in the paper.

a) The Core System Model

The system model includes the context machine and the transphase machine.

The *Context* machine declares the common given sets (types).

MACHINE
  Context
SETS
  STUDENT; *COURSE*
END

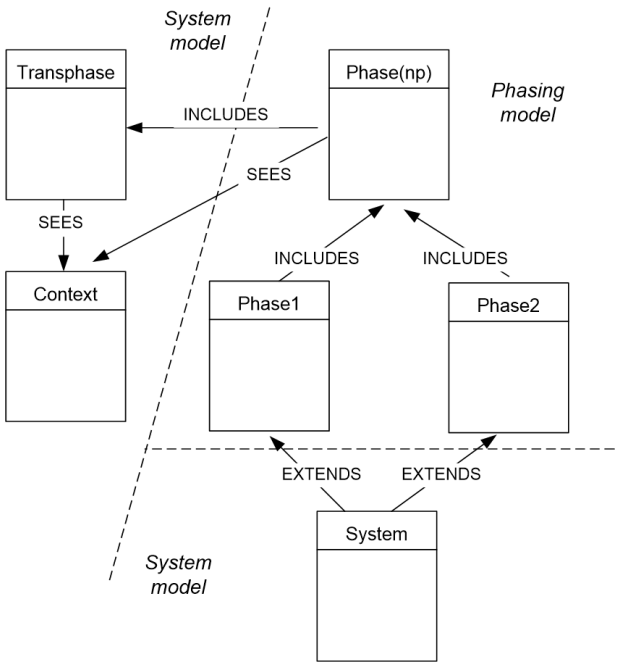

Fig. 4. Structure of the B Specification

The *Transphase* machine declares the shared variables and operations: this is the common system model.

MACHINE
Transphase
SEES
Context
VARIABLES
c_open_courses /* the current courses */
, *c_courses_reg* /* the current registrations */
, *c_year* /* the students of the current year */
, *results* /* the past results */
INVARIANT
$c\_open\_courses \subseteq COURSE$
$\wedge\ c\_year \subseteq STUDENT$
$\wedge\ c\_courses\_reg \in c\_year \leftrightarrow c\_open\_courses$
$\wedge\ results \in STUDENT \leftrightarrow COURSE$
INITIALISATION
$c\_open\_courses, c\_year := \emptyset, \emptyset$
$\|\ c\_courses\_reg, results := \emptyset, \emptyset$

OPERATIONS
student_creation $\widehat{=}$
/* register a new student at university */
PRE
$STUDENT - (\mathrm{dom}(results) \cup c\_year \cup \mathrm{dom}(c\_courses\_reg)) \neq \emptyset$
/* there exists a new student */
THEN
ANY *st* WHERE
$st \in STUDENT - (c\_year \cup dom(results))$
THEN
$c\_year := c\_year \cup st$
END;
student_removal(st) $\widehat{=}$
skip /* not specified yet */
course_creation $\widehat{=}$
skip /* not specified yet */
course_suppress(co) $\widehat{=}$
skip /* not specified yet */
END

Such a transphase means that we can always add new students and courses during the university schedule. If not, these variables have to be stored in the specific phases. Some operations are not specified here.

b) The Phasing Model

The phasing model installs the general framework schema: phases, subphases. It is then specified for each phase.

The Abstract Phase Model This machine specifies the general phasing mechanism for our case study.

MACHINE
Phase(*np*)
CONSTRAINTS
$np : NAT \land np > 1$
SEES
Context
INCLUDES
Transphase
CONSTANTS
al_interph
PROPERTIES
$al\_interph \in 0 \,..\, np \leftrightarrow 1 \,..\, np$
$\land\ al\_interph = \{(0 \mapsto 1), (1 \mapsto 2), (2 \mapsto 1)\}$
/* allowed interphases: 0 to 1 = first year */
/* 1 to 2 = end of registration */
/* 2 to 1 = start a new year */
VARIABLES
c_phase /* the current phase number */
, *subphase* /* the current subphase number */
/* 1 : init, 2 : run, 3 : close */
INVARIANT
$c\_phase \in 0 \,..\, np$
$\land\ subphase \in 1 \,..\, 3$
INITIALISATION
$c\_phase := 0$ /* the initial phase number */
$\parallel\ subphase := 3$ /* the initial subphase number */
OPERATIONS
$n\_phase \leftarrow what\_is\_the\_phase \mathrel{\widehat{=}}$
$n\_phase := c\_phase$
END

The `np` parameter is a constant, which value is given for the actualised machine. In our example, the correct actualisation is `Phase(2)`, meaning that there are two phases. The `al_interph` constant stores which interphases are allowed. It is a relation because various interphase can occur. The PROPERTIES clause is a predicate on the constants. In our example, it is a simple yearly cycle: 0 to 1, 1 to 2, 2 to 2. The `subphase` variable highlights the internal structure of a phase, the invariant means that each phase has exactly three subphases. The INITIALISATION clause[6] means that a phase, with number 0 and subphase 1, asserts a correct phase. In B, it does not exactly means the value by default or at the beginning, but it is often interpreted as so by the readers.

