# A Logico-Categorical Toolkit for Automated Rigorous Software Development

Carlos Henrique C. Duarte[1,2]*

[1] Univ. Estácio de Sá, Rua do Bispo 83, Rio de Janeiro, RJ, 20261-902, Brazil
[2] BNDES, Av. República do Chile 100, Rio de Janeiro, RJ, 20001-970, Brazil

**Abstract.** We describe a toolkit which is devoted to support a rigorous software development approach based on Category Theory and General Logics. The toolkit consists in an integrated collection of interactive theory manipulation and automated reasoning tools implemented using the functional language Haskell. We also present a precise comparison of this with other research efforts.

*Keywords:* Category Theory, General Logics, Automated Reasoning, Formal Methods, Software Development.

## 1 Introduction

Software development processes are inherently very intricate and have become more and more demanding due to the introduction of new technologies and tools that allow us to provide implementations for a widening range of non-functional requirements, such as concurrency, distribution and mobility.

At the heart of some difficulties software engineers face during the development process are the ineffectiveness and inadequacy of many *ad hoc* methods and techniques that are in use today. Ideally, a software engineer should easily experiment with and validate the fuzzy notions gathered during requirement elicitation activities. The manipulation of the validated notions should be traceable from the design to the implementation. In addition, many software artifacts produced in this process should be reusable.

The usual way of satisfying all these needs is the adoption of software tools such as toolkits and integrated development environments (IDEs) that facilitate stepping forward in the process. It happens that no such software tool will be surely effective without a degree of formality, in order, for instance, to avoid the introduction of errors in designs and programs throughout the development.

A still promising approach to ensure the required level of formality in performing software development activities is the integrated use of Category Theory [9] and General Logics [18]. The former advocates the use of categorical notions as a way of putting software artifacts together whereas the latter proposes the adoption of many distinct logical structures in which to formulate and relate

* e-mail: `carlos.duarte@computer.org`; web: `http://chcduarte.webs.com`

these artifacts. The ability to switch logical systems is recognised as a desirable feature in stepwise software development [16].

Conversely, software development approaches are not effective just due to well established formal foundations. In order to instrumentalize a formal development process, a set of software tools should be adopted, ranging from model checkers, proof checkers and theorem provers to facilities for manipulating general logical structures. Respectively, these tools can be adopted at least in the tasks of requirement experimentation and validation, correctness assurance, traceability and reuse. An integrated environment to support verification based and transformation/synthesis based software development would be an ultimate goal in structuring an effective formal approach.

In this paper, we present a collection of software tools (toolkit) devoted to support the approach and the functionalities described above. Certainly, full support for all these needs should be provided by an IDE, but our strategy to achieve this goal eventually is to develop first a very generic toolkit to serve as a basis for further developments. In a way, this research effort can be regarded as an attempt to provide a generic implementation for the theoretic framework that instantiates a paradigm proposed in [11].

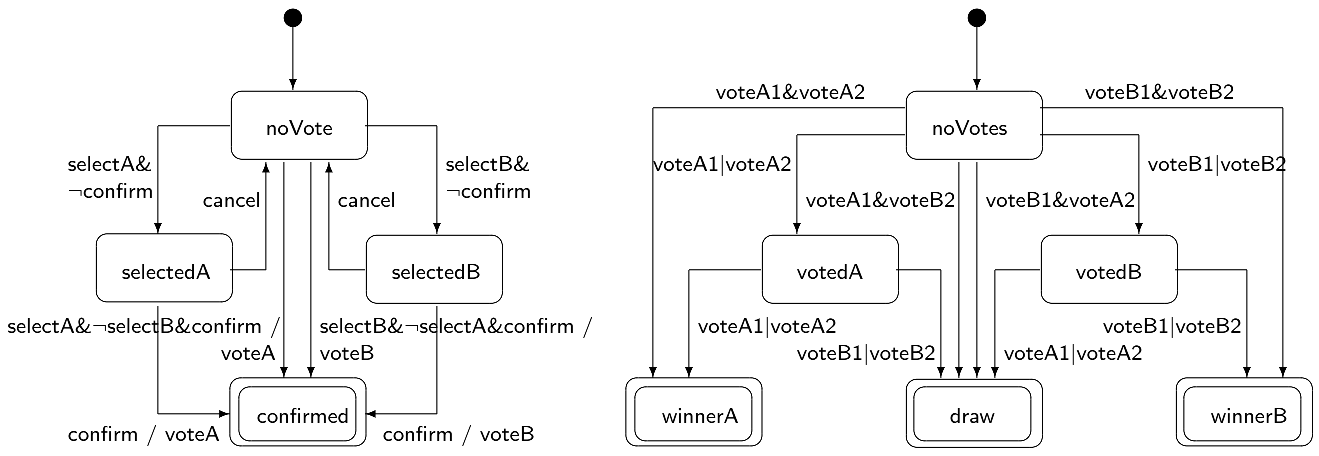
We choose the functional language Haskell [14] as a basis for the proposed developments. The choice of a/the language is justified not only due to the significant amount of symbolic computation required (which we believe to be more precariously supported by other programming paradigms or integrated tools), but also due to the availability of reliable and efficient interpreters and compilers for Haskell, as well as of some automated reasoning tools that can be subsequently integrated into the framework.

