Formally Modelling Time into Changing University Structures

William S. Shu[1] **and Cyprian F. Ngolah**[1,2]
[1]Department of Computer Science, University of Buea, P.O. Box 63, Buea, Cameroon.
[2]Department of Electrical and Computer Engineering, University of Calgary, Canada.
e-mail: wsshu@yahoo.com, ngolahcf@yahoo.com

ABSTRACT

In order to computerise an organisation or enterprise that constantly adapts to social and technological changes over time, a practical model for time is presented that captures changes to basic components of a university in the past, the present and the future, while maintaining their temporal relations. The model allows for the evolution of its structures and activities over time, the exploration of future and hypothetical activities (forecasting) and the analysis of past activities. Data correction is also done, but without falsifying the past. This paper demonstrates how the model is used to incorporate time into the formal specification of university components, such as departments, courses and students, and the operations that modify or access them.

Keywords: Time, Change, Formal Specification, Enterprise Modelling, University Structures.

RESUME

Afin d'informatiser une entreprise ou une organisation qui s'adapte aux mutations sociales et technologiques, un modèle pratique du temps est presenté qui décrit les mutations des composants de base d'une entreprise, tout en exprimant leurs relations temporelles. Le modèle permet l'évolution de ses structures et activités au cours du temps, l'experimentation avec des activités futures ou hypothétiques (les prévisions), et l'analyse des activités passées. La rectification des saisies est faite, sans pourtant falsifier le passé. Cet article démontre comment le temps est integré dans une spécification formelle des composants d'une université, tels que les départements, les cours et les étudiants, et les opérations qui modifient ou répondent aux requêtes sur cette dernière.

Mots Clés: Temps, Changement, Spécification formelle, Modélisation de l'entreprise, Structures d'Université.

1 INTRODUCTION

The structure of an enterprise changes over time in order to adapt to business needs, social and political concerns, and technological advances. This is especially so today in the face of globalisation where the structure, mission and even survival of an enterprise or organisation is in constant but uncertain change; these changes being generally outside the control of the institutions affected.

Universities (higher education) are not exempt (Newby, 1999; Bates, 1999; Ellsworth, 1997). Computerising a university, or adapting its current computing infrastructure must allow for current and future support of teaching, research and administration (Shu and Moore, 1998) and, for its credibility, critically high standards of its records, say. This necessitates a model of time that permits current university activities to take place, past activities to be queried or verified, and future or hypothetical ones explored. Also, the apparently random changes must progress in the "correct" direction as determined by user, managerial and other intentions.

In observing the computerisation of certain universities, it was quickly noticed that a critical factor was how the influence of time was managed with respect to change, to the factors that induce the change, to on-going university activities, to changing workflow procedures, and to changing technological demands. These latter had varying priorities and were best described using separate formal logics. Furthermore, they all had to influence and be managed by the computing infrastructures of that organisation.

A desirable model of time, then, should be effectively computable (Vila, 1994; Gabbay, 1994) when integrated into the structure of a system whose entities and their relationships vary over time, and can map to logic theories/models applied to different facets and tasks of that system. The resultant system should permit a user to quickly automate the specification of new requirements, and validate them, while guaranteeing valid prior temporal assertions on the basic system. In short, time should be engineered into an evolving structure so that its use therein is flexible, extensible, consistent and mechanical.

Hence, this paper presents how time is modelled and incorporated into the specification of core operations and components needed to restructure a university, while allowing for daily university activities as well as data processing corrections. Such operations are detailed in Shu and Moore (1998). Without loss of generality, a university is used as a case study. This is for two main reasons. First, it addressed the practical concern cited above. Second, the structures of universities are reasonably well understood. Their changes are relatively slower than in most commercial enterprises, though they usually span larger time intervals. They are thus "static" enough to ease validation of the model, despite a flurry of change. At a more abstract level, the theoretical model of time to adopt is varying, not known *a priori*, and no single model would do.

In the model proposed, states [of objects] are "located" onto a time line where one may move back and forth carrying out what-if-analyses (forecasting) and/or checking or replaying past activities. The focus is not on reasoning about actions and change *per se* (philosophical) but on practically applying actions — to effect change or in spite of change — in order to obtain the intended assurances. Assorted logics (*c.f.*, Galton, 1987; Allen, 1984; Pinto, 1993; Ma and Knight, 2001) may be freely straddled, while side-stepping fundamental complications of temporal systems (*c.f.*, Vila, 1994). However, this work is biased towards possibly reified temporal logics.

Temporal activities are best described formally (Nissanke, 1997). This, together with quality requirements and corrections in a rapidly changing environment strongly suggest the use of formal methods. Formality in the specification is expressed by using the RAISE Specification Language, RSL (The RAISE Language Group, 1995) — a formal specification language that has tools for proving the correctness of a given specification.

Section 1.1 identifies a possible model for university structures that allows for change over time. Section 2 defines a working model for time by motivating duration and change. In section 3, the model is demonstrated in RSL specifications of components and operations of a university as it evolves over time. Section 4 situates the model's handling of time in relation to temporal reasoning, hypothetical evolution and enterprise modelling. Section 5 is the conclusion.