The Phase 1 Model This machine specifies the behavior during registration. In phase 1, one can add new (or remove) students or courses, register a student for a course or cancel a registration.

MACHINE
*Phase*1
INCLUDES
Phase(2) /* instanciated abstract machine */
PROMOTES
student_creation
VARIABLES
res /* the results at the beginning of phase 1 */
INVARIANT
$res \in STUDENT \leftrightarrow COURSE \land c\_phase = 1$
$\land\ (subphase > 1 \Rightarrow results = res)$
/* the results are constants in phase 1 */
INITIALISATION
/* the INITIALISATION of Phase(2) applies */
$c\_phase := 1 \parallel res := \emptyset$
/* an arbitrary specific INITIALISATION is given for phase 1 to facilitate the proof obligation */
OPERATIONS
$init\_phase1 \mathrel{\widehat{=}}$
PRE
$c\_phase = 1$ /* allowed for phase 1 */
$\land\ subphase = 1$
THEN
$c\_phase := 1 \parallel subphase := 2$
$\parallel\ c\_courses\_reg := \emptyset$ /* no registered students */
/* current course and students remain */
$\parallel\ res := results$ /* current results are stored */
END
$registration(st, co) \mathrel{\widehat{=}}$
/* registers the student st to the course co */
PRE
$c\_phase = 1$ /* allowed for phase 1 */
$\land\ st \in c\_year$ /* st is a current student */
$\land\ co \in c\_open\_courses$ /* co is an open course */

[6] INITIALIZATION in the B Book [2].

```
THEN
   c_courses_reg := c_courses_reg ∪ {st ↦ co}
    /* a new registration */
END
cancel_registration(st, co) ≙
   skip    /* not specified yet */
course_suppress(co) ≙
   skip    /* not specified yet */
student_suppress(st) ≙
    /* delete the st student */
PRE
   c_phase = 1 /* allowed for phase 1 */
   ∧ st ∈ c_year /* st is a registered student */
   ∧ st ∉ dom(c_courses_reg)
THEN
   student_removal /* from transphase */
END
close_phase1 ≙
PRE
   c_phase = 1 ∧ subphase = 2
    /* allowed for phase 1 */
THEN
   IF c_courses_reg ∉ (c_year ⇸ c_open_courses)
   THEN
      /* some registration are lost arbitrarily */
      ANY ccr WHERE  /* a correct registration */
         ccr ∈ c_year ⇸ c_open_courses
         ∧ dom(ccr) = dom(c_courses_reg)
         ∧ ccr ⊆ c_courses_reg
      THEN
         c_courses_reg := ccr
      END
   END
   || c_year := dom(c_courses_reg)
      /* current students are registered */
   || c_open_courses := ran(c_courses_reg)
      /* current courses are registered */
   || subphase := 3
   END
END
```

The operation *student_creation* comes from the *Transphase* included machine. The PROMOTES clause means that *student_creation* is now an operation of the *Phase*1 machine. The included INITIALISATION clause is composed sequentially with the *Phase*1 clause INITIALISATION clause. The students can register for any course, but at the end of phase1 each student is registered once at most (to be correct in phase 2): the closure operation transforms a relation to a function whom registered students have one course only. The current students and open courses are those registered only.

All the operations are authorized for phase 1 only. This predicate is required in the precondition when the global system includes all phase definitions.

In this phase, we have an example of dynamic constraint (section 2.4): the result variable is constant during this phase. This is represented by a new variable *res* set at the beginning of the phase and an invariant on that variable, which asserts that *result* = *res* for all the phase (this is the goal of an invariant!).

The Phase 2 Model This machine specifies the behavior during the regular period. The current students and courses do not evolve (dynamic constraint), but students can change their registration (take another current course). Success to examinations are "private" variables of phase 2. At the end, the current successes are stored in the *result* variable to prepare the next year.

The specification is given in Section B of the web appendix[1]. All the operations are authorized for phase 2 only.

c) The System Model

The system model is the union of the concrete phase machines and implements and the interphases. The phases machines are included in the system machine and all their operations are promoted as system operations: this is the definition of the EXTENDS clause. The resulting machine is obtained by putting together the following clauses of *Transphase*, *Phase*, *Phase*1, *Phase*2, *System*: concatenation of SET clauses, concatenation of CONSTANTS clauses, conjunction of PROPERTIES clauses, concatenation of VARIABLES clauses, conjunction of INVARIANT clauses, multiple composition of INITIALISATION ([2], p. 313).

```
MACHINE
  System
EXTENDS
  Phase1; Phase2
INITIALISATION
  c_phase, subphase := 0, 3
OPERATIONS
interphase ≙
  /* implements a non-deterministic phase
     transition function */
PRE
  subphase = 3/* in closure of the previous phase*/
THEN
  ANY toPhase WHERE
    (c_phase ↦ toPhase) ∈ al_interph
  THEN
    c_phase, subphase := toPhase, 1
  END;
END
END
```

In this example, we implement a simple non-deterministic transition relation. We can be more specific in defining an operation for each interphase that calls the *close* and *init* operations of the phases, or provides some specific parameters.