**Contributions.** To our knowledge, the described research is the first attempt to provide a full implementation, as part of a generic rigorous software development infrastructure, for all the general logical notions described in [18] (but see Section 6 for related work). In carrying out this research, we have also developed a framework of foundational type classes and abstract data types in Haskell that appears to be an innovation.

**Organization.** Section 2 presents the description of a distributed voting system that is used as an example throughout the paper to illustrate our ideas. Section 3 contains an overview of the categorical and general logical notions that are most relevant for developing our work. Section 4 describes the implementation of the proposed toolkit. Next, in Section 5, we illustrate how this toolkit can be used for rigorous software development. At the end, we present some conclusions and prospects for future research.

## 2 The Distributed Voting System Example

The issue of performing large scale geographically distributed elections with the support of on-line interactive software systems distributed over computer networks has been extensively discussed. We use as an example in this paper the development of a solution for this problem.



(a) Voting Terminal (b) Voting Server

**Fig. 1.** Simplified Voting System Behaviour Specification.

An electronic election process is normally performed with the support of so called voting machines (also called terminals) scattered throughout the whole region involved in the process. There is at least one server connected to each machine so that votes can be collected from electors and dispatched to the server.

There are quite a few requirements that a solution must meet:

1. Each elector may choose at most one out of many candidates;
2. It must be possible for the elector to change its mind before confirming the choice;
3. Once the selection of a candidate is confirmed, it is impossible to change it;
4. Each vote is confidential. In particular, the server only becomes aware of each vote for a candidate, but never of the identity of the respective elector;
5. The election final result is the only information made globally available by the server after the established number of votes are collected;
6. The result of the election may be a winning candidate or it may end in a draw;
7. The solution should be configurable, in the sense that it should be possible to define the (number of) candidates at deployment time;
8. The solution should be scalable, in the sense that it should be possible to increase the number of locations and terminals without design changes.

We choose to structure our design matching the physical components required in the solution. As a matter of simplification, we assume that the election is to be carried out involving just two candidates and two geographically separated electors, consequently requiring the support of only one server and two terminals. Later on, we outline how our solution can be extended, satisfying

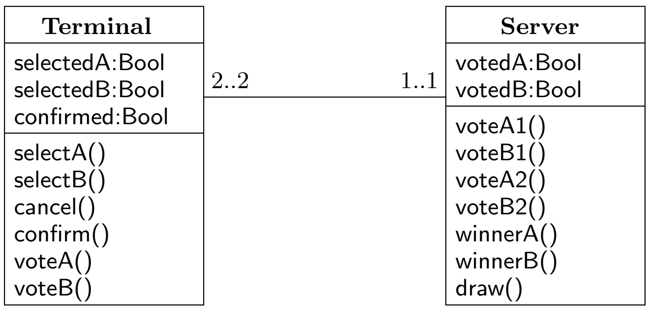


**Fig. 2.** Simplified Voting System Structure Specification.

the configurability and the scalability requirements. We do not address here the dissociation of each terminal from a elector, since it would require a complex unique identification scheme for electors.