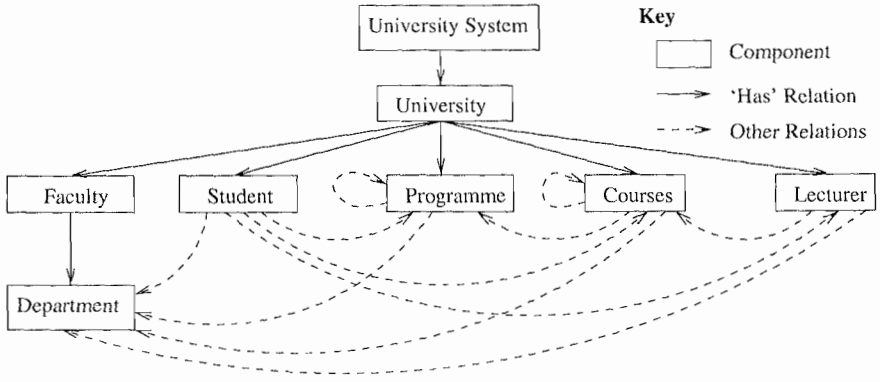

Figure 1: Relations among university components

1.1 University structures and activities

The university structure assumed abstractly has a university (administration) with faculties, faculties with departments, departments with programmes, courses, students and lecturers, and programmes with courses and students. However, in the specification model adopted, the university (structure) is conceptually viewed as the subtree "University" (Figure 1) with sub-components linked directly to it via unique identifiers (id's) by which they are referenced. Only departments are linked to it indirectly, via faculties. Other relations among components are built using their id's. Thus, they can be changed easily and dynamically. The "university system" describes the whole university as it evolves over time.

The time component over various university activities and states is highlighted. Activities of the university take place over various periods of time, likewise the persistence (states) of its structures. Thus a university, faculty or department is open to run over an extended period of time, possibly hundreds of years. It may then be closed permanently, but after letting it wind down its activities. Duration for departments may be much shorter than for faculties or universities, and their closure are sometimes temporary. Thus, the university, faculties and departments may have status opening, open, or closed.

Programmes and courses run for arbitrary periods of time so long as there are students to take them. They may run for some parts of an academic year, but not in others. They may be closed down, but must allow all students on them to graduate (or change programmes or courses) before doing so. Programmes persist for arbitrary durations. Though other durations could apply, a course normally runs for one semester, possibly more than once in an academic year.

A student is admitted to a university and is expected to enrol for periods of one academic year until he/she graduates. He or she may enrol for less than an academic year (usually a semester) if that is the minimum period needed for graduation requirements. The student may suspend one or more semesters. Also, a student may finish one degree programme, then enrol for another. He or she may be on or off courses at different times. He or she may change these courses, as well as departments and faculties, at various times in his/her stay at the university. Staff members employed teach various courses, supervise and examine students over varying time periods. A lecturer may change departments and faculties, and may leave the university for extended periods of time. He or she may also leave the university permanently.

2 A WORKING MODEL FOR TIME

2.1 Modelling duration

One should be able to capture the state of components in a university due to events that took place, are taking place or will take place in future. One would typically want to know the state of a component at a specified time or interval of time and hence what activities are possible with them.

A university evolves over time and operations (events) act on its components, which change state as a result. This change of state persists over an interval of time, which may vary from hours (*e.g.*, lectures) through months (*e.g.*, courses) to decades or centuries (departments and faculties). But the event itself is modelled as applied at an instant in

time. A user should be able to examine how hypothetical, future as well as past events unfold and analyse their outcomes in given situations.

Reckoning with future activities is useful for three main reasons. First, for planning purposes, it permits one to hold information that may be needed or scheduled later. For instance, course allocation to lecturers and course selection by students are done on a yearly basis, but a course may be taught only over one semester. One may then record information for later semesters and use or modify them as necessary. Thus, in the current semester, a lecturer does give the course and a student does take it. The allocation is modified for later ones should some other lecturer give the course or the student withdraw from it. Alternatively, the lecturer may be rescheduled to give the course in an earlier or later semester; the student may be similarly rescheduled to take the course.

Second, it permits one to explore or foresee what could happen if certain activities are carried out. This is useful in anticipating risks, or exploring the consequences of decisions when restructuring the university. Actual risk management is not part of the concern here but it must be allowed for.

Third, changes to the university are viewed as edits in an edit environment, as one would edit a program using a syntax (or structure) editor (Shu and Moore, 1998): parts of the university can be altered, provided any constraints on university components are met. This makes updates simpler because it is often easier to modify an existing version of an object than to create a new one from scratch. Also, a given edit operation may be undone or subsequently redone. In addition, for instance, a tried out activity may be moved (transferred) into the present from the future.

2.2 Past, present and future times

Time is partitioned into closed intervals that are discrete, non-overlapping and contiguous, though activities (processes) may overlap in time or may take place over non-contiguous intervals. Time "flows" from the past, through the present to the future on a time line as determined by a clock *timenow*. Activities (and their effects) which have already taken place form part of the past and cannot be changed. Activities which are taking place form part of the present. They or their effects may be altered. Most present activities eventually form

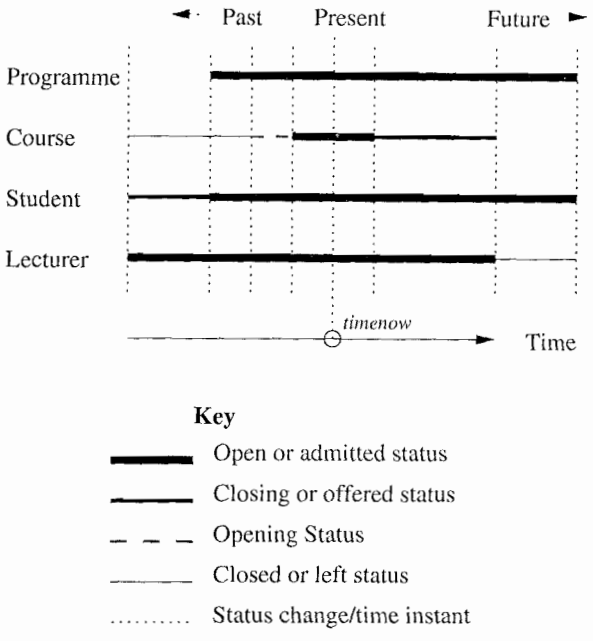

Figure 2: Change in status of object over time

part of the past. Future activities are yet to take place. Some or none of them eventually become part of the present, and perhaps the past. As an example, Figure 2 shows status changes for some components from a university structure and how their past, present and future change with *timenow*, the actual real time. For example, the status of a course is *open* (*i.e.*, running) and that of a student is *admitted* (*i.e.*, registered in the university).