4.3. Verification

Using a free B Toolkit [3], we write the specifications in an ASCII format. A syntactic control is provided and proof obligations are checked. The proof obligations deal mainly with invariant, initialisation and operations. Some specification add-ons improve the effective proof of the above proof obligations. Other individual theorems can be proved. We replayed with AtelierB [12], an up-to-date supporting tools. We had to modify the specification to conform to recent B rules. Details are in Section C of the web appendix[1]. The machine correctness establishes the feasibility of the phasing system by enabling cyclic "stable" invariants.

5. Conclusion

In this paper, we showed how multiple invariants are necessary for a predictable evolution of an Information System. Several invariants is a powerful means to control precisely the system properties at different steps of its evolution lifecycle. Since the functions of the system (our operations) are valid against the invariant, a general and global invariant is usually quite empty or very complicated (many conditions in the predicates).

We proposed a conceptual model based on phases and interphases. We implemented these concepts using a formal method, the B notation. The use of B was motivated by the need to specify formally the invariants and operations in order to prove their validity. This was done on a simple case study using a free B prover. Refinement and implementation of the machines have not been specified yet. The resulting specification is quite readable, even for those unfamiliar with formal notations. The framework can be reused for other case studies.

Nevertheless, this work has to be extended. For example, it might be interesting to provide an interphasing where some variables are "lost" in intermediate phase, for example by providing temporal constraints on the variables. Here is a classification of such properties ([5], p. 77). (i) a reachability property states that some particular situation can be reached. (ii) a safety property expresses that, under certain conditions, something never occurs. (iii) a liveness property expresses that, under certain conditions, something will eventually occur. (iv) a fairness property expresses that, under certain conditions, something will (or will not) occur infinitely often.

We experiment various implementations for a phasing system (transitions arrays, a single phasing machine...). Having a hierarchical structure improves readability, reusability and maintainability but also proof obligations. To be more general, we shall present the refinement process of the B machines but also general frame like those of the "design patterns"[6]. However, a true application on information systems, should handle database implementation in addition to programs. A track is to study the feasibility of SQL triggers to handle the phasing concepts.

References

[1] Aboulsamh, M.A., Davies, J.: A formal modeling approach to information systems evolution and data migration. In: International Workshop on Business Process Modeling, Development and Support. pp. 383–397. Springer (2011)

[2] Abrial, J.R.: The B-Book Assigning Programs to Meanings. Cambridge University Press (1996), iSBN 0-521-49619-5

[3] Abrial, J.R., Cansell, D.: Click'n prove: Interactive proofs within set theory. In: Basin, D.A., Wolff, B. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 2758, pp. 1–24. Springer (2003), also available at http://www.loria.fr/~cansell/cnp.html

[4] Abrial, J.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)

[5] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P., McKenzie, P.: Systems and Software Verification. Springer (2001), iSBN 3-540-41523-8

[6] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-state Verification. In: 2nd Workshop on Formal Methods in Software Practice (March 1998)

[7] Group, G.T.R.E.: Glossary of reengineering terms (2022), `https://sites.cc.gatech.edu/reverse/glossary.html` [Accessed: (12/04/2025)]

[8] Group, O.M.: Unified Modeling Language Specification, version 1.5. Tech. rep., Object Management Group, http://www.omg.org/cgi-bin/doc?formal/03-03-01 (Jun 2003)

[9] Gustas, R.: Modeling approach for integration and evolution of information system conceptualizations. In: Frameworks for Developing Efficient Information Systems: Models, Theory, and Practice, pp. 146–175. IGI Global (2013)

[10] Habrias, H.: Spécification formelle avec B. Hermes Lavoisier (2001), iSBN 2-7462-0302-2

[11] Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Object Technology Series, Addison-Wesley, 1 edn. (2003), iSBN 0-321-19442-X

[12] Lecomte, T.: Atelier b. Formal Methods Applied to Complex Systems: Implementation of the B Method pp. 35–46 (2014)

[13] Miller, J., (Eds), J.M.: Model Driven Approach, MDA Guide Version 1.0.1. Tech. rep., Object Management Group, http://www.omg.org/docs/omg/03-06-01.pdf (Jun 2003)

[14] Molnár, B., Benczúr, A., Béleczki, A.: Formal approach to modeling of modern information systems. International Journal of Information Systems and Project Management 4(4), pp. 69–89 (2016)