CAMILA: Formal Software Engineering Supported by Functional Programming

J. J. Almeida, L. S. Barbosa, F. L. Neves and J. N. Oliveira

Computer Science Department
University of Minho
Largo do Paço — 4710 Braga
Portugal

{jj,lsb,fln,jno}@di.uminho.pt

Abstract. This paper describes two experiences in teaching a formal approach to software engineering, at undergraduate level, supported by CAMILA, a functional programming based tool. Carried on in different institutions, each of them addresses a particular topic in the area: *requirement analysis* and generic *systems design* in the first case, *specification* and *implementation development* in the second.

CAMILA, the framework common to both experiences, animates a set-based language, extended with a mild use of category theory, which can be reasoned upon for program calculation and classification purposes. The project affiliates itself to, but is not restricted to, the research in exploring Functional Programming as a *rapid prototyping* environment for formal software models. Its kernel is fully connectable to external applications and equipped with a component repository and distribution facilities.

The paper explains how CAMILA is being used in the educational practice, as a *tool to think with*, providing a kind of cross-fertilization between students' understanding of different parts of their curriculum. Furthermore, it helps in developing a number of *engineering skills*, namely the ability to analyze and classify (information) problems and models and to resort to (the combined use of) different programming frameworks in approaching them.

Keywords Education and applications of functional programming, functional prototyping, program calculation.

1 Introduction

A significant majority of the students who apply for an undergraduate degree in Software Engineering do possess, from the outset, some *working knowledge* in computation. Yet some are really sparkling with particular systems or tools. University teaching is expected to build on this knowledge for developing *engineering skills*. That is to say, the ability to classify (information) problems and models and to resort to (the combined use of) different programming frameworks in approaching them. Moreover, the basic problem solving strategy people get used to from school physics — *understand the problem, create*

a mathematical model, reason in the model, calculate a solution — should be taught as the usual way of dealing with Software Engineering problems.

It is widely recognized that declarative languages allow for the expression of concepts and structures at a high level of abstraction. Moreover, they stimulates a kind of compositional reasoning which paves the way to sound methodological principles. This may explain the increased use of those languages in teaching a number of areas ranging from compiler construction to databases, interface design to fuzzy reasoning or discrete mathematics (see [HPe95] for a recent overview). Furthermore, functional languages are being adopted in several institutions as first programming languages [TH95].

This paper describes two experiences in using a particular functional programming system — CAMILA [1] — to teach the principles of a Software Engineering methodology at undergraduate level. They were carried out in different institutions and each of them addresses a particular topic in the area: *requirement analysis* and generic *systems design* in the first case, *specification* and *implementation development* in the second.

The common framework to both experiences is CAMILA, an emerging platform for (mathematical) software development. Based on a notion of *formal software component*, CAMILA encompasses a set-theoretic language and an inequational *calculus* [Oli90,Oli92] for classifying and reasoning about such components. In particular, it enables the synthesis of target code programs by transformation of the initial specifications. Its kernel is a functional prototyping environment [BA95], fully connectable to external applications, equipped with a component repository and distribution facilities.

CAMILA was originally devised as a collection of interrelated support tools for teaching formal specification methods. A number of educational projects and non trivial case-studies carried out in industrial contexts (see, *e.g.* [SS94] or [Oli95]) have shown the language potential in exploring a broader scope of the curriculum. In fact, using CAMILA as a common working framework provided a kind of cross-fertilization between students' understanding of a number of different domains. Furthermore it helps in concentrating on the essential and requiring more and more precision and clarity in problem formulation.

CAMILA development affiliates itself to, but is not restricted to, the research in exploring Functional Programming as a *rapid prototyping* environment for formal software models, whose origin can be traced back to Peter Henderson's me too [Hen84]. In fact, a major achievement of Functional Programming has been to enforce a view of programming as a *mathematical* activity, at the right (human) level of abstraction, concentrated on exploring the complexity (and beauty) of problems to be solved. Since the pioneer-

[1] CAMILA (see `http://camila.di.uminho.pt`) is named after a Portuguese 19th-century novelist — Camilo Castelo-Branco (1825 - 1890) — whose immense and heterogeneous writings, deeply rooted in his own time experiences and controversies, mirrors a passionate yet difficult life.

ing working of McCarthy [McC63], the interplay of research in Functional Programming, Formal Specification Methods and Semantics, has made it possible to liberate software development from *ad hoc* approaches and to set up information technologies with sound mathematical basis. We believe that this programme remains, at present, a fundamental concern, with relevant implications in education.

In the next two sections we try to support this claim by describing in some detail the above mentioned teaching experiences using CAMILA in a Software Engineering course at undergraduate level. The CAMILA language and *calculus* will be introduced along the way. Finally, section 4 presents some conclusions and comparison with related work.

2 Case A: Systems Modelling and Design

The first experience to be reported here concerns the use of CAMILA for approaching requirement analysis and software design, as part of the Systems Design course taught to third-year students of the Computer Systems Engineering degree at the University of Bristol (UK). This degree is particularly oriented towards computer architectures, communications and systems programming, though in the first two years students are supposed to undertake courses in programming (using HASKELL and C) as well as in logic, discrete mathematics and λ-*calculus*.

The Systems Design course aims to challenge students' ability to deal with the difficulties of analyzing and managing the design of "real" information systems as well as to develop teamwork skills. In this context a medium-size case study, in the form of a (often ambiguous) user requirements document, is assigned to a group of up to ten people. A full prototype of the system, exhibiting its functionality and modular organization, is expected within a 3 months period. Typical themes have been, in recent years, an *Emergency Network Management System*, a *Control System for a TaxiBus Service*, *Data Mining over a Temporal Warehouse Database* and a study of *Fault-Tolerance Strategies for Distributed Systems*.

In a first phase, CAMILA is used to model the project requirements, capturing its structure as a network of *software components*.

When analyzing the structure of an information system, Software Engineering draws a fundamental distinction between *entities*, which represent information sources, and *transformations* upon them. The former will originate the data structures, the later the algorithms. A similar distinction appears in the definition of an algebra (sets and functions) or relational structure (sets and relations), which makes such mathematical objects suitable in modelling information systems. Hence, mathematically, a *software component* stands for a (multi sorted) *algebra* or a *relational structure*, which expresses, in a concise but meaningful way, the specification of (some part of) an information system.

Components are described in the centenary notation of set theory. For several years in their past education students have become familiar with such a notation as a *tool to think with*. This course intends to build on such a background.

In fact, the specification language is an executable fragment of set theory, exploring the "pure" mathematical notation arising from the (Cartesian closed) structure of the category Set of sets and set-theoretical functions. It should be stressed, however, that the categorical properties and constructions are used in an implicit way: whoever uses CAMILA, namely the students involved in the educational projects reported here, are not supposed to have any kind of familiarity with category theory.

Basic set constructors capture essential operations upon information:

- *Cartesian product* $(A \times B)$ for aggregation in the spatial axis,
- *coproduct* $(A+B)$, for choice (i.e., aggregation in the temporal axis) and
- *exponentiation*, or function space, (A^B) for functional dependence.

Notice that when processing entities definitions, the prototyper generates automatically the constructor and projections associated to each product type as well as the canonical embeddings and origin predicates associated to coproducts types[2]. The universal tupling and co-tupling operations are also available. A number of derived constructors are also defined in the CAMILA kernel notation. These include, for finite A and B,

$$
\begin{array}{rcll}
\mathbf{2}^A & & & \textit{finite subsets} \\
\mathbf{2}^{A\times B} & & & \textit{binary relations} \\
A \hookrightarrow C & \triangleq & \sum_{K\subseteq A} C^K & \textit{finite mappings} \\
A^* & \triangleq & \sum_{n\in\mathbb{N}} A^n & \textit{finite sequences}
\end{array}
$$

as well as the "null" alternative $(A+\mathbf{1})$, where $\mathbf{1}$ is a terminal (singleton) set, and recursive definitions in the form $X = \mathcal{F}(X)$, where $\mathcal{F}$ is a set-theoretical expression involving the above constructs.

These constructs are directly expressible in the prototyping language[3], from which a high level description is automatically generated (in the form of a LaTeX file). The basic algebras associated with them (*e.g.*, intersection or union of two sets, joining of binary relations, domain or range of a mapping, concatenation of sequences and reduce operators, structure definition

[2] In a coproduct $A \xrightarrow{i_A} A+B \xleftarrow{i_B} B$ origin predicates are defined as in is-$A(x) \triangleq \exists_{a\in A}\, i_A(a) = x$.

[3] The concrete syntax for the above mentioned derived constructors being `A-set`, `A <-> B`, `A -> B` and `A-seq`, respectively.

by enumeration or comprehension, etc.) are also available as primitives in the language. So are the propositional connectives and quantifiers. Anonymous function definitions, in the form of λ-expressions, and high-order functions are also allowed.

An important feature (to our knowledge new in an executable notation) is the fact that the set constructors also act upon functions (either primitive or user-defined) lifting its effect to the generated structure. Technically this is rooted on the language's catagorial foundations and amounts to saying that entities in CAMILA are modeled by (endo)functors in Set, uniformly transforming either sets and functions [OB97]. For example, the expressions `(f-set)-seq` and `(f-seq)-set` correspond, respectively, to the action upon the function $f : A \longrightarrow B$ of the functors $(\mathbf{2}^{-})^{*}$ and $\mathbf{2}^{(-^{*})}$. In the first case f is applied to all the elements of the sets aggregated in a sequence; in the second to all the elements of the sequences collected in a set [4].

These enables a particularly fruitful *modular calculus* in which students are able either to *enrich* or *specialize* their system components by composing them with suitable functors, respectively, on the right and on the left.

For example, starting from a simple model of mappings from attributes to something — $Att \hookrightarrow _$ — one may define a (parameterized) data relation as a set of such mappings: $\mathbf{2}^{Att \hookrightarrow _}$. This may be either *enriched*, *e.g.* by aggregating a decision tree component on the attributes $DT = P \times (Att \hookrightarrow DT)$, P standing for a set of propositions about attributes, or *specialized*, *e.g.* to allow for associations of attributes to sequences of values. Doing both amounts to the composition:

$$(_ \times DT) \circ \mathbf{2}^{Att \hookrightarrow _} \circ _^{*}$$

yielding the new model

$$S(_) = \mathbf{2}^{Att \hookrightarrow _^{*}} \times DT$$

This may be further instantiated to $S(Val)$, Val standing for a particular representation of values in the system, or further lifted to a temporal indexed structure by composing again on the right with the functor $T \hookrightarrow _$, where T is a non-empty set thought of as a discrete representation of time. This yields $T \hookrightarrow S(Val)$, one of the structures actually used in the temporal warehouse database project. The following diagram shows the functor compositions involved:

---

[4] `f-seq` can be easily recognized as the classical LISP `mapcar` functional.

$$
\begin{array}{l}
Att \hookrightarrow _ \\
\Big\downarrow {\scriptstyle \mathbf{2}^{-} \circ _} \\
\mathbf{2}^{Att \hookrightarrow _} \xrightarrow{\;_ \circ _^{*}\;} \mathbf{2}^{Att \hookrightarrow _^{*}} \\
\qquad\qquad\qquad\quad \Big\downarrow {\scriptstyle (_ \times DT) \circ _} \\
\qquad\qquad\quad (\mathbf{2}^{Att \hookrightarrow _^{*}}) \times DT \\
\qquad\qquad\qquad\quad \Big\downarrow {\scriptstyle (T \hookrightarrow _) \circ _} \\
\qquad\qquad T \hookrightarrow ((\mathbf{2}^{Att \hookrightarrow _^{*}}) \times DT)
\end{array}
$$

What is interesting in this process is the fact that all the functions defined upon the simple $Att \hookrightarrow _$ model may be lifted to the new structures in a functorial way, and this mechanism is indeed supported by the language. For example, having defined a query operation $q : S(Val) \longrightarrow Val$ upon $S(Val)$, one may directly use $T \hookrightarrow q : T \hookrightarrow S(Val) \longrightarrow T \hookrightarrow Val$ on the temporal enrichment.

The CAMILA notation provides a straightforward way to make progress from very simple models to complex ones. Moreover it offers a framework for *classifying* models (and problems) encouraging *re-use* of previous solutions. In fact, architectural relationships in the CAMILA repository such as *is-a*, *is-used-by*, *is-special-case-of* or, as we shall discuss in the following section, *is-implementation-of*, are formally decided rather than fixed by intuition. Furthermore they are built-in in the CAMILA components repository. As an example of the former consider again the entity DT of a component modelling *decision trees*. This may be specialized into a model of either *genealogical diagrams* or *subject taxonomies* as soon as we make $Att = \mathbf{2}$ or $P = \mathbf{1}$, respectively[5] (see [Oli92] for details).

As most projects in Software Engineering have to deal with distribution and communication issues, and these form a fundamental part of the curriculum, the orientation towards concurrent and distributed systems emerged as a major theme in the CAMILA project from the very beginning. The prototyping environment offers a small set of communication primitives so that, in a second phase of the Design Project, students are able to distribute their components along a network of CAMILA processes, simulating the physical architecture of the "real" system. A prototyper tool (called `interface`) generates for each component the algebraic signatures of the available external services offered[6]. Different communication disciplines — ranging from syn-

[5] Making simultaneously $Att = P = \mathbf{1}$, DT boils down to a model of the natural numbers (by $DT \cong \mathbf{1} \times (\mathbf{1} \hookrightarrow DT) \cong \mathbf{1} \hookrightarrow DT \cong DT + \mathbf{1}$).