Within the above categories of time (past, present, future), one may have relative categories of time. (*c.f.*, tenses in natural languages.) For example, in future past, one may seek future activities in relation to a given past. The myriad possibilities available are not examined here. Such relative activities are handled by having a "current time" with respect to which activities are reckoned: locate a current time, then locate time periods and activities relative to it. In this way, some of the complexities of tenses in temporal logic (Galton, 1987; Nissanke, 1997), such as its inabilities in handling events (changes of state) are avoided, though one can still define and use temporal operators.

Hence, tenses are constructed in a consistent but less problematic fashion than, say, in tense logic. Each activity is located w.r.t. its current time, and these times may be compared with each other or with *timenow* for the appropriate temporal ordering. Potential ambiguities of, say, relative

presents and pasts are thus resolved on a common time line or viewed as interpretations in the real world, possibly in a separate temporal system.

2.3 Time slices

For all the objects in the university (departments, students, *etc.*) the periods of time that are of interest are those over which certain operations may be performed, that have certain status (*e.g.*, *open* department), or hold certain values (*e.g.*, a student in this department). Such time intervals need not be contiguous. For example, the semesters over which a given student takes courses for a degree need not follow each other. However, disjoint time intervals are generally used for a given state or event as a way of defining their meaning (Nissanke, 1997): an interval over which an event occurred or a state was observed helps define that event or state unambiguously. Thus, one basically wants to:

- Define a continuous time interval. This is defined from fixed begin- and end-times, or by prescribing suitable constraints. Such an interval is called a *period*.
- Define a time "span" of interest. A *span* is a time-ordered sequence of non-overlapping time intervals (*periods*). It may be defined via fixed time intervals and/or constraints to be satisfied. Its intervals need not be contiguous.
- Locate an instance (point) in time, possibly within a *period* or *span*. Such an instance is called a *moment*.
- Assert if a time interval is in the past, present or future relative to *timenow* or "current time".

Time intervals — *moment*, *period* and *span* — are collectively called *time slices*. Commonly used periods such as semester, session (academic year), programme duration, can be defined in terms of suitable *time slices*.

2.4 Modelling change

In order to track real time the clock, *timenow*, can only advance in time even though in most cases one can work with logical time. Events and the change of state of objects take place on its tick. To model the past, present and future, all periods that are closed before *timenow* are taken to be in the past. Those containing *timenow* are in the present. Those intervals that start after *timenow* are in the future. This view gives one a simple way of detecting events that cannot be changed *w.r.t. timenow*. Note that given this interval view of time, a completed activity may be in the past (cannot be changed) whereas one that started before it, but has not yet finished, remains in the present. For composite activities, only those component activities that do not demand alteration of the past are allowed in the present. However, it is recognised that systems should allow for error correction and this is discussed in section 2.5.

In order to remain in the present, a time period is automatically extended so that the current value of *timenow* is always in it. This is the case, for instance, when a student has not met graduation requirements within the previously expected period. The extension corresponds to some activity in the present that remains current. Instances of future events could be hypothetical (experimental), or hold information that is needed or rescheduled later. Thus, *timenow* does not just progress into future events. The future events or states it progresses into must be explicitly rescheduled (moved) to the present and be guaranteed to reflect the desired states or events of the enterprise should *timenow* go past them. Otherwise, they must be rescheduled into the future, probably well past the current value of *timenow*, or be completely removed from the time line corresponding to that reality defined by *timenow*.

Note that removing the events or states from this time line may leave them in some other, possibly hypothetical time line with its own *timenow*. This suggests a branching time model, which is discussed in section 4.

2.5 Correcting the past

Rescheduling present and future activities simply means changing the time interval(s) associated with them and appropriate objects; past activities or past states of components cannot be changed (falsified) as indicated earlier. At the same time, in most systems [data processing] corrections are made to entries, usually within predetermined deadlines. Thus, for example, wrong marks awarded to a student who took a course may be corrected. But the fact that the student took the course must never be changed. If the student never

took the course, correcting only the wrong mark still leaves the past false. Thus, changing or falsifying the past involves deliberately or inadvertently altering events known or assumed to correspond to reality. Corrections pertain to errors introduced in the process of capturing that reality, such as data entry errors; the system built here is assumed (as is usual) to capture this reality and those who operate it do not wilfully falsify its records.

Establishing which is *changing* the past and which is *correcting* past entries quickly becomes a philosophical debate outside the scope of this work. Corrections are best seen as [allowable] edit operations (section 2.1) on the university. The past, however expressed, remains immutable; appropriate preconditions are used to preclude those edits that are seen as changing it. Deadlines for corrections to be made are reckoned by a special clock, *vtime*, that indicates when past events in the university are validated and hence cannot be changed. *vtime* cannot be later than *timenow*.

2.6 Time operations with university entities

Objects change state (value or status) when operations are carried out on them. For each *moment* over time, related states should remain consistent. Before an object changes state, one must therefore guarantee that all objects depending on its preceding state(s) remain consistent. This is done, given the above model of time, by simply making the relevant time intervals for these states part of the past, where they cannot be changed (but perhaps corrected). Equivalently, *timenow* is moved just beyond the intervals. For objects whose state persist with the present, their associated period extends with *timenow*. But this persistence-of-state may also have to be terminated at a predetermined point. For example, the interval associated with end-of-semester may not be extended just because *timenow* runs into it.

Built into the operations of objects is the automatic closure of time intervals (on state change) so that prior operations remain consistent. Also, intervals that contain *timenow* (*i.e.*, are in the present) are automatically extended to always include *timenow*, should they not have a state change. However, one may sometimes explicitly override these automatic operations.