The behavior of each kind of object in our solution is described by the state transition diagrams presented in Figure 1. When each voting machine is turned on, no votes have been accounted for. By pressing one of the two selection buttons, the elector may choose out of candidates A and B. It is possible to cancel a selection, but only if this choice is not confirmed yet. A vote is issued with the confirmation. At the server side, votes are counted up to the time when the two electors have confirmed their choices. The interaction between the two kinds of objects is unidirectional and synchronous, with each voting machine possibly generating an event of kind `vote` so that it can be detected by the server.

The rather simple structure of the system with this configuration is depicted in Figure 2. The state of a terminal is determined by the information if either candidate has been chosen and the respective vote confirmed or else if the machine has just been turned on or a vote canceled. The state of the server is determined by the information concerning whether or not a vote was received in favor of each candidate. The methods of each object are those required to perform the state transitions presented in the respective behavior diagrams.

## 3 Categorical and General Logical Notions

Category Theory has been proposed as a foundation for Mathematics and Computer Science alternative to Set Theory. Not only these theories are formulated in substantially different grounds but they also have distinct focuses of concern.

Whereas Set Theory is structured in terms of sets and their elements, Category Theory is founded upon the notion of arrow or morphism between objects. A very pragmatic definition extracted from [23] is presented below:

**Definition 1 (Category).** A *category* is a graph $(O, A, s, t)$ whose nodes $O$ are called objects and edges $A$ are called arrows. When applied to an arrow $f : a \to b$, $s$ returns its domain $a$ and $t$ returns its co-domain $b$. For each $a$ in $O$, there is an identity arrow $i_a : a \to a$ and for each pair $f : a \to b$ and $g : b \to c$ there is another composition arrow $g \circ f : a \to c$ of $f$ with $g$. Moreover, the following equations must hold for all objects $a$, $b$, $c$, $d$ and arrows $f : a \to b$, $g : b \to c$ and $h : c \to d$: $(h \circ g) \circ f = h \circ (g \circ f)$ and $f \circ i_a = f = i_b \circ f$.

Sets, groups and rings, as well as some kinds of specifications, programs and proofs, with their respective morphisms, constitute examples of categories.

The study of General Logics is formulated precisely in terms of categorical notions. Perhaps the most important notions for formulating such study are those of theory (presentation) and morphism. A *theory presentation* consists in a set of symbols, called *signature*, together with a set of sentences, called *axioms*. An easy way of defining presentation morphisms is to propose signature morphisms, which relate symbols in different signatures, and inductively lift them to the sentences in each set of axioms.

Theory presentations are formulated in terms of a language, a morphism that relates the category of signatures to the category defining the shape of sentences (being a functor in this way), and a (reflexive, monotone and transitive) consequence relation $\mid\!\sim$ which, by way of transitive closure, allows us to determine the corresponding *theories*, sets of sentences entailed by the presentation axioms. An *entailment system* is defined precisely by a category of signatures, a language functor and a consequence relation [18]. Entailment systems may be strongly or weakly structural depending on the preservation or not of consequences when translated by lifted signature morphisms [8].

Entailment systems may be defined semantically or proof-theoretically. In order to define a consequence relation semantically, it is necessary to propose a functor associating each signature to a respective category of models (model functor). A satisfaction relation $\models$ is used to attest whether or not a sentence is true in a specific model. A category of signatures, language and model functors and a satisfaction relation for which morphisms preserve truth, seen as a whole, define what is called an *institution* [10]. Seen together with an entailment system with a naturally isomorphic language (that is, there is an isomorphism between the respective language functors, defining a natural transformation in this way) and whose consequence is determined by the satisfaction relation, institutions define what is called a *logic*. Logics defined in this way automatically meet the *soundness requirement*, which states that every entailed sentence is valid, meaning that it is satisfied by all models.