[6] In subsequent phases this same tool allows for the automatic generation of on-line helpers in the format of the UNIX `man`.

chrony to asynchrony, or from point-to-point to multicast — can be prototyped in CAMILA just in the *same* functional style used in modelling any other software component. Moreover, structural properties of these disciplines can be documented (and compared) as equations on functional terms. Notice that this usually provides a fresh look on the students previous background on protocols and communications.

In summary we shall point out that resorting to CAMILA as a framework to study Systems Design has been proved helpful in a number of issues:

- CAMILA helps in concentrating on the essence of the different aspects to be modeled, recording their structure and properties at an abstract level, while retaining the possibility of *executing (*i.e., *animating) the design.* More than being seen as another course in the curriculum, CAMILA is understood as a common language to integrate previous knowledge, interrelating concepts and put them to work.
- It requires more precision from students, contributing to developing strategies for the correct formulation of systems requirements and questioning their own designs by asking the prototyper questions like *and what if ...?*. Notice the target students have no previous experience on formal specification methods, which are introduced, through a Z course, only on the following semester.
- Finally, we have found that using CAMILA as a project framework allows for a more effective communication among the design team, developing teamwork skills, and emphasizes the incremental and iterative character of the software design process. A distinguished feature of CAMILA is the capacity to handle *partially defined functions*, i.e. functions whose signature has been declared but whose computation rule has not yet been supplied. Whenever the interpreter is requested to evaluate such a function it will prompt the user for a value and proceed with the calculation. This enables to test incomplete prototypes, eventually integrating material which is still under development.

3 Case B: Refinement and Interconnection

This section describes the experience gathered at Minho University (Portugal) in teaching formal methods for software development at undergraduate level, supported by functional programming tools. Although these methods have been taught to Minho undergrads since 1984, it is only after the advent of the CAMILA environment (1990) and of the first steps of the SETS reification calculus [Oli87] that such a teaching effort becomes systematic and follows a regular pattern — that of the *formal development life cycle* depicted in Figure 1.

As systems modelling in CAMILA has been discussed in the previous section, we will be concerned here with the development phase, which is the main subject of a final year undergraduate course in Formal Methods at

Minho. The overall target is to teach students how to *develop a client-server architecture* for a (possibly distributed) information system.

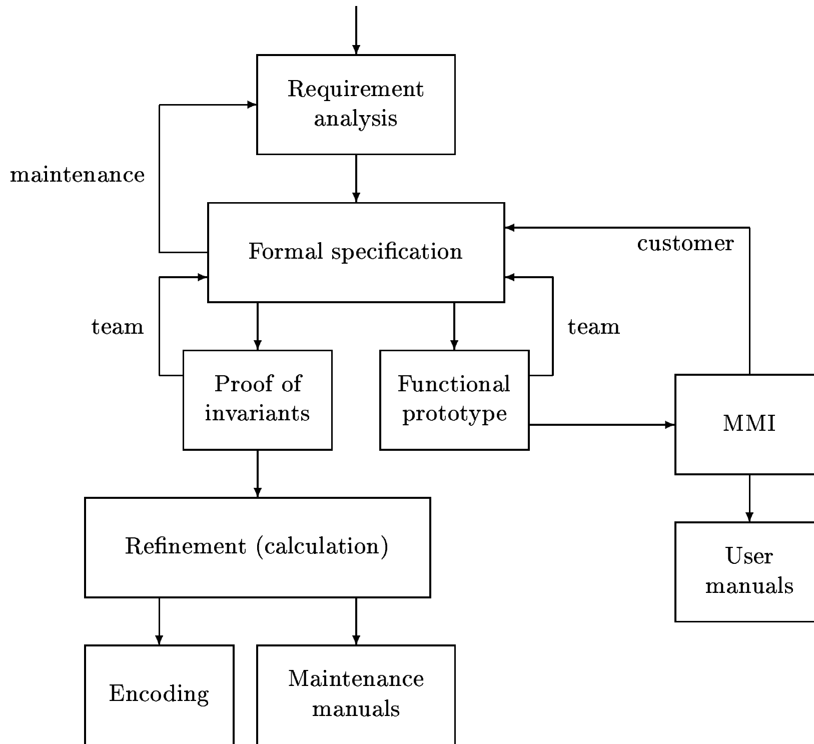

Fig. 1. A Formal Software Development Life Cycle

Design starts from developing a formal model (specification) from a set of requirements. This is done in a stepwise-elaboration style, each stage of the model being immediately prototyped in the CAMILA functional animator and quick feedback about its behavior being gathered within the design team (see the *team* arrow). CAMILA's type-checking filters primary specification (syntactic) errors and unexpected semantic behavior is likely to be spot and corrected.

After a few iterations, the design should become stable and all members of the team should believe in it. It will be time to bring the "costumer" in (either the teacher or whoever wrote the requirements), just to check for casual misreadings of the requirements. To comply with current standards on human-machine interaction, it will be convenient to "hide" the prototype behind a capable window-manager. This is the start-point of the client-server

bi-partition, made possible by the fact that CAMILA code can be easily embedded in C/C++ code[7].

Students are encouraged to define the functional API which emerges naturally from the algebraic structure of the formal model itself, so that their prototype may be called from the outset as a normal C/C++ server application. It is such an algebraic signature which induces the specification of a "canonical" syntax-directed-editor-like interaction layer (see box *MMI* in Figure 1), following a formal approach to user-interface design described elsewhere (see *e.g.* [MO90,Mar95]). The next task is to encode such a structural editor in a particular technology, typically TCL/TK, Borland's DELPHI or JAVA, and provide for its interconnection to the C/C++ protocol which embeds the original CAMILA prototype (see Figure 2).

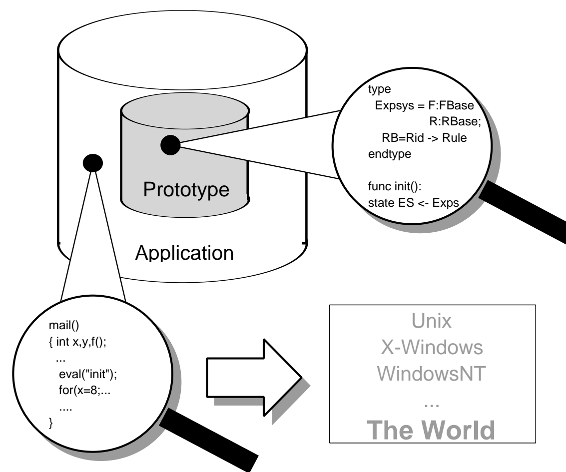

Fig. 2. Design Embedding in CAMILA

At this point, the existing formal specification and its prototype are likely to undergo a new iteration of changes, as suggested by the costumer's taste and experimentation with the prototype system. These may either be concerned with the user-interface (adding to or modifying the underlying signature) or the semantic model itself (adding to or modifying the behavior of

[7] But note that this client-server split is not bound to the need for a user interface. A good illustration is provided in report [RP95]: in a cross-fertilization with a parallel course on compiling, a batch version of a production planning system was made available as an interpreter (written in ELI), the semantic actions of which were emulated by CAMILA animation of the system's formal specification, on the background.

the provided functionality). This is the "higher-energy" design iteration loop which corresponds to the *costumer* arrow in Figure 1.

Once both team and costumer agree to exit from this loop, two parallel activities may start: the preparation of the user-manuals, made possible by the belief that the user-interface has reached a stable phase[8], and the so-called *reification phase* (see box *Refinement (calculation)* in Figure 1). The latter is preceded by what may be called a (fine-grain) "design certification" effort: should *data-type invariants* be present in the formal specification, time has come for providing invariant preservation formal proofs concerning all relevant operations. This is perhaps the most expensive task in the whole life-cycle (as it requires good skills in mathematics) but it is of vital importance to the overall quality of the design.

After discharging all proof obligations one may proceed to the reification phase. This is a systematic process, that of step-wise transformation of the original specification into another formal specification which can be eventually recognized as the formal semantics of a particular command, or program fragment in the intended target server technology. A particularity of the CAMILA life-cycle at this point is to adopt a *program calculation* strategy instead of the more conventional "invent-and-verify" one. For this purpose, a particular *abstraction invariant calculus* (using the terminology of [Mor90]) has been developed — SETS [Oli90,Oli92] — which exploits the categorical foundations of the CAMILA's notation so that more and more concrete data-structures modelling the specification sorts can be found by calculation, accompanied by the synthesis of abstraction functions and induced implementation-level invariants.

Different laws of SETS lead to different implementation structures and platforms. For instance, the most common target technology is that of relational databases, typically materialized by a particular SQL server. A database table is trivially formalized, in the SETS notation, by a relation in $\mathbf{2}^{A \times B}$ or a mapping in $A \hookrightarrow B$, for A,B arbitrary products of "atomic" types. Therefore, all SETS laws which somehow "lead" to such structures are welcome by such a target environment[9]. A SETS law is an (in)equation of the form

$$A \trianglelefteq_f^{\phi} B$$

[8] In practice, it is hard to obtain such a high degree of satisfiability: because the user-interface is "what" the costumer has access to and effectively "sees", changes and more changes (some of them at pure cosmetic level, though) are likely to be suggested until the project's very end.

[9] Should the target programming language be, for instance, C, then laws leading to structures of the $1 + A$ pattern (the "pointer to A abstraction") will become relevant, in particular recursive structures of the $X = 1 + \mathcal{F}(X)$ shape. Besides the relational database paradigm, including a re-interpretation of *normalization theory* as a subset of the SETS theory [Rod93], *hash tables* are a class of implementation device which has been carefully calculated in SETS [Oli94].

stating that every instance of A can be reified into the corresponding instance of B, by adopting abstraction function f and provided that concrete invariant ϕ is enforced over B. On the whole, the following abstraction invariant is synthesized:

$$\lambda ab \,.\, \phi(b) \wedge (a = f(b))$$

For instance, law

$$A \hookrightarrow D \times (B \hookrightarrow C) \trianglelefteq^{dpi}_{\bowtie_n} (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{1}$$

states that finite mapping nesting can be flattened. Repeated application of this law makes it possible to boil arbitrarily nested, intricate mapping-based data structures, down to products of atomic relation tables. The relevant abstraction function ($\bowtie_n$) computes a kind of "nested join" and invariant *dpi* will guarantee that such a join operation is effectively computed (see *e.g.* [Oli92] for details).

SETS is pregnant of useful laws for data reification. Among these, laws which like (1) "push the $\times$ construct outwards" make for *horizontal refinement* [Gog86] and distribution [OBM97] in a natural way. This has a beneficial consequence at CAMILA prototyping level: students may choose which "factor" of a given product (*e.g.* the left-hand-side of (1)) will be *vertically reified* in the next place. This leads to what (in the CAMILA terminology) is called a "hybrid" prototype: parts of the system which are already fully reified may cohabit (and communicate) with other parts still awaiting for their reification to take place (*i.e.* functionality still emulated by a CAMILA process).

Of course, such temporary configurations of the system (which may require abstraction/representation functions explicit at run time) cannot be expected to be particularly efficient ones. But they provide for smooth, stepwise reification and testing. Should the system be too complex or students run out of time, a hybrid prototype system will be tolerated[10]. Otherwise, all CAMILA components will eventually yield place to full implementations[11].

Concerning operation reification, and once the overall abstraction invariant has been calculated, two alternatives are available: either the Oxford Refinement Calculus [Mor90] or the Fold-Unfold Calculus [Dar82]. The latter has been more popular simply because of the functional flavor of CAMILA

---

[10] Report [SRVN96] of last year's course provides a good example of this procedure, in which students went further to displaying data evaluated by the CAMILA subprototype in blue-background text-boxes and data produced by already calculated SQL code (over ORACLE) in white ones, thus providing evidence of the hybridization process itself at demo time.

[11] Report [RP95] of the 1995/96 course was particularly successful in showing how flexible the interplay between TCL/TK, C-embedded CAMILA, and Pro-C-embedded SQL may happen to be. A full CAMILA prototype was demonstrated side by side with the full SQL version (final implementation). Switching from CAMILA to SQL was (at compile-time) obtained simply by swapping a `.h` file.

specification. For each abstract function

$$\sigma : A \longrightarrow B$$

in the formal specification, and refinement diagram

$$\begin{array}{ccc} A & \xrightarrow{\sigma} & B \\ \uparrow f & & \uparrow g \\ A_1 & & B_1 \end{array}$$

where A_1 and B_1 are implementations of, respectively, A and B (witnessed by the abstraction functions f and g), the exercise amounts to finding a solution for σ_1 in equation

$$\forall_{a_1 \in A_1} \cdot g(\sigma_1(a_1)) = \sigma(f(a_1)) \tag{2}$$

thus "closing" the diagram:

$$\begin{array}{ccc} A & \xrightarrow{\sigma} & B \\ \uparrow f & & \uparrow g \\ A_1 & \xrightarrow[\sigma_1]{} & B_1 \end{array}$$

The categorical basis of the SETS calculus helps particularly in this reasoning by providing many "functorial" or "natural transformation" ("Wadler-like" [Wad89]) theorems for function transformation [OB97].

The final tasks of code and documentation generation (see the *Encoding* and *Maintenance manuals* boxes in Figure 1) are softened by the overall discipline. The former is a repetitive exercise of re-writing "abstract code" into some concrete syntax, that of the chosen target language (*e.g.* the same tail-recursive abstraction will lead to different `for`-loops in PASCAL or C). The latter is but a systematic transliteration of mathematics into natural language, browsing the whole specification and reification process.

Last but not least, a few words about the *maintenance* arrow in Figure 1. The academic context in which these experiments have been carried out has not yet provided sufficient experience in this area. However, the overall formal discipline can only help in keeping the impact of changes under control, once

the documentation can be easily followed up (provided the maintainer knows the adopted formal method). The "locality of effect" of both the formal specification and the formal reification process (as inherited from the underlying mathematics) is the good news here, in contrast with chaotic development and eventual lack of information.

4 Conclusions and Comparison

In recent years CAMILA has been used on several educational projects as part of undergraduate and master degrees curricula, witnessing the adequacy of Functional Programming in developing abstraction, re-use and calculation skills in Software Engineering (see [Ker95] for a similar discussion in a related area). The two experiences reported here may provide some evidence on this claim.

With respect to the first, we would like to point out that, quite surprisingly (or perhaps not), manipulating and combining Set-expressions appears to be a much more effective way (with a clear semantics and rich *calculus*) of understanding and reasoning about a sheet of requirements, than the tons of "diagrams" and "structured analysis" used in traditional approaches to teaching Software Engineering. Notice, however, that CAMILA incorporates a tool for generating the mathematics underlying traditional Entity-Relationship diagrams [OC93] useful whenever a particular system component is already implemented and some reverse engineering is needed to reason about it. Design evolves in an experimental setting, starting with simple, yet executable models and calculating possible elaborations. The words of P. Halmos about mathematics, in [Hal85], may be adopted to the kind of approach to Software Engineering CAMILA stimulates:

> *Mathematics is not a deductive science – that's a cliché. When you try to prove a theorem, you don't just list the hypotheses, and then start to reason. What you do is trial and error, experimentation, guesswork. (...) the source of all great mathematics is the special case, the concrete example. It is frequent in mathematics that every instance of a concept of seemingly great generality is in essence the same as a small and concrete special case.*

With respect to the second experience, our main conclusion is that the CAMILA life cycle provides a sober blend of the formal methods reasoning and the functional programming traditions[12], in a way which appears to be reasonably successful in both academia and industry.

But the potential of Functional Programming in teaching may be further assessed by looking at CAMILA experiences in the extreme points of the educational spectrum: teaching set theory and elementary discrete mathematics

[12] Sometimes, one has the impression of programming and reasoning in a categorical version of Backus FP [Bac78].

at secondary school level [VA94], one the one hand, training programming professionals in industry, on the other.

The CAMILA approach to programming technology claims to provide a smooth way to teaching and using (constructive) formal methods in software engineering. Similar motivations may be found in the research on *formal specification methods*, such as VDM [Jon86], Z [Spi89], B [Lan96], RAISE [Geo91], COLD-K [FJ92] or LARCH [GH93]. In fact, a CAMILA component resembles what is called a *model* in the VDM meta-language or a *schema* in Z. We could stress, however, the lighter notation of CAMILA, borrowed from set theory, and the direct correspondence to the prototyping functional system. But what is, to our knowledge, new is the associated calculus for model reasoning and refinement as well as the full incorporation, at the executable level, of the *functorial* structure of functions. On the other hand, CAMILA lacks the sophisticated interface and documentation management features available, for instance, in RAISE, and the debugging and code generation facilities of the IFAD VDM-SL toolbox [FL97].

Acknowledgments

The CAMILA project has been supported by the JNICT council under PMCT contract nr. 169/90. Current research is funded by the LOGCOMP project under PRAXIS XXI contract nr. 2/2.1/TIT/1658/95. The on-going interaction with our students at Minho and Bristol Universities has provided much insight into both the tools and the methodology. Fruitful discussions with Bruce Pilsworth on the Bristol experience have been illuminating.

References

[BA95] L. S. Barbosa and J. J. Almeida. CAMILA: a reference manual. Technical Report DI-CAM-95:11:2, DI (U. Minho), 1995.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–639, August 1978.

[Dar82] J. Darlington. Program transformation. In *Funct. Prog. and Its Applications: An Advanced Course*. Cambridge Univ. Press, 1982.

[FJ92] L. Feijs and H. Jonkers. *Formal Specification and Design*. 35. Cambridge Tracts in Theoretical Computer Science, 1992.

[FL97] J. Fitzgerald and P. G. Larsen. *Modelling Systems, Pratical Tools and Methods*. Cambridge University Press (to appear), 1997.

[Geo91] C. George. The RAISE specification language: a tutorial. In *Proc. of VDM'91*. LNCS (551), 1991.

[GH93] J. Guttag and J Horning. LARCH*: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[Gog86] J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.

[Hal85] P. Halmos. *I Want To Be a Mathematician*. MAA Spectrum, 1985.

[Hen84] P. Henderson. me too: A language for software specification and model building. Preliminary Report, University of Stirling, 1984.

[HPe95] P. H. Hartel and M. J. Plasmeijer (editors). *Functional programming languages in education (Proc. of FPLE, Nijmegen).* LNCS (1022). Springer-Verlag, 1995.

[Jon86] Cliff B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.

[Ker95] E. T. Keravnou. Introducing computer science undergraduates to principles of programming through a functional language. In P. H. Hartel and M. J. Plasmeijer, editors, *Functional programming languages in education (FPLE), LNCS 1022*, pages 15–34, Nijmegen, The Netherlands, Dec 1995. Springer-Verlag.

[Lan96] K. Lano. *The B Language and Method: A Guide to Practical Formal Development.* FACIT. Springer-Verlag, 1996.

[Mar95] F. M. Martins. *Métodos Formais na Concepção e Desenvolvimento de Sistemas Interactivos.* University of Minho, 1995. Ph. D. thesis (in Portuguese).

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[MO90] F. M. Martins and J. N. Oliveira. Archetype-oriented user interfaces. *Computer & Graphics*, 14(1):17–28, 1990.

[Mor90] C. Morgan. *Programming from Specification.* Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[OB97] J. N. Oliveira and L. S. Barbosa. SETS (subtyped) polymorphism as natural transformation. Technical report, (submitted for publication), Jan. 1997.

[OBM97] J. N. Oliveira, L. S. Barbosa, and F. S. Moura. Can distribution be (statically) calculated? Technical report, DI/INESC, 1997. (In preparation).

[OC93] J N. Oliveira and M. Cruz. Formal calculi applied to software component knowledge elicitation. Technical report c19-wp2d, project c.1.9. sviluppo di metodologie, sistemi e servizi innovativi in rete, INESC, 1993.

[Oli87] J. N. Oliveira. Refinamento transformacional de especificações (terminais). In *Actas das XII "Jornadas Luso-Espanholas de Matemática"*, volume II, pages 412–417, Braga, Portugal, 4-8 Maio 1987.

[Oli90] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, 1990.

[Oli92] J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[Oli94] J. N. Oliveira. Hash tables — a case study in $\trianglelefteq$-calculation. Technical Report DI/INESC 94-12-1, DI/INESC, U. Minho, December 1994.

[Oli95] J. N. Oliveira. Fuzzy object comparison and its application to a self-adaptable query mecha nism. In *IFSA'95*, volume I, pages 245–248, 22–28 July 1995. Proc. of the 6th International Fuzzy Systems Association World Congress, S. Paulo, Brazil.

[Rod93] C. J. Rodrigues. Sobre o desenvolvimento formal de bases de dados. Master's thesis, University of Minho, 1993. (In Portuguese).

[RP95] M.A.R. Rebelo and R.M.N. Pinto. Trabalho prático — departamento de planeamento de produção. Technical Report LMCC-Op6-95/1, Universidade do Minho, Feb. 1995. (In Portuguese).

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice-Hall International, 1989.

[SRVN96] A.C. Sampaio, J.M. Rodrigues, L.F. Varandas, and M. Noval. Informatização de uma empresa de construção civil. Technical Report LMCC-Op6-96/1, Universidade do Minho, Jun. 1996. (In Portuguese).

[SS94] Systena and Syntax Sistemi Software. Integrated SOUR Software System — Demo Session Manual. Technical report, SOUR Project, 1994. Ver.1.2, © Systena & SSS, Via Zanardelli 34, Rome & Via Fanelli 206-16, Bari, Italy.

[TH95] S. Thompson and S. Hill. Functional programming through the curriculum. In P. H. Hartel and M. J. Plasmeijer, editors, *Functional programming languages in education (FPLE), LNCS 1022*, pages 85–102, Nijmegen, The Netherlands, Dec 1995. Springer-Verlag.

[VA94] M. Vieira and R. Alves. CAMILA in secondary schools. Technical Report (in Portuguese), Universidade do Minho, 1994.

[Wad89] P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, September 1989.