Paradigm integration in a specification course

Manuel A. Martins
CIDMA, Dep. Mathematics
Universidade de Aveiro
martins@ua.pt

Alexandre Madeira, Luís Soares Barbosa, Renato Neves
HASLab - INESC TEC
Univ. Minho
madeira@ua.pt, lsb@di.uminho.pt, nevrenato@gmail.com

Abstract

As a complex artefact, software has to meet requirements formulated and verified at different levels of abstraction. A basic distinction is drawn between behavioural (dynamic) and data (static) aspects. From an educational point of view, although disguised under a number of different designations, both issues are usually present, but kept separated, in typical Computer Science undergraduate curricula. It is often argued that they tackle orthogonal problems through essentially different methods. This paper explores an alternative path in which students progress from equational to hybrid specifications in a uniform setting, integrating paradigms, combining data and behaviour, and dealing appropriately with systems evolution and reconfiguration.

1 Introduction

Motivation. Fundamental infrastructures of modern societies, including those related to finances, health services, education, energy and water supply, are critically based on information systems. Moreover, our way of living depends on software whose reliability is crucial for our own work, security, privacy, and quality of life. This explains why the quest for programs whose correctness could be established by mathematical reasoning, which has been around for a long time as a research agenda, seems to be finally emerging as a key concern in industry. Companies are becoming aware of the essential role played by formal logic. The growing demand for highly skilled professionals who can successfully design complex systems at ever-increasing levels of reliability and security, places serious challenges to higher education and training programmes.

This paper aims at contributing to the on-going debate on how a rigorous discipline of software development, with sound, mathematical foundations can be successfully integrated in the curricula of undergraduate degrees in Computer Science. In the sequel we discuss the *rationale* and design of one semester course, entitled *Introduction to Software Specification*, to be placed in the last semester of a three-year long undergraduate degree. The course is expected

- To invite Software Engineering students to revisit elementary mathematical concepts which constitute a background for formal specification and development methods. Such notions come essentially from Logic — including *e. g.* those of a signature, sentence, axiom, model and satisfaction. Although most students have already been exposed to them (typically in a previous course on Logic or Discrete Mathematics), they are re-visited with a clear application purpose: that of formally expressing software requirements.
- To help students building up correct intuitions on the *data - behaviour* symmetry when describing software and introduce them to suitable conceptual tools to handle them. The latter cover *abstract data types* (building on previous experience with data structures in programming languages), on the one hand, and *transition systems* (building up on a first course on automata and formal languages).
- To motivate for the role of (formal) specifications in Software Engineering and illustrating their use through a notation and framework which is kept close to standard mathematical notation as far as possible and has reasonable tool support.

Context. The need for such a course arose in a very concrete context: the re-organisation of university degrees motivated by the implementation of the *Bologna Agreement* in the European Union, a process which in Portugal is currently going through its first nation-wide assessment and review. In Portugal, the *Bologna Agreement* led to the split of traditional 5-years courses into separate Bachelor (3 years) and Master (2 years) degrees. Bachelor degrees are expected to provide large-spectrum training in standard Computer Science areas (e.g. *programming*, *databases*, *computer networks*, *compilers*, etc.) as well as basic skills in Mathematics and Engineering. The latter includes project planning and entrepreneurism, among others. Master degrees, on the other hand, are usually intended to offer vertical specializations.

At Universidade do Minho, the Master degree in Soft-

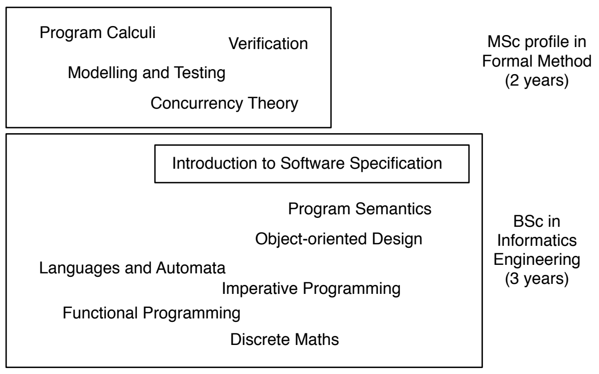


Figure 1. The context.

ware Engineering is organised around a number of vertical specialisations in specific domains of Computer Science, for example *Artificial Intelligence*, *OLAP and Data Mining*, *Computer Graphics* or *Formal Methods in Software Engineering*. Each of these specialisation profiles sum up to 30 ECTS and includes 4 regular courses plus a integrated laboratory with a year long engineering project.

The *Introduction to Software Specification* course discussed in this paper is placed in the final semester of the Bachelor level, and offered as an option to students who intended to pursue a specialisation in Formal Methods. Figure 1 shows this context, highlighting the subjects lectured in each level which are most relevant as either an antecedent or a follow-up of this course. The effort put on re-designing University education in a new *3+2 framework* entailed the need to re-think undergraduate degrees in order to concentrate in a unique course each core curricular area, with its essential concepts and methods. In what Formal Methods are concerned this means to design an larger spectrum course able to provide a broad overview of the area. Several experiments were implemented along this first 5 years *post-Bologna* experimental period. Most of them chose to focus on a single framework/method and limit explicitly its scope to hopefully go deeper in the syllabus. Model-oriented development (e.g. in VDM), algebraic specification (e.g. with OBJ), verification (typically in a model-checking version, e.g. with SPIN) or concurrency theory (e.g. with mCRL2) are examples of such courses.

This paper reports on a *second thought* on this experience and discusses an alternative approach to the introduction of formal methods in the Software Engineering curriculum. The pedagogical intuition was to provide undergraduates with a solid exposition to a formal framework/method, motivating for the relevance of the area and training in its methodological principles. Later, at a MSc stage, students would have the opportunity to choose a specialisation in this area. A questionnaire to young professionals who have recently completed their undergraduate studies pointed out, as reported below in section 2, the relevance of tool support as well as of methodologies able to tackle in a common framework different aspects of software design.

The course. Our starting point was, therefore, the following observation: software is a complex artifact, which deals with a multitude of different concerns and has to meet requirements formulated (and verified) at different abstraction levels. A basic distinction, whose didactical relevance can be easily appreciated, is drawn between *behavioural* and *data* requirements. While *processes* are dynamic and active, *data* is static and passive. Typically, the emergent behaviour of a software system is determined by the concurrent execution of several processes which exchange data in order to influence each other's behaviour. From an educational point of view, although disguised under a number of different designations, both approaches used to be part of a typical Computer Science undergraduate curriculum: abstract behavioural structures are typically studied in a Process Algebra course (often on top of a previous course on languages and automata), while abstract data structures are covered in algebraic specification courses. The latter are typically concerned with the concept of *abstract data type* [11]. Even if a number of attempts to integrate data and behaviour specifications do exist, as in LOTOS or mCRL2 It is often argued that those specifications tackle orthogonal problems through essentially different methods.

To design the course we proceed by identifying both a class of problems to motivate students from the outset, and a formal framework.

For the former we chose to address requirements for *reconfigurable systems*. Often, and most typically in service-oriented applications, what a software component may offer at each stage depends on its own evolution and history. Software components act as *evolving structures* which may change from on mode of operation to another, entailing corresponding updates in what counts, at each mode or stage, as a valid description of their behaviour. For example, a component in a sensor network may be unable to restart a particular equipment if in an alarm stage of operation, but not in a normal one. On the other hand, the way it computes the result of sensoring a number of hardware control devices may change from one mode to another (changing, for example, the pallete of weights used to compute a weighted sum of measurements). This lead us to consider not only standard *algebraic specification* techniques to specify services or components interfaces, but also *modal* and *hybrid* languages to express the systems' evolution and reconfiguration.

For the choice of a formal framework we considered the *mathematical structures* suitable to model software systems; the *languages* in which such models can be specified and, finally, the *relationship* between the (semantic) structures and the (syntactic) formulation of requirements as sentences in the specification language. A quite canonical, even if not very popular, formulation resorts to the notion of an *institution* [2] which, as an abstract representation of a logical system, encompasses syntax, semantics and satisfaction, and provides ways to relate, compare and combine specification logics. It is remarkable how institutions not only provide a standard, mathematically solid basis for the approach discussed here, but also pave the way to suitable tool support. The HETS tool [8] is the working platform for the course.

Paper structure. In the sequel we explore such an alternative approach to formal methods introduction in a Software Engineering curriculum. Section 2 collects the result of an assessment questionnaire in which the design of the new course was partially based. The course *rationale* is introduced in section 3 and illustrated through the discussion of a case-study from the last course unit in section 4. Finally, section 5 concludes.

2 Curriculum assessment

In order to assess the impact of formal methods education at undergraduate level, a questionnaire was given to a sample of 25 young professionals who have recently completed a MSc degree in Informatics at Universidade do Minho, Portugal. The sample was balanced in terms of gender, age (ranging from 26 to 34 years), and background. All of them are currently employed.

Students were asked to focus exclusively on their undergraduate studies when answering the questionnaire. A working assumption was that the temporal gap between their conclusion of an undergraduate degree and the moment in which they participate in this research (of at least 2 years, but greater than that in most cases) would provide the necessary critical distance.

The questionnaire consisted of 8 open questions. The first two identify academic background and professional status; the other six intended to assess their own undergraduate education in formal methods and the impact it has (or has not) in their current activities. This study has, of course, limited statistical relevance. But it does provide an indicator with respect to a number of topics addressed by almost all the responses, even if with different emphasis.

A first question asked for an enumeration of both positive and negative aspects of previous formal education in formal methods. On the positive side the most consensual answers mentioned the development of abstract reasoning and the acquisition of skills to *"become more agile in solving hard problems"* and methodic in programming, avoiding, as an answer puts it, *"developing software by patching solutions"*. A curious answer from a successful software engineer put it plain: *"I'm not afraid of working with symbols and formulas, like most software engineers I encounter are"*. On the negative side, most answers refer the impact of a steep learning curve, especially to less mathematical-oriented students; the lack of (explicit) interrelation among different components of the formal methods curriculum; and the significant gap with respect to software testing.

The questionnaire explored known difficulties and asked for concrete suggestions for curricular improvement. Most frequent suggestions were:
- Increased tool support;
- Project work supported by good case studies;
- Bigger emphasis on applicational areas;
- Methods able to cope with all aspects of systems' requirements (avoiding a multitude of notations and tools);
- A closer connection to classical courses on logic and algebra.

The questionnaire also brought evidence of the relevance of formal methods education to subsequent studies or professional practice. The positive impact is unanimously referred, even if most people are not using formal methods in their professional activity. The answers are, in our opinion, quite expressive: *"It gave me (and still gives) the agility to approach a problem in a methodic way"* or *"the algebraic specification skills play a central role in my day-to-day research"*. An engineer employed by a big company stresses *"I had very few opportunities to apply a full-blown formal method in my professional practice. Most of the times I did apply fragments of some formal methods to small things, such as data models for instance. However, I find that the mindset I developed during my studies helps me every day to deal with complex problems to apply the right amount of abstraction and even to express myself better."*

3 Designing an alternative course

The course rationale. The course was designed based on the authors' own lecturing experience and the analysis of the questionnaires mentioned above. The *rationale* was to explore a uniform framework for specifying system's requirements either *functional* (i.e. relative to the meaning of individual services or operations) or *behavioural* (i.e. relative to its overall evolution and reaction to external stimulus), and emphasise a strong connection between *modelling* and *verification*.

The course has a standard typology: a lecture per week (1 hour), an exercise class devoted to pen-and-pencil resolution of exercises previously proposed and their discussion (2 hours) and a laboratory session with the HETS system (1 hour). Students work on groups of two elements.

The course develops around a triangle whose vertices and repeatedly revisited: *models*, *languages* in which such models and their properties are expressed and the *satisfac-*

tion relation between them which enables property verification and design assessment. Another methodological option concerned the adoption of a *generic framework*, in which progressively more elaborated requirements could be represented, in contrast to one with a narrower scope or clearly oriented to a particular specification style. This has the advantage of focusing students and enhancing their ability to work at higher abstraction levels.

This favoured the choice of an institutional approach and the HETS framework. HETS is able to handle specifications in different logical systems and to relate them through the relationships connecting the underlying logics. Actually, the notion of an institution was introduced by Goguen and Burstall in the late 1970s (with the seminal journal paper [2] being printed rather late) in response to the population explosion of specification logics. Its original intention was to provide an abstract framework for specification of, and reasoning about, software systems.

The course structure. As mentioned above, the course targets *reconfigurable* systems. They involve components which may evolve in time through a number of different stages or modes of operation, to which correspond different configurations of the services made available through its interface. The envisaged teaching/learning process develops around three specification stages: *algebraic*, *moda*l and *hybrid*. The idea is to cover the whole spectrum of basic specification logics in three course units, all of them sharing HETS as the common tool support. A fourth unit in the syllabus explores a number of case-studies in the project of reconfigurable systems. The course illustration in section 4 is taken from this last unit. Before that, let us review the *rationale* under each of them.

The algebraic stage. At a first stage each system *configuration* is specified axiomatically as a "stand-alone" *algebraic theory*; its model being a concrete algebra satisfying such a theory. Component's functionality is therefore given in terms of input-output relations modeling operations on *data*. This stage covers the classical stuff on algebraic specification, namely the concepts of *signature*, *sentence*, *equation* and equational reasoning, *model* and *satisfaction of an equation*. The envisaged learning outcome is the ability to master these concepts and capturing informal requirements about component's functionality by defining a (syntactic) *universe of discourse* and formulating properties as axioms.

The modal stage. The second stage emphasises the *reactive* nature of the systems at hands. Component's evolution is modelled by a transition system: a configuration changes in response to a particular event in the system. Modal logics are introduced as specification languages for state transition systems. Modal formulas are evaluated inside such systems, at a particular state, and modal operators disclose access to information stored at other states accessible from the current one via a suitable transition. The main learning outcome is to make students familiar with the modal framework and the meaning of modalities as a language to specify transition structures.

The hybrid stage. The third stage starts with a crucial observation: functional and transitional behaviour are strongly interconnected in practice as the functionality offered by the system, at each moment, may depend on the stage of its evolution. This entails the need for
- enriching the basic modal language with the ability to refer to *individual* states, regarded as possible system's configurations or modes of operation;
- distinguishing *global* behaviour (in the underlying transition system) from *local* behaviour expressed, at each state, by a particular specification.

The first requirement leads to the introduction of *nominals* as explicit references to specific states of the underlying transition system. Conceptually this leads to exposing students to another basic and pervasive notion in Computer Science, that of *naming*. Hybrid logics [1] are the appropriate tool for this last stage in the course. The need for formulating specific *local* requirements, on the other hand, leads to imposing extra structure upon states. Actually, different states are interpreted as different *modes* of operation and each of them is equipped with an algebraic specification of the corresponding functionality. Technically, specifications become *structured* state-machines, where states are specified as *algebras*, rather than as *sets*.

Tool support. The institution-based framework adopted in the course allows for a smooth integration of these three stages supported by the HETS platform. HETS acts a "motherboard" of logics where different "expansion cards" can be plugged in. These pieces are individual logics (with their particular analysers and proof tools) as well as logic translations. To make them *compatible*, logics are formalised as institutions and translations as comorphisms.

Actually, institutions provide a systematic way to relate logics and transport results from one to another [7], which means that a proving strategy/tool for one logic can be used to reason about specifications written in another one. A fundamental study of how an hybrid language can be systematically endowed in an arbitrary institution was the object of the authors recent research [6, 9]. This so-called *hybridisation method* was implemented in the HETS platform [8], becoming part of its official release from August 2013. This provides for free the proof support environment needed for this course. It should be stressed, however, that, in despite of the crucial role played by institution theory in this approach, no familiarity with institutions is required from students.

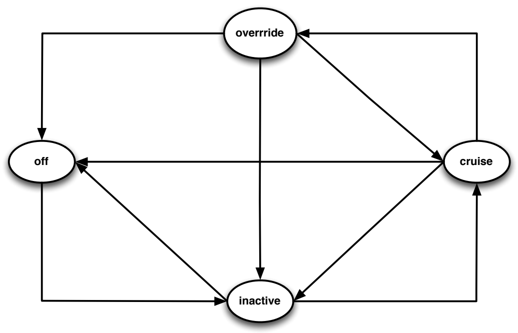


Figure 2. The transition structure.

4 A glimpse of a course session

The course contents and methodology is better introduced through the presentation of a typical problem addressed first in the exercises class and latter in the laboratory, in the last stage of the course. For space limitations we focus only on a fragment of the original problem. The example, small but self-contained, is taken from a description of requirements for an *automatic cruise control* (ACC) system summarised in [3] as follows:

> *"The mode class CruiseControl contains four modes, Off, Inactive, Cruise, and Override. At any given time, the system must be in one of these modes. Turning the ignition on causes the system to leave Off mode and enter Inactive mode, while turning the cruise control level to const when the brake is off and the engine running causes the system to enter Cruise mode. (...) Once cruise control has been invoked, the system uses the automobile's actual speed to determine whether to set the throttle to accelerate or decelerate the automobile, or to maintain the current speed (...)To override cruise control (i.e., enter Override), the driver turns the lever to off or applies the brake".*

These requirements are captured by the state machine depicted in Figure 2 and expressed in *hybrid propositinal logic* ($\mathcal{HPL}$); local properties are given as propositions. The set of $\mathcal{HPL}$ formulas is defined by

$$\varphi, \psi ::= p \,|\, i \,|\, \neg\varphi | [\lambda]\varphi \,|\, @_i\varphi \,|\, \varphi \wedge \psi \,|\, \varphi \vee \psi \,|\, \varphi \Rightarrow \psi$$

where λ ranges over a set Λ of modal operators. Models of this logic are state-machines with an additional function $\mathrm{state} : \mathrm{Nom} \to S$ which assigns to each nominal a state. This allows explicit reference to particular states in a specification. Thus, models are tuples $\mathcal{P} = \langle S, \mathrm{state}, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}\rangle$ where S is a set of states, $R_\lambda \subseteq S \times S$ is the accessibility relation associated to the modality λ and $P_s : \mathrm{Prop} \to \{\top, \bot\}$ is the function that assigns the propositions on the state $s \in S$. The satisfaction relation is defined as in standard modal logic (e.g. $\mathcal{P} \models^s p$ iff $P_s(p) = \top$; $\mathcal{P} \models^s [\lambda]\varphi$ iff $\mathcal{P} \models^{s'} \varphi$ for any s' such that $(s, s') \in R_\lambda$) adding the following cases related to nominals:

- $\mathcal{P} \models^s @_i\varphi$ iff $\mathcal{P} \models^{\mathrm{state}(i)} \varphi$;
- $\mathcal{P} \models^s i$ iff $\mathrm{state}(i) = s$.

Moreover, we abbreviate formulas $\neg[\lambda]\neg\varphi$ and $\langle\lambda\rangle\varphi \wedge [\lambda]\varphi$ to $\langle\lambda\rangle$ and $\langle\lambda\rangle^\circ\varphi$, respectively.

Back to the example, a modality $next$ is introduced to denote the state-machine accessibility relation. Nominals in set $\{\mathit{off}, \mathit{inactive}, \mathit{override}, \mathit{cruise}\}$ correspond to the operation modes mentioned in the requirements. The first element students can formally capture within the logic is the transition structure, as in, for example,

- (T_1) $@_{\mathit{off}}\ \langle next\rangle\ inactive$
- (T_2) $@_{override}\ (\langle next\rangle\ \mathit{off} \wedge \langle next\rangle\ inactive \wedge \langle next\rangle\ cruise)$

Local properties can also be expressed resorting to the satisfaction operator $@_i$, for each nominal i, to refer to the corresponding state. For instance, the requirement that the ignition is off when the system is in the *off* mode, while it is on and the engine running ($\mathit{EngRunning}$) in the $cruise$ mode, is modelled by

- (L_1) $@_{off}(\neg \mathit{IgnOn})$
- (L_2) $@_{cruise}(\mathit{IgnOn} \wedge \mathit{EngRunning})$

Symbols $\mathit{EngRunning}$ and IgnOn, with a self-explanatory designation, are propositions whose validity is discussed in each configuration (state). Others are used in the sequel. Definitional properties can also be captured, as in

- (A_1) $\mathit{LeverOff} \Leftrightarrow \neg\, \mathit{LeverCons}$
- (A_4) $\mathit{HighSpeed} \Rightarrow \neg\, \mathit{CruiseSpeed} \wedge \neg\, \mathit{LowSpeed}$

The second step in the case study is to equip each state of the underlying transition system with a first-order structure, to model its local functionality. Therefore, hybrid structures are enriched with a family of first-order structures indexed by the set of states, i.e., they become structures

$$\mathcal{M} = \langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (M_s)_{s\in S}\rangle$$

where first-order structures in the family $(M_s)_{s\in S}$ are defined over the same signature and universe, say M. Each M_s models the system's behaviour at state $s \in S$. Note that at state s each first order formula is evaluated in the structure M_s. Properties are now expressed in a hybrid first order language $\mathcal{H}$ whose detailed presentation we omit here (but see [5]). We focus instead on the sort of properties students are supposed to formulate. An algebraic specification is used to model system's functionality. This entails the need for introducing data types able to support the envisaged notions of *time*, *speed* and *acceleration*.

spec TIMESORT =INT
with sort *Int* $\mapsto$ *time*, **ops** 0 $\mapsto$ *init*, *suc* $\mapsto$ *after* **end**
spec SPEEDSORT =INT **with sort** *Int* $\mapsto$ *speed* **end**
spec ACELLSORT =INT **with sort** *Int* $\mapsto$ *accel* **end**

Operation $Pedal$ models the accelerations applied by the driver at each moment. On the other hand, $Automatic$ captures accelerations applied on the engine by the ACC, and $CurrentSpeed$ records the current speed. Finally, constant $MaxCruiseSpeed$ represents the maximum speed allowed on the ACC mode:

spec ACCSIGN =
TIMESORT **and** SPEEDSORT **and** ACELLSORT
then ops *Pedal* : *time* $\rightarrow$ *accel*;
Automatic : *time* $\rightarrow$ *accel*;
Speed : *speed* $\times$ *accel* $\rightarrow$ *speed*;
CurrentSpeed : *time* $\rightarrow$ *speed*;
MaxCruiseSpeed : *speed*

Students are asked to identify properties that globally hold, in all possible configurations, and the ones which model local requirements. In the first group we have, for example,

$\forall$ *s* : *speed*; *a* : *accel*; *t* : *time*
• (G_1) $Speed(s, a) \geq 0$
• (G_2) $CurrentSpeed(t) = 0 \wedge Pedal(t) \geq 0 \Rightarrow CurrentSpeed(after(t)) \geq 0$
• (G_3) $Pedal(t) > 0 \Leftrightarrow CurrentSpeed(t) < CurrentSpeed(after(t))$
• (G_4) $Speed(s, a) = s \Leftrightarrow a = 0$
• (G_5) $CurrentSpeed(after(t)) = Speed(CurrentSpeed(t), Pedal(t))$

Local properties refer to specific configurations. For example, in state *off*, $Speed$ and $Pedal$ are null and no other operation in the interface react. Thus,

$\forall$ *t* : *time*; *s* : *speed*; *a* : *accel*
• (L^1_{off}) $@_{off} CurrentSpeed(t) = 0$
• (L^2_{off}) $@_{off} Speed(s, a) = 0$

On the other hand, in state $inactive$, the speed and acceleration depend on the accelerations automatically introduced in the system, i.e,

$\forall$ *s* : *speed*; *a* : *accel*
• $(L^1_{inactive})$ $@_{inactive} Speed(s, a) = s + a$

$\forall$ *t*: *time*; *s* : *speed*; *a* : *accel*
• $(L^{1'}_{cruise})$ $@_{cruise}[CurrentSpeed(t) > MaxCruiseSpeed \Rightarrow Automatic(after(t)) < 0]$
• $(L^{2'}_{cruise})$ $@_{cruise}[CurrentSpeed(t) \leq MaxCruiseSpeed \Leftrightarrow Automatic(after(t)) = 0]$
• (L^3_{cruise}) $@_{cruise} Speed(s, a) = s + a$
• (L^4_{cruise}) $@_{cruise} Pedal(t) \geq 0 \Rightarrow Pedal(t) = Automatic(t)$

An interesting feature in this example is that properties local to states $override$ and *off* do coincide. The system's behaviour on both states only differs in what concerns the definition of the allowed transitions. Actually, students may now be invited to revisit the specification of the transition system presented above. It turns out that some propositions may be re-stated by means of properties of local states. For instance,

$\forall$ *t*: *time*;
• (L_1) $@_{cruise}[CurrentSpeed(t) = 0 \Rightarrow \langle next \rangle^{\circ}(inactive \wedge CurrentSpeed(after(t)) = 0)]$

Finally, in the laboratory session students are invited to translate hybrid to first order specifications and use HETS to animate them, as represented in Fig. 3. On translating to $\mathcal{FOL}$ we end up with the following signature:

ops
$Speed^*$: st^* $\times$ *speed* $\times$ *accel* $\rightarrow$ *speed*;
$Pedal^*$: st^* $\times$ *time* $\rightarrow$ *accel*;...
pred
next : st^* $\times$ st^*; $IgnOn^*$: st^*; ...

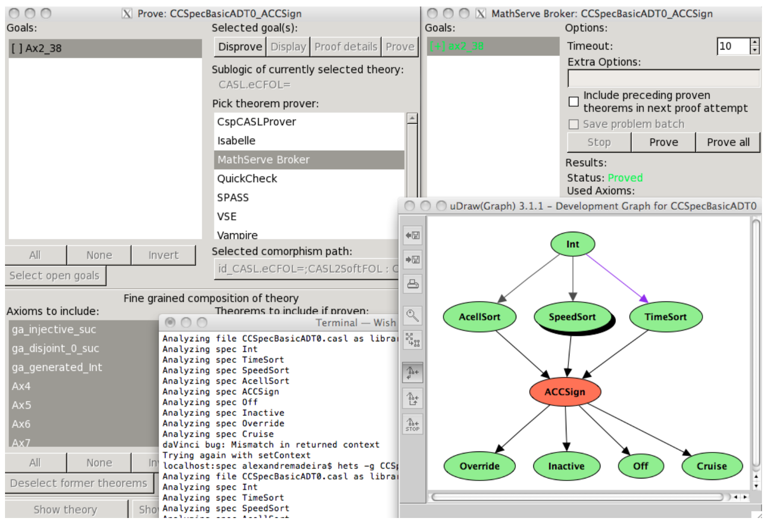
where global properties are universally quantified, and local properties take as an argument the respective nominal. For instance, global properties (G_1) and (G_2) are translated into

$\forall$ *s* : *speed*; *w* : st^*; *a* : *accel*;*t* : *time*
• (G_{1^*}) $\geq^*(w, Speed^*(w, s, a), 0^*(w))$
• (G_{2^*}) $CurrentSpeed^*(w,t) = 0^*(w) \wedge \geq^*(w, Pedal^*(w,t), 0^*(w))$.

and local properties (L^1_{off}) and (L^4_{cruise}), into

$\forall$ *t* : *time*
• $(L^{1^*}_{off})$ $CurrentSpeed^*(off, t) = 0^*(off)$
• $(L^{4^*}_{cruise})$ $\geq^*(cruise, Pedal^*(cruise,t), 0^*(cruise)) \Rightarrow Pedal(cruise,t) = Automatic^*(cruise,t)$.

Two exercises. The ACC case study, a fragment of which was discussed above, is one of the worked examples addressed in an exercises class followed by a laboratory in a total of 3 hours of contact time with the lecturer. The session was prepared by a simpler exercise proposed the week before. Students were expected to work this exercise at home and use the laboratory slot of the previous week to test their specifications. Similarly, after the session devoted to the ACC case study a follow up exercise is proposed. Again, students are expected to work out the specification by their own and use the lectures in the following week to discuss their proposals and analyse them in HETS. We close this section by a brief presentation of these two exercises.



Goals:
[ ] Ax2_38

Selected goal(s):

Sublogic of currently selected theory:
CASL.eCFOL=

Pick theorem prover:
CspCASLProver
Isabelle
MathServe Broker
QuickCheck
SPASS
VSE
Vampire

Selected comorphism path:
id_CASL.eCFOL=;CASL2SoftFOL : C

Axioms to include:
ga_injective_suc
ga_disjoint_0_suc
ga_generated_Int
Ax4
Ax5
Ax6
Ax7

Theorems to include if proven:

```
Analyzing file CCSpecBasicADT0.casl as librar
Analyzing spec Int
Analyzing spec TimeSort
Analyzing spec SpeedSort
Analyzing spec AcellSort
Analyzing spec ACCSign
Analyzing spec Off
Analyzing spec Inactive
Analyzing spec Override
Analyzing spec Cruise
daVinci bug: Mismatch in returned context
Trying again with setContext
localhost:spec alexandremadeira$ hets -g CCSp
Analyzing file CCSpecBasicADT0.casl as librar
Analyzing spec Int
Analyzing spec TimeSort
Analyzing spec SpeedSort
```

Goals:
[+] ax2_38

Options:
Timeout: 10
Extra Options:
☐ Include preceding proven theorems in next proof attempt
☐ Save problem batch

Results:
Status: Proved
Used Axioms:

Figure 3. A HETS session.

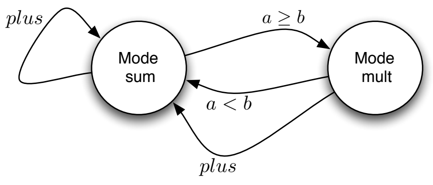


Figure 4. The adaptable calculator.

Warming up exercise: a reconfigurable calculator.

> ***Exercise.*** *An 'adaptable calculator', depicted in Fig. 4 offers a single binary operation over naturals, denoted by $\star$. When the first argument is larger that the second, $\star$ behaves as addition; otherwise it behaves as a multiplication. Additionally, an event plus should reconfigure $\star$ into the addition state.*

Specifying this calculator requires as a first step the identification of two operation modes, say $mult$ and sum. Then two modalities, $shift$ and $plus$, are added to cater for state transitions. The latter corresponds to the $plus$ button, while the former aggregates the two conditions on the arguments. The problem statement also suggests that transitions can be triggered by a property of the arguments involved. Therefore, one may add two propositions, $ALargeOrEqB$ and $ASmallerB$, characterising when the first argument is smaller or larger than the second one. Therefore,

$$ALargeOrEqB \Rightarrow sum \;\text{ and }\; ASmallerB \Rightarrow mult$$

Again, the calculator dynamics can be specified as follows:

$$@_{mult}\langle shift\rangle sum,\; @_{sum}\langle shift\rangle mult \;\text{ and }\; [plus]sum$$

Typically, these propositions can be further characterised, for example,

$$ALargeOrEqB \Leftrightarrow \neg ASmallerB$$

and events related through, e.g.

$$[plus][shift]p \Leftrightarrow [plus]p$$

The next step adds structure to both execution modes, based on the following first-order signature:

sorts Nat;
ops $0 :\to nat$; $suc : nat \to nat$; $p : nat \to nat$; $\star : nat \times nat \to nat$;
pred $\geq : nat \times nat$; $< : nat \times nat$;

over which the common knowledge about the data handled by the calculator and invariant properties of the $\star$ operation (e.g. its commutativity and associativity) are stated, e.g.

$$\begin{aligned}&\forall\, n, k : Nat\\ &\quad p(suc(n)) = n;\ \wedge \geq (suc(n)), n) \wedge < (n, suc(n))\\ &\quad \star(n, k) = \star(k, n)\ \wedge \star(n, \star(k, l)) = \star(\star(n, k), l)\end{aligned}$$

Then local properties of the $\star$ operation are specified in $\mathcal{H}$, for example,

$$\begin{aligned}&\forall\, n : Nat\\ &\quad @_{sum} \star (n, 0) = n \wedge @_{sum} suc(n) = \star(n, suc(0))\\ &\quad @_{mult} \star (n, 0) = 0 \wedge @_{mult} \star (n, suc(0)) = n\end{aligned}$$

Finally, students are supposed to revisit the specification *dynamics* to express the original transition propositions in terms of state values, leading to e.g.

$$(\forall_{n:Nat} : \star(n, 0) = n) \Rightarrow [shift](\forall_{n:Nat} : \star(n, 0) = 0)$$

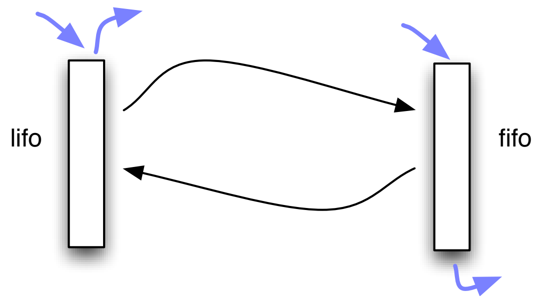


Figure 5. The plastic buffer.

$\forall\, n,k : Nat$
$\quad \big(\geq (n,k) \Rightarrow \star(n,k) = z \big) \Rightarrow @_{mult} \star (n,k) = z$
$\quad \big(< (n,k) \Rightarrow \star(n,k) = z \big) \Rightarrow @_{sum} \star (n,k) = z$
$\quad [plus] \star (n,k) = z \Rightarrow @_{sum} \star (n,k) = z$

Follow up exercise: a plastic buffer.

> ***Exercise.*** *An 'plastic buffer' (Fig. 5) is a versatile data structure with two distinct modes of execution: in one of them it behaves as a stack; in the other as a queue. Reconfiguration is triggered by the number of items stored, which measures the difference between the rates associated to the incoming and outcoming data streams: when this value is larger than $limit$, the buffer turns into a stack; otherwise into a queue.*

This has a similar structure to the *warming up* example, acting as a consolidation exercise of the whole unit.

5 Concluding

The paper introduced the *rationale* for a somehow not very standard introductory course on Formal Methods. From an institution-based framework, kept implicit along the lectures, the course aims at conducting students through two orthogonal paradigms (equational and hybrid) which are then combined in common specification framework.

In [6] the authors introduced a method to hybridize arbitrary institutions. The approach underlying the course proposed here is actually based on a particular instance of such a general method. However, other possible 'hybridizations' (eg. of institutions of multialgebras or partial algebras) are suitable to explore a wide range of exercises in a similar spirit. Moreover, the course skills may be easily expanded into new directions: for instance, functional and imperative programming languages may be presented as institutions (see [11]) whose hybridization may be used to develop reconfigurable algorithms. In [10], the authors have also presented the logic underlying ALLOY [4] in an institutional setting. This paves the way to hybridising ALLOY and combining in the course the use of the traditional ALLOY model finder with theorem proving (in HETS) in an integrated way.

Acknowledgements. On-going collaboration with Răzvan Diaconescu is greatly acknowledged. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme and by National Funds through FCT within project `FCOMP-01-0124-FEDER-028923`. M. Martins was also supported by the project PEst-OE/MAT/UI4106/2014.

References

[1] Patrick Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of IGPL*, 8(3):339–365, 2000.

[2] Joseph A. Goguen and Rod M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39:95–146, 1992.

[3] C. L. Heitmeyer, J. Kirby, and B. G. Labaw. The SCR Method for Formally Specifying, Verifying, and Validating Requirements: Tool Support. In *ICSE*, pages 610–611, 1997.

[4] Daniel Jackson. *Software Abstractions (Logic, Language, and Analysis)*. MIT Press, 2nd edition, 2011.

[5] Alexandre Madeira, José M. Faria, Manuel A. Martins, and Luís S. Barbosa. Hybrid specification of reactive systems: An institutional approach. In G. Barthe et al, editors, *Software Engineering and Formal Methods*, volume 7041 of *LNCS*, pages 269–285. Springer, 2011.

[6] Manuel A. Martins, Alexandre Madeira, Răzvan Diaconescu, and Luís S.Barbosa. Hybridization of institutions. In A. Corradini et al editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *LNCS*, pages 283–297. Springer, 2011.

[7] Till Mossakowski. Foundations of heterogeneous specification. In M. Wirsing et al, editors, *Recent Trends in Algebraic Development Techniques 2002*, volume 2755 of *LNCS*, pages 359–375. Springer, 2003.

[8] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, Hets. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems* , volume 4424 of *LNCS*, pages 519–522. Springer, 2007.

[9] Renato Neves, Alexandre Madeira, Manuel A. Martins, and Luís S. Barbosa. Hybridisation at work. In R. Heckel and S. Milius, editors, *Algebra and Coalgebra in Computer Science*, volume 8089 of *LNCS*, pages 340–345, 2013.

[10] Renato Neves, Alexandre Madeira, Manuel A. Martins, and Luís S. Barbosa. An institution for alloy and its translation to second-order logic. In Thouraya Bouabana-Tebibel and Stuart H. Rubin, editors, *IRI (best papers)*, Advances in Intelligent Systems and Computing, pages 45–75. Springer, 2013.

[11] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs on Theoretical Computer Science, an EATCS Series. Springer, 2012.