A consequence relation may also be approximated (exactly or not) by a derivability relation $\vdash$, which is normally defined in terms of a set of axiom schemes and inference rules. A derivability relation based on a specific language, together with a corresponding category of proofs, define what is called a *proof calculus*. In some cases, the exact definition of a consequence relation in this way is not possible, due to the logical system not being complete (the converse of the soundness requirement does not hold, in other words). This is not, however, a problem since proof calculi for some logics are useful even without being complete. A logic endowed with a proof calculus determines a *logical system*.

**Specification** CTerminal
  **actions** *select*, *vote*, *cancel*, *confirm*
  **attributes** *selected*, *confirmed*
  **axioms**

$$\mathbf{beg} \to \neg selected \qquad (1.1)$$
$$\mathbf{beg} \to \neg confirmed \qquad (1.2)$$
$$select \to \neg selected \wedge \neg confirmed \wedge \neg cancel \qquad (1.3)$$
$$select \to \mathbf{X}(selected) \qquad (1.4)$$
$$(select \vee cancel) \vee (selected \to \mathbf{X}(selected)) \qquad (1.5)$$
$$(select \vee cancel) \vee (\neg selected \to \mathbf{X}(\neg selected)) \qquad (1.6)$$
$$cancel \to \neg confirmed \wedge \neg confirm \qquad (1.7)$$
$$cancel \to \mathbf{X}(\neg selected) \qquad (1.8)$$
$$confirm \to \neg confirmed \qquad (1.9)$$
$$confirm \wedge selected \to \mathbf{X}(vote \wedge confirmed) \qquad (1.10)$$
$$confirm \vee (confirmed \to \mathbf{X}(confirmed)) \qquad (1.11)$$
$$confirm \vee (\neg confirmed \to \mathbf{X}(\neg confirmed)) \qquad (1.12)$$
$$vote \to selected \wedge confirmed \qquad (1.13)$$
$$confirmed \to \mathbf{X}(\neg vote) \qquad (1.14)$$

$$\begin{tikzcd}
& \text{RawTerminal} \arrow[dl, "\text{RTtoCT}"'] \arrow[dr, "\text{RTtoCT}"] \arrow[dd, dotted, "\text{RTtoST}"] & \\
\text{CTerminal} \arrow[dr, dotted, "\text{CTtoST}"] & & \text{CTerminal} \arrow[dl, dotted, "\text{CTtoST}"'] \\
& \text{STerminal} \arrow[d, dotted, "\text{STtoT}"'] & \\
& \text{Terminal} & \\
\text{Selector} \arrow[uuu, "\text{StoCT}"] \arrow[ur, dotted, "\text{StoT}"] \arrow[dr, "\text{StoR}"'] & & \text{Selector} \arrow[uuu, "\text{StoCT}"'] \arrow[ul, dotted, "\text{StoT}"'] \arrow[dl, "\text{StoR}"] \\
& \text{Regulator} \arrow[uu, dotted, "\text{RtoT}"'] &
\end{tikzcd}$$

**Fig. 3.** Formal Specification of Voting Terminals.

Let us return to our example and present concrete instances of some of these notions. Some others will be exemplified in Section 5.

First note that there is an interesting symmetry in the state transition diagram for voting machines presented in Figure 1, in that the left half of the diagram is identical to the right half, differing only due to the choice of names of intermediary states and of the respective transitions. Symmetries like this usually consist in an opportunity for proposing elegant designs and eventually reduce the required work.

We are going to attempt to define here a voting machine almost only in terms of one of its halves. This can be carried out by proposing a diagram equivalent to that in Figure 1 defined by two connected identical diagrams, one for each candidate, but with some shared structure, namely the initial and final states, as well as the transitions that lead to them. Instead of doing this using diagrams, though, we are going to adopt a temporal propositional logical system.

As expected, we define the half voting machine, here called CTerminal, using a theory presentation. Transition labels in the diagram are captured as action symbols in the presentation signature. State labels are represented by attribute symbols. The resulting signature is presented at the top of Figure 3.

The theory presentation axioms are written using a language defined in terms of signature symbols together with classical and temporal propositional connectives. In particular, the nullary connective **beg** denotes the instant in which the time line begins and **X** denotes the next instant in relation to a given one. The time dependent interpretation of this kind of connective, as well as the standard interpretation of the classical propositional connectives, naturally lead to the definition of an institution, as shown in [4], wherein an associated sound proof calculus is proposed. Such proof calculus shall be presented here in Section 4.