Thus activities for a given object are recorded in basic non-overlapping time intervals. Combinations of such basic intervals may be used to reconstruct the *span* of a given activity. Similar constraints apply to intervals associated with states. In order to mechanically handle persistence of objects in the present and the future, entities in the university are also modelled with the following status values (states).

- *fixed*: A time interval in the present has fixed limits and the later limit will not automatically extend with *timenow*; by default, the interval is extended, if it would not run into a future interval.
- *xptal*: An object is an experimental or hypothetical version and must not be included in the actual system. *timenow* must not run into it. That is, present and past cannot have hypothetical objects (except in past or present relative to a future moment).

These states help ensure that in each *moment* activities of components of the university remain consistent with each other, with the status of various objects used or affected, and with the reality that is being modelled. Further states may be defined in order to capture other constraints. Examples include the temporal assertions:

- *always(n)*: for states that always hold true for the next n time units.
- *sometimes(n)*: for states that must hold true within the next n time units.

3 UNIVERSITY EVOLUTION OVER TIME

This section illustrates through an RSL (The RAISE Language Group, 1995) specification of components of a university how the model of time is used. In it, a data type for the university system as it evolves over time is given. The way clock ticks may be implemented so that the system remains consistent is also given, as well as an illustration of how the time factor is incorporated into usual university operations. Figure 3 informally explains some of the RSL symbols and terms used.

3.1 University data type

System0 below is a record data type to capture the evolution of the university system over time. Some of its *constructors* — *i.e.*, operations used to build type values in RAISE — given here

Operator	Description
$\rightarrow$, $\xrightarrow{\sim}$, $\overrightarrow{m}$, **dom** f	Total function, partial function, map, domain of function f.
$\vee$, $\wedge$, $\sim$, $\Rightarrow$, $\forall$, $\exists$	Logic operators: and, or , not, implication; quantifiers.
$^\circ$, λ, $\times$, $\equiv$	Function composition and abstraction; product; equivalence.
T-**set**	Powerset: The set of sets whose values are of type T.
T == x |y|z	Define T as a type with variants (values) x, y, z.
{| v:T • p(v) |},	Sub-type: A type of those v of type T where p(v) holds.
⟨ x | y ⟩, [w | z]	List and map comprehension. e.g., list of x's where y holds.
⟨ x, y ⟩	Enumerated list of values: x, y.
[x $\mapsto$ y]	Map of values: value x mapped to value y.
x † y	Override map value in x to that in y; add y if non-existent.
x:T $\leftrightarrow$ chg_x	Reconstructor: use chg_x to update field x of a record.
z:: x:S y:T ...	Record z has fields x:S, y:T, ... Field x has type S, *etc.*
let x = y **in** ...	Define x to be y in the expression ...
pre x, **post** y	x is precondition and y postcondition of an expression.

Figure 3: Some RSL operators

are: univs for the university (structure) at successive time intervals; timenow the system (university) clock; and vtime the validation clock. Notice that univs is a map that captures the state of a given university at a given time; chg_univ is a matching operation to update univs. Univ is a data type for the university structure. A valid university system, given by the subtype System, must have valid Univ components and must not have any experimental components (status *xptal*) at all times earlier than *timenow*. The latter is ascertained via the function embeds_a_modstatus(xptal, on, u, t).

```
type
  System0 ::
    univs : Time m→ Univ ↔ chg_univs
    timenow : Time ↔ chg_timenow
    vtime : Time ↔ chg_vtime,
  System = {| sy : System0 •
                    is_valid_System(sy) |},
  UniId
value
  is_valid_System : System0 → Bool
  is_valid_System(sy) ≡
    (∀ t : Time • t ∈ dom univs(sy) ⇒
      let u = univs(sy)(t) in
        is_valid_Univ(u) ∧
        (∀ t' : Time • t' < timenow(sy) ⇒
         ~embeds_a_modstatus(xptal,on,u,t')
        )
      end
    )
```

3.2 Clock ticks on university system

Clock ticks are conceptually simple, but require a more elaborate function to keep the system consistent from tick to tick (*c.f.*, section 2.4). On each clock tick, the university makes a transition to a new state. To do this, the state of each system component at the current tick is replicated to the new state except on the following occasions:

- The component does not exist, ceases to exist, or it contains the status *fixed*.
- The new time instant already has a component that should be part of the university system. An advantage of this approach is that the order of update (during implementation) is immaterial; the component might have been updated earlier during the same clock tick.

The function tick moves the system clock, *timenow*, to the next position in time. tick is applied within the function set_nextstate_sys. Before doing so, set_nextstate_sys calls set_nextstate_uni to move the university into its next state. set_nextstate_uni replicates current states by calling auxiliary functions to set the next state for values in each component type of the university. These functions are of the form set_nextstate_xxx where xxx corresponds to a component type (not all shown). set_nextstate_uni also reports via a boolean value if the state change was successful.

To check that constraints are met, canSet_nextstate_sys tries out a transition on a

copy of the university and tests if it is well-formed. This is done by checking the boolean value b returned by set_nextstate_uni. set_nextstate_sys also ensures that no experimental results are in the new state through embeds_a_modstatus. (Naturally, in any efficient implementation, set_nextstate_uni will be executed at most once.)

The function consistent asserts if the state(s) of a university before a given time t are consistent (*c.f.*, section 2.4). The RSL specification for assertions such as present and operations to initialise clocks are immediate and so are not shown.

```
value
  set_nextstate_sys : System ~→ System0
  set_nextstate_sys(sy) ≡
    let
      t = timenow(sy), u = univs(sy)(t),
      (u', b) = set_nextstate_uni(u, t),
      sy' = chg_univs(univs(sy) †
                      [t + 1 ↦ u'], sy)
    in
      tick(sy')
    end
    pre canSet_nextstate_sys(sy),

  canSet_nextstate_sys : System → Bool
  canSet_nextstate_sys(sy) ≡
    let
      t = timenow(sy),  u = univs(sy)(t),
      (u', b) = set_nextstate_uni(u, t)
    in
      b ∧
      ~ embeds_a_modstatus(xptal,on,u,t+1)
    end,

  /* return next state and validity status */
  set_nextstate_uni: Univ × Time → Univ0 × Bool
  set_nextstate_uni(u, t) ≡
    let u1 = set_nextstate_facs(u, t) in
      if is_ok_uni(u1) then

                  ⋮

        let u6 = set_nextstate_students(u5,t) in
          if is_valid_Univ(u6, t+1) then
            (u6, true)
          else
            (u5, false)
          end
        end
                  ⋮
      else
        (u, false)
      end
    end,

  /* true if valid univ in partial update */
  is_ok_uni : Univ0 → Bool,

  set_nextstate_students :
      Univ × Time → Univ0
  set_nextstate_students(u, t) ≡
    let sm =
      [ sid ↦ [ t + 1 ↦ s ] |
        sid : StuId, s : Student •
          isin_uni(sid, u, t) ∧
          ~ isin_uni(sid, u, t + 1) ∧
          s = students(u)(sid)(t) ∧
          /* true if status fixed is off */
          has_modstatus(fixed,off,sid,u,t)
      ]
    in
      chg_students(students(u) † sm, u)
    end
value
  /* true if university is consistent
      *before* time given */
  consistent : System × Time → Bool
  consistent(sy, t) ≡
    ( ∀ t', t1 : Time • t' < t ∧t1 < t ∧
        t1 ∈ dom univs(sy) ⇒
        is_valid_Univ(univs(sy)(t1), t')
    )

value
  /* timenow tracks current time via ticks */
  tick : System → System
  tick(sy) as sy'
      post timenow(sy') = timenow(sy)+1

axiom ∀ sy : System • time(sy) ≤ timenow(sy)
```

3.3 Incorporating time constraints

Time constraints are added to operations specified on university components by defining operations (usually of the same name) on the university system structure, System, and adding constraints to their corresponding preconditions.

Preconditions asserted at the system level of the university are done through the function canEdit_sys. canEdit_sys checks that an operation

is not done in the past and uses canCorrect_sys to assert if past entries can be validated.

Other operations at the system level of the university are essentially those at the Univ level with time constraints added. For example, add_stu_uni below applies the corresponding operation at the Univ level but its precondition, canAdd_stu_uni, additionally applies canEdit_sys.

```
value
  /* precondition asserted at system level */
  canEdit_sys : System × Time → Bool
  canEdit_sys(sy, t) ≡
      (∼ past(sy, t) ∨ canCorrect_sys(sy, t)),

  /* precondition to correct unvalidated events*/
  canCorrect_sys : System × Time → Bool

value
  /* add students to univ. at system level */
  add_stu_uni : StuId ×
       Student × System × Time ~→ Univ
  add_stu_uni(sid, s, sy, t) ≡
    let u = univs(sy)(t) in
      add_stu_uni(sid, s, u, t)
    end
    pre canAdd_stu_uni(sid, sy, t),

  canAdd_stu_uni :
       StuId × System × Time → Bool
  canAdd_stu_uni(sid, sy, t) ≡
    let u = univs(sy)(t) in
      canAdd_stu_uni(sid, u, t) ∧
      canEdit_sys(sy, t)
    end
```

3.4 Incorporating time into operations

The student is represented by the type StuType which associates with a unique identifier of type StuId the information about a student as it evolves over time. StuId is used to identify the student within other objects of the university. Details held on a student are in the record data type Student. *c.f.*, System in section 3.1.

The function is_valid_Student defines valid state transitions for student records. For instance if a student has status *left*, he or she cannot be on any course. With any status other than *offered*, he or she must belong to a department (and hence, a faculty by the first implication).

```
type
  /* Student is a person who takes courses/
     programmes, is in a unique dept, has
     course marks, and other details. */
  StuType = StuId ~m~→ Time ~m~→ Student,
  Student0 ::
    status : StuStatus ↔ chg_status
    modstatus : ModStatus-set ↔ chg_modstatus
    fac : FacId ↔ chg_fac
    dep : DepId ↔ chg_dep
    programmes : PgmId-set ↔ chg_programmes
    results : ResType ↔ chg_results
    etc : StuOther ↔ chg_etc,
  StuStatus == offered | admitted | left,
  StuOther,
  Student = {| c : Student0 •
                    is_valid_Student(c) |},

  /* course marks */
  ResType = CseId ~m~→ Result,
  Result

value
  /* courses by student */
  courses : Student0 → CseId-set
  courses(s) ≡ dom results(s)

value
  /* valid stud with faculty must have dept */
  is_valid_Student : Student0 → Bool
  is_valid_Student(s) ≡
    (fac(s) = nil_FacId ⇒ dep(s) = nil_DepId) ∧
    let st = status(s) in
      (st = left ⇒ courses(s) = {}) ∧
      (st ∈ {admitted, left} ⇒
            dep(s) ≠ nil_DepId) ∧
      (courses(s) ≠ {} ⇒ st = admitted)
    end
```

The structure of operations to register or remove a student into/from a programme is illustrated below. The given time, t, is used in expressions such as [sid ↦ [t ↦ s′]] in change_stud_pgm to identify the state of the student at time t. (Here, student sid has multiple states, one s′ for each t.) change_stu_pgm enrols a student into the programme. It adds the programme id to the set of programmes the student takes via the reconstructor functions chg_programmes. Its precondition canChange_stu_pgm ensures that the student and the programme are in deed of the university, he or she is duly admitted but not already on the programme,

and he/she has all programme prerequisites. Also, the programme must be running (given by the conjunct ∼is_status(closed, p)).

del_stu_pgm removes a student from a programme by removing the programme id via the reconstructor function chg_programmes. The precondition canDel_stu_pgm ensures that the student was in deed on the programme in the university. To transfer a student from one programme to another, del_stu_pgm may be used to remove him or her from the old programme and change_stu_pgm used to include the student into the new one.

```
value
  /* add stud to programme; must have prereqs */
  change_stu_pgm : StuId ×
      PgmId × Univ × Time ~→ Univ
  change_stu_pgm(sid, pid, u, t) ≡
    let
      s = students(u)(sid)(t),
      p = programmes(s),
      s' = chg_programmes(p ∪ {pid}, s)
    in
      chg_students(students(u) †
                   [ sid ↦ [ t ↦ s' ] ], u)
    end
    pre canChange_stu_pgm(sid, pid, u, t),

  canChange_stu_pgm : StuId ×
      PgmId × Univ × Time → Bool
  canChange_stu_pgm(sid, pid, u, t) ≡
    isin_uni(pid, u, t) ∧ isin_uni(sid, u, t) ∧
    let s = students(u)(sid)(t),
        p = programmes(u)(pid)(t)
    in
      pid ∉ programmes(s) ∧
      is_status(admitted, s) ∧
      ∼ is_status(closed, p) ∧
      pass_pgm_preqs(sid, pid, u, t) ∧
      (has_degree_programmes(sid,u,t) ⇒
            ∼ is_degree_pgm(pid, u, t)
      ) ∧
      (is_degree_pgm(pid, u, t) ∧
       ∼ has_degree_programmes(sid,u,t) ⇒
       has_degree_programmes({pid},
                   dep(s), fac(s), u, t)
      )
    end,

  /* remove student from programme */
  del_stu_pgm :
      StuId × PgmId × Univ × Time ~→ Univ
  del_stu_pgm(sid, pid, u, t) ≡
    let
      u' = if t ≥ 1 ∧
           canSet_modstatus(fixed,on,sid,u,t−1)
        then
          set_modstatus(fixed,on, sid, u, t−1)
        else
          u
        end,
      s = students(u')(sid)(t),
      s' = chg_programmes(programmes(s) \
                                {pid}, s)
    in
      chg_students(students(u') †
                   [ sid ↦ [ t ↦ s' ] ], u')
    end
    pre canDel_stu_pgm(sid, pid, u, t),

  canDel_stu_pgm : StuId ×
      PgmId × Univ × Time → Bool
  canDel_stu_pgm(sid, pid, u, t) ≡
    isin_uni(sid, u, t) ∧ isin_uni(pid, u, t) ∧
    let s = students(u)(sid)(t),
        p = programmes(u)(pid)(t)
    in
      pid ∈ programmes(s) ∧
      ∼ is_status(closed, p)
    end
```

The specifications, as presented, facilitate formal verification (proofs), automatic generation of their code, and the subsequent addition of new operations as the university evolves. Also, they allow for code/algorithm optimisation in specification refinements, while avoiding bias to specific implementations. For instance, preconditions, such as canAdd_stu_uni(sid,s,u,t), that are guaranteed to be true (elsewhere in the executed code) are not tested in the following code fragment to enrol a new student onto a [degree] programme:

```
value
  /* Register student into programme of univ */
  registerDegree_stu_pgm : Univ × Time → Univ
  registerDegree_stu_pgm(u, t) ≡
    let s : Student,
        fid = get_FacId(),  /* get id from input */
        did = get_DepId(fid),
        pid = get_PgmId(did, fid),
        sid = unique_stu(), /* new, unique id */
```

```
    u' = add_stu_uni(sid, s, u, t),
    u'' = add_stu_dep(sid, s, u', t),
  in
    change_stu_pgm(sid, pid, u'', t)
  end
```

4 DISCUSSION

The temporal dimension of information — the change of information over time — is imperative in considering change as well as how to integrate it computationally within any system (Nissanke, 1997; Vila, 1994). Unfortunately, logic systems are mostly focused on philosophical issues of time, the need for expressiveness and power in well-formed mathematical theories of time, and contentions with contradictions to acceptable realities; for computing applications, effective computability of models and efficiency are additional requirements (Gabbay, 1994). Thus, event calculus (Kowalski and Sergot, 1986) focuses on events and situation calculus (McCarthy and Hayes, 1987; Pinto and Reiter, 1993) stresses on the actions that induce change. Temporal logics (Allen, 1984; Shoham and McDermott, 1988; Galton, 1987) incorporate time as a mathematical sort in a multi-sorted logic for states, and reification (Ma and Knight, 2001) build temporal assertions and structures over other logics.

The proposed model uses time points and time intervals (*c.f.*, moment and duration) and structures time as discrete, bounded (towards the past) and linear. The basic system is sound but need not be complete since it does not make inferences from a temporal reasoner (Vila, 1994).

Change [of state] is discrete, and causality is only defined or inferred by constraints (requirements) on the university system, and is usually derived from atemporal requirements. These are finite, computable and refutable. Real-time computing issues may not arise since "low-level" concurrency (*i.e.*, at processor or transaction level speeds) is encapsulated as a specific task that provokes a change of state on the time line.

4.1 Applying the model

Time is captured through a clock whose ticks are used to trigger events and hence change the state of components in a university. A time argument defines the desired moment at which operations and assertions are carried out. Preconditions for objects, then, additionally ensure that an otherwise valid operation does take place at the correct point in time. Extension to possibly non-contiguous intervals of time thus becomes conceptually trivial.

Considering states at instants in time permits temporal constraints to be consistently checked and possibly refined to time intervals in ways that a given problem would naturally demand. Using *timenow* to partition intervals into past, present and future provides a simple, robust and sound approach to avoid temporal inconsistencies and dilemmas, as would easily arise in tense logics, say, and yet permit other layers of logics (*e.g.*, reification) and applications to be consistently built over it. Thus, a user or person moves back and forth along the time line making his or her assertions in context.

Abstractly, one queries a "database" into which temporal information is coded. All manipulations, whatever the logic system, must respect the underlying time line (ontology). Any inconsistencies arise from viewpoints over this "database" of time-varying states, rather than inconsistencies in its updates and evolution.

This is deliberate. In streamlining temporal activities onto a time line and with the ability to pick and mix temporal ontologies, contexts and priorities may have to be defined. Thus, the logical, intuitive perception of time seems to break down: past, present and future activities may overlap in time and a present activity may have started before a past one! The viewpoint used guarantees that events "flow" in order, but states may be accessed in an atemporal fashion. Hence the *viewpoint anormaly*.

Basic temporal elements are thus fused into atemporal activities but in a simple way. Arbitrary levels of temporal complexity, possibly through pure temporal expressions of varied temporal logics, may be added in a modular fashion. Theoretically, a map from these to the underlying logic may be necessary.

4.2 The frame problem

For computational efficiency, most things should not change, even when changes induce other changes. The frame problem in logic is to determine those things that do not change. In the model, the past is immutable and the future is

changeable only via the present. In the present, states persist *de facto*, unless explicitly changed by an event or through implications arising from such events. Such implications are derived from constraints in the the formal specification of the system. Things that change for a given event are relatively few and often do so in algorithmically flexible ways (*c.f.*, section 3.2). Specifying every relevant condition for change (the qualification problem; Shoham and McDermott, 1988) is implicit in the problem specification. Any condition that is omitted is a specification validation issue which could be redressed as if it were a new change to the system. However, elaborate general solutions for these calls for non-monotonic reasoning (*c.f.*, Shoham, 1987) and temporal constraint satisfaction (*c.f.*, Schwalb and Vila, 1998). These are beyond the scope of this article.

4.3 Hypothetical evolution in time

Allowing for possible hypothetical evolution of university states suggests a formulation using situation calculus to capture all possible ways in which events can unfold (Fox *et al.*, 1998; Pinto and Reiter, 1993). In this work, only one branch of the many hypothetical alternatives after a given action is selected to describe the evolution of the world as it actually unfolds. The overarching concern is to capture events and states in the correct time order. Performing situation analysis is more naturally handled under applications (operations) built over the basic model. Thus, analysing future events (before they occur) is useful in forecasting/planning; analysing events that might have occurred is useful in fault-detection and for performance improvements on the system. These are also useful in software validation and in eliciting rules that govern change in the system implemented.

Status values such as *fixed* and *xptal* and related assertions on them (*e.g.*, has_modstatus) are empirical mechanisms to enforce state persistence and transitions. Thus, persistence can be maintained in both real and hypothetical situations, and the recreation of states through transitions from hypothetical to real.

Using multiple versions of possibly distinct universities naturally suggests a branching time model (*c.f.*, Fox *et al.*, 1998) even though consistency issues rather than process concurrency is our driving concern. The demands for a branching time model are circumvented by considering universities as objects in an edit environment, each with its own clock. Subcomponents transferred among them need not carry over prior temporal constraints. Also, a temporal relationship between separate universities (or components) within the edit environment is not required: two universities need not evolve in related time frames, and a separate module for branching time temporal logic could always be added for specific applications.

4.4 Situating the work

Institutions have addressed the issue of adapting higher education to technological change, to the size and diversity of student populations, to courses offered and their delivery mechanisms, and to the type of staff needed (Bates, 1999; Ellsworth 1997). But change is generally viewed in terms of policies (aims) and services rendered, and rarely from the point of view of the technology handling it. Our viewpoint is novel in this regard.

Adapting a university for change at best comes under enterprise modelling (*c.f.*, Petrie, 1992 ; Vernadat, 1996; Ojo and Janowski, 2001), where one conceptualises a working model for his or her business (here, higher education) and related engineering (business process reengineering). That is, it calls for a careful examination of the enterprise's resources, the activities and processes that operate on the resources, how automated and manual processes integrate into a working system, and how priorities and scheduling are defined for effective use of its resources. It calls for a re-appraisal of the university's current processes and procedures so that it rapidly and fully takes into account, and presents, the implications of change, from risks to advantages (Reeder, 1994 ; Keller, 1983).

In the face of change then, one needs to analyse, build and rebuild such enterprises as engineering artifacts (*c.f.*, Brown and O'Sullivan, 1995; Glasson *et al.*, 1994). In other words, a semantic (theoretical) basis and empirical methods to manage change within a university structure have to be developed. The theoretical basis depends on formality in the temporal notions of objects. In this work, the RAISE methodology (The RAISE Language Group, 1995; The RAISE Method Group, 1995) provides a methodological framework as well as formal description tech-

niques to use. Empirically, change is addressed by putting together core operations to define new operations and queries. Another empirical novelty includes mechanically (syntactically) addressing conceptual and semantic issues; for example, status values is used to capture persistence and hypothetical evolution.

Conceptually, the technology rather than conventional temporal logic theories defined the basic structure of time used. It allowed for various temporal logics to be built, used and changed over it in a flexible, sound and consistent manner — even when completeness of the underlying logic system cannot be guaranteed.

4.5 Further work

The work so far shows how time may be incorporated into a changing university structure. However, further work still has to be done on defining operations for forecasting (what-if analyses) and for carrying out validation and control checks on its evolution over time.

The specific limitations in the map between other logics and the basic model is currently implicit in specification requirements. However, more explicit guidelines for maps may be desirable.

Notions in situation calculus and the branching time model can be used to enhance forecasting, as well as any time-ordered updates on university components. Also, the distinction between changing the past versus correcting it is still to be explored in depth.

In the model developed, future events and experimental results are recorded and finally incorporated into the university system as *timenow* runs past them. An alternative view worth exploring is to have a *plan* for future events or intended operations. All valid operations of the plan that can be fulfilled are carried out on appropriate clock ticks.

5 CONCLUSION

The structures and activities of an organisation or enterprise change over time as it adapts to technological and social changes. Computerising such an organisation necessitates a model of time that permits current operations to take place, past activities to be verified, and future ones explored.

Using the university as an example, a time representation was chosen that uses a clock whose ticks signal when to match university activities and states to those in the real world. Operations are carried out and enterprise components persist in a given state over intervals of time, but with the ability to observe university activities or states at other moments.

Hence a single view of time is adopted with the past, present and future defined as intervals that are before, including or after the current time, and with consistent transitions among them. A special clock, *timenow*, corresponds to the moment "now" in the real world. Another special clock, *vtime*, permits one to make corrections to the records of past events, as is expected in data validation, but without changing or falsifying the past. RSL specification components were then given to illustrate how a university may evolve in the model of time and how typical operations may be adapted to incorporate time constraints.

6 ACKNOWLEDGEMENTS

Many thanks to Chris George and Richard Moore for various suggestions, supervision of the work, improving on tools used and performing consistency checks on the specifications. The authors wish to thank all at UNU/IIST, at University of Buea, and beyond for making this project and their stay at UNU/IIST possible.

7 REFERENCES

ALLEN, J., (1984). Towards a General Theory of Action and Time. Artificial Intelligence 23, pp. 123 - 154.

BATES, A. W. (1999). Restructuring the University for Technological Change. In: (eds) Brennan, J., Fedrowitz, J., Huber, M. and Shah, T., What Kind of University? International Perspectives on Knowledge, Participation and Governance. Society for Research in Higher Education and The Open University Press, Buckingham.

BROWNE J. and O'SULLIVAN, D. (1995). Re-Engineering the Enterprise. Chapman and Hall, London.

ELLSWORTH, J. B.,(1997). Technology and Change for the Information Age. The Technology Source (http://Horizon,unc.edu/TS/).

FOX, M. S., BARBUCEANU, M., GRUNINGER, M. and LIN, J. (1998). An Organizational Ontol-

ogy for Enterprise Modeling. In: Prietula, M. J., Carley, K. M., and Gasser, L. (eds), Simulating Organizations: Computational Models of Institutions and Groups. The MIT Press, Menlo Park California, pp. 131-152.

GABBAY, D. M., HODKINSON, I. and REYNOLDS, M. (1994). Temporal Logics, Mathematical Foundations and Computational aspects. In: Oxford Logic Guides 28 (volume 1). Clarendon Press, Oxford.

GALTON, A. (1987). Temporal Logic and their Applications. Academic Press, London.

GLASSON, B. C., HARWRYSZKIEYCZ, I. T., UNDERWOOD, B. A. and WEBER, R. A. (eds) (1994). Business Process Re-Engineering: Information Systems Opportunities and Challenges. Proceedings of the IFIP TC8 Open Conference on Business process Re-Engineering: Information Systems Opportunities and Challenges, Queensland Gold Coast, Australia, 8 - 11 May. North Holland, 1994.

KELLER, G. (1983). Academic Strategy: The Management Revolution in American Higher Education. John Hopkins University, Baltimore.

KOWALSKI R. A. and SERGOT M. J. (1986). A logic-based calculus of events. New Generation Computing 4, pp. 67-95.

MA, J. and KNIGHT, B. (2001). Reified Temporal logics – A survey. Artificial Intelligence Review, 15(3), pp. 189 - 217.

McCARTHY, J. and HAYES, P.J. (1987). Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: ed: Ginsberg, M. L., Readings in Non-monotonic Reasoning, Kauffman, Los Altos, CA, pp. 26-45.

NEWBY, H. (1999). Higher Education in the Twenty-first Century – Some Possible Features. Perspectives 3(4), pp. 106-113.

NISSANKE, N. (1997). Realtime Systems. Prentice Hall Series in Computer Science, Prentice Hall, London.

OJO, A. and JANOWSKI, T. (2002). Formalizing Production Processes. In: Hung D. V., George, C. and Janowski, T. (eds), Specification Case Studies in RAISE. Springer Verlag Series in Formal Aspects of Computer and Information Technology.

PETRIE Jr., C. J. (ed) (1992). Proceedings of the First International Conference Enterprise Integration Modeling. MIT Press, Cambridge, Massachusetts.

PINTO, J. and REITER, R. (1993). Temporal Reasoning in Logic Programming: A Case for the Situation Calculus. In: Proceedings of the 10th International Conference on Logic Programming, Budapest. MIT Press, Cambridge Massachusetts, pp. 203-221.

REEDER, J.(ed) (1994). Business Process Redesign: For Higher Education. National Association of College and University Business Officers.

SCHWALB, E. and VILA, L. (1998). Temporal Constraints: A survey. Constraints 3(2-3), pp. 129-149.

SHOHAM, Y. (1987). Non-monotonic logics: Meaning and Utility. In: Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI'87), Milan, Italy pp. 388-393.

SHOHAM, Y., and McDERMOTT, D. (1988). Problems in Formal Temporal Reasoning. Artificial Intelligence 36(1), pp. 49-61.

SHU, W. S. and MOORE, R. (1998) Dynamically Reconfigurable University: Formally Specifying Core Operations. UNU/IIST Technical Report No. 146.

The RAISE Language Group. (1995). The RAISE Specification Language. BCS Practitioner Series, Prentice Hall, London.

The RAISE Method Group. (1995). The RAISE Development Method. BCS Practitioner Series, Prentice Hall, London.

VERNADAT, F.B. (1996). Enterprise Modeling and Integration: Principles and applications, Chapman and Hall, London.

VILA, L, 1994. A Survey on Temporal Reasoning in Artificial Intelligence. AI Communications 7(1), pp. 4-28.

Received: 25/01/2003
Accepted: 21/05/2004