Still concerning our specification, axioms (1.1) and (1.2) respectively state that initially neither has a candidate been selected nor a selection been confirmed. Axiom (1.3) postulates that a selection may only happen if the candidate has not been selected nor the choice confirmed yet, and in particular cannot be performed simultaneously with a cancellation. Moreover, axiom (1.4) says that immediately after a selection, this occurrence is recorded. Note that (1.5-1.6) assert that the value of *selected* may be changed only due to the occurrence of *select* or *cancel*. Axioms (1.7-1.8) and (1.9-1.10) concerning *cancel* and *confirm*, respectively, are analogous to (1.3-1.4) regarding *select*.

A critical part of the specification is the control over occurrences of *vote*. Any confirmation of a selection triggers the announcement of the respective vote (1.10). However, the vote can only be announced if the respective candidate was selected and the choice has been confirmed (1.13). In addition, the status of whether or not a choice has been performed can only be changed if the elector performs a confirmation (1.11-1.12). Axiom (1.14) ensures that this happens only once.

A way of putting two CTerminal specifications together and obtain a description of the voting machines presented in Section 2 is to propose first a mediating theory presentation that contains the corresponding shared structure. This is called RawTerminal here, since it abstracts away the fact that terminals can be used to choose candidates. That is, the signature of presentation RawTerminal contains only the symbols *cancel*, *confirm* and $confirmed$. Since we use theory morphisms to relate these objects (set inclusion functions), not even the symbol names are required to be the same. The axioms of this presentation consist in local versions of (1.2), (1.7), (1.9), (1.11) and (1.12).

Next, in order to obtain a composition of two CTerminal specifications, called STerminal in our diagram, we compute a standard categorical construction called pushout, which consists in the sum of the given presentations amalgamated through their shared symbols and axioms. The resulting presentation is unique up to isomorphism, meaning that any other combination of the given parts along the original morphisms can be shown to be related to the produced amalgamation by a unique isomorphism. A generalization of this categorical construction, called co-limit, can be used to put together many CTerminals, thus showing how to satisfy the configurability requirement.

Our construction is still not complete yet, since the composed CTerminals can interfere into the behavior of one another. Specifically, an elector must be prevented from choosing both candidates simultaneously or another candidate without canceling its previous choice, since this could cause an undesirable behavior in case an attempt of confirmation would be performed later on. That is, the following properties would not be automatically ensured:

$$\mathit{selectA} \to \neg \mathit{selectB} \wedge \neg \mathit{selectedB} \tag{1}$$

$$\mathit{selectB} \to \neg \mathit{selectA} \wedge \neg \mathit{selectedA} \tag{2}$$

$$\begin{tikzcd}
 & \mathsf{Connector} \arrow[dl] \arrow[dr] \arrow[dd, dotted] & & \mathsf{Connector} \arrow[dl] \arrow[dr] \arrow[dd, dotted] & \\
\mathsf{Terminal} \arrow[dr] & & \mathsf{SServer} \arrow[dl] \arrow[d, dotted] \arrow[dr] & & \mathsf{Terminal} \arrow[dl] \\
 & \mathsf{STServer} \arrow[r] & \mathsf{Server} & \mathsf{STServer} \arrow[l] &
\end{tikzcd}$$

**Fig. 4.** Configuration of the Voting System.

To produce the required specification, we use again a pushout construction, this time adopting SELECTORS as mediators, empty axiom set presentations whose symbols are mapped into those of CTERMINAL and REGULATOR, which in turn has (1) and (2) as its only axioms. This specification is called TERMINAL here.

We omit the design of voting servers in this paper, since it can be developed along the same lines, in particular aiming to satisfy the scalability requirement in the same way we designed voting machines to satisfy configurability. The configuration of the whole system is presented in Figure 4.

## 4 Outline of the Toolkit

We have developed an integrated collection of interactive tools to support rigorous software development activities according to the approach described so far. The toolkit can be divided in two parts: a collection of foundational type classes and abstract data types (ADTs), and an implementation in the form of ADTs of the general logical notions described in Section 3. The structure of both parts of the toolkit is illustrated by the UML class diagram presented in Figure 5.

Central to our implementation of foundational type classes and ADTs is the observation that the Haskell language (and other currently existing programming languages that provide support for infinite data objects) ignore some foundational issues concerning Set Theory, thus providing a naive implementation for sets. This is, however, a crucial issue when we attempt to address Category Theory using the programming language, since some of these issues arise already in the definition of what is understood as a category. Specifically, only in particular cases the objects $O$ and the arrows $A$ of a category constitute sets, the reason for making reference to them as collections in general. In order to provide a more faithful treatment of Category Theory, we depart from the built in implementation of sets in Haskell.

The most generic software artifacts of our framework are retainers, containers, closure classes and classes. Their type classes define the interfaces that must be followed by their respective parametric instance types. Retainers are objects that can be used to store data. If facilities for retrieving data from a retainer are

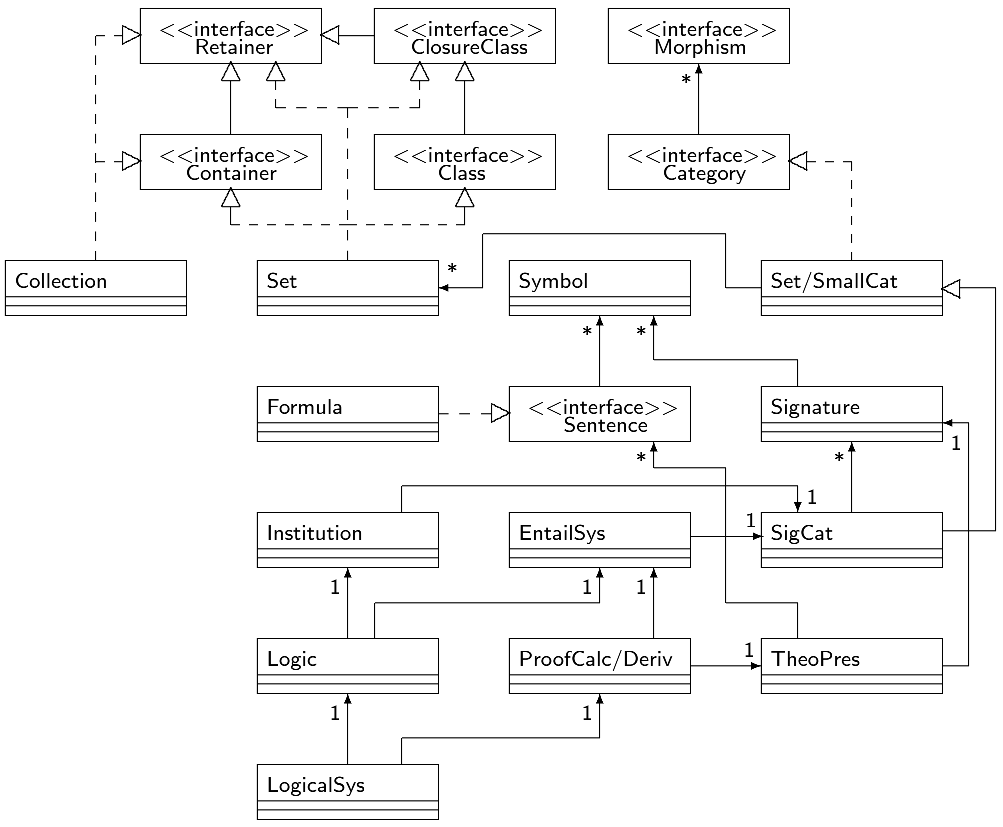


**Fig. 5.** Structure of the Toolkit.

available, then we in fact have a container. Closure classes and classes have the standard interpretation from Class Theory: their instances are collections and sets. The only distinction between closure classes and classes is that the latter also implement higher order methods for computing power classes and iterations of class operations throughout the members of a class.

The other roots of our software artifact hierarchy are morphisms and categories, on the one hand; symbols and sentences, on the other. Morphisms may be defined extensionally or intentionally, in which case a Haskell function may be adopted. Categories are defined in terms of a base data type and are instantiated as small categories or set categories. The corresponding ADTs provide facilities for calculating with categories, such as computing pullbacks and pushouts, limits and co-limits. In their turn, symbols have some structure to carry identifiers, types and possibly some annotations. For instance, the fact that a symbol denotes an attribute or an action as proposed in our specification is recorded as a symbol annotation. Many functions are provided by sentences, such as traversal, normalisation, sub-sentence search and substitution.

The remaining software artifacts of our framework are, *strictu sensu*, implementations of the general logical structures described in Section 3. These implementations support some standard type class interfaces, such as for testing equality, well definedness and for coersing and parsing values. Of particular interest in our work are the categorical facilities for lifting morphisms to higher order structures. For instance, sets of symbol morphisms can be lifted to signature morphisms, which can be lifted to sentence morphisms, which in turn can be lifted to theory presentation morphisms. Lifting is, recognisably, a very important feature in logical frameworks [13, 21].

The facilities for supporting automated reasoning are also an important feature of our toolkit. In fact, the general logical approach facilitates tool implementation and their integration with third party tools, since the interfaces are explicit and well defined. For instance, any automated reasoning tool must conform with one of the following types in order to be able to interact with the tools we have implemented (where `s` denotes any type of `Sentence`):

```
type ModelChecker = Institution s o -> s -> Maybe Bool
type ProofChecker = TheoPres s -> TheoPres s -> Derivation s -> Bool
type AutoTheoProver = TheoPres s -> TheoPres s -> s -> Maybe (Derivation s)
```

In particular, proof checking and theorem proving rely on theory presentations both at the base and meta logical levels, since each proof calculus presentation is given in terms of a meta theory presentation and the support for producing derivations may also depend upon a base level presentation. A full implementation of the calculus adopted in Section 3 is presented in Figure 6.

Concerning the definition of proof calculi, our work is based exclusively on formulae and their sets, since other structures such as lists and bags do not appear to provide substantial extra expressiveness, whereas the implementation of particular list or bag based (sequent) calculi would require a considerable amount of work. Moreover, at the present moment we do not allow auxiliary definitions to be made within a calculus, avoiding the use of meta-level substitution. These definitions must be made at the logical level, using the iff connective.

At the meta-level of our toolkit, each proof calculus definition is treated as a conventional presentation, meaning that logical connectives are defined as meta-signature symbols and inference rules as meta-logical axioms, as show in Figure 6. Meta-theories can be provided as input to a compiler, which performs a type inference process and generates Haskell code afterwards. Sentences written in the corresponding logic are manipulated in this way, as illustrated in Section 5.

Since we support just propositional Hilbert style proof calculi at the moment, we have only implemented an interactive backward proof construction method using a simple unification algorithm [20]. Consequently, the supported proof strategy consists in attempting to unify each goal sentence with an axiom or the conclusion of a proof rule, making any premises as new subgoals. The explicit use of theory presentations and morphisms guides the derivation process and captures any required switch in derivation context.

## 5 Automating Rigorous Software Development

In this section, we present the outline an automated proof of a characteristic property of our simplified voting system, namely that, once an elector votes in some candidate, no vote will ever be issued again by the same elector:

$$\vdash_{\textsc{Terminal}} \mathit{voteA} \vee \mathit{voteB} \to \mathbf{XG}(\neg \mathit{voteA} \wedge \neg \mathit{voteB}) \tag{3}$$

In order to carry out the proof, we use the `metal` tool, which at the present moment integrates under a single textual user interface the whole functionality of our toolkit. Since the development of a derivation using this tool requires the selection of a proof calculus and a base-level theory, we adopt, using the following commands, the calculus presented in Figure 6 of the propositional fragment of the logic presented in [6] and a textual representation of Terminal which resembles the specification in Figure 3:

```
metal 0.1: Welcome!
[MetaL]> gen "IFBTPL.mth"
Generated proof calculus corresponding to the given meta-theory.
[MetaL]> calc IFBTPL
Please wait while (re)loading.
metal 0.1: Welcome!
[IFBTPL]> theo Terminal
[IFBTPL,Terminal]> goal ((voteA || voteB) => X (G ((~voteA) && (~voteB))))
```

The derivation begins with the retrieval of some auxiliary temporal logical lemmas (such as those presented below) that were previously proved at the meta-logical level and are added for subsequent use to the meta-theory currently loaded into the tool, as if they were part of the corresponding proof calculus:

```
[IFBTPL,Terminal] load "HS.thm"
Read {p => q, q => r} |- :: [HS] {p => r}.
[IFBTPL,Terminal] load "ORL.thm"
Read {p => r, q => r} |- :: [ORL] {(p || q) => r}.
[IFBTPL,Terminal] load "ANDR.thm"
Read {p => q, p => r} |- :: [ANDR] {p => (q && r)}.
[IFBTPL,Terminal] load "EXCGX.thm"
Read {} |- :: [EXCGX] {(X (G p)) <=> (G (X p))}
[IFBTPL,Terminal] load "DISTG&.thm"
Read {} |- :: [DISTG&] {(G (p && q)) <=> ((G p) && (G q))}.
[IFBTPL,Terminal] load "DISTX&.thm"
Read {} |- :: [DISTX&] {(X (p && q)) <=> ((X p) && (X q))}.
```

The lemmas above are used in an interactive process wherein the user selects one of the possible unifications (matching the existing proof calculi rules, axioms or lemmas with each goal) so as to advance in the derivation construction:

```
[IFBTPL,Terminal]> choices
1. ORL [p :: [MVAR] |-> voteA, q :: [MVAR] |-> voteB,
        r :: [MVAR] |-> X (G ((~voteA) && (~voteB)))] [1]
2. HS [p :: [MVAR] |-> voteA || voteB,
       r :: [MVAR] |-> X (G ((~voteA) && (~voteB)))] [1]
3. MP-IFBTPL [q :: [MVAR] |->
              (voteA || voteB) => X (G ((~voteA) && (~voteB)))] [1]
[IFBTPL,Terminal]> stepw 1
[IFBTPL,Terminal]> status
current derivation:
1.  voteA => X (G ((~voteA) && (~voteB)))
2.  voteB => X (G ((~voteA) && (~voteB)))
3.  (voteA || voteB) => X (G ((~voteA) && (~voteB)))
    {ORL [p :: [MVAR] |-> voteA, q :: [MVAR] |-> voteB,
          r :: [MVAR] |-> X (G ((~voteA) && (~voteB)))] [1,2]}
```

Note that formulae 1 and 2 above are logically equivalent up to symbol renaming. This allows us to safely focus on the derivation of either of them, say 1, and reuse its proof to justify the other derivation branch using an automorphism.

Also note that the application of ORL above consists in one of the simplest cases of proof rule application, in that no (new) meta-variable is introduced in the derivation. Using Hilbert style, however, it is not always possible to follow a constructive process of this kind. In the application of HS below, for instance, the introduced meta-variables have to be substituted afterwards:

```
[IFBTPL,Terminal]> choices +1
1. HS [p :: [MVAR] |-> voteA,
       r :: [MVAR] |-> X (G ((~voteA) && (~voteB)))] [1]
[IFBTPL,Terminal]> stepw 1
[IFBTPL,Terminal]> status
current derivation:
1.  voteA => _q1 :: [MVAR]
2.  _q1 :: [MVAR] => X (G ((~voteA) && (~voteB)))
3.  voteA => X (G ((~voteA) && (~voteB)))
    {HS [p :: [MVAR] |-> voteA,
         r :: [MVAR] |-> X (G ((~voteA) && (~voteB)))] [1,2]}
4.  voteB => X (G ((~voteA) && (~voteB)))
5.  (voteA || voteB) => X (G ((~voteA) && (~voteB)))
    {ORL [p :: [MVAR] |-> voteA, q :: [MVAR] |-> voteB,
          r :: [MVAR] |-> X (G ((~voteA) && (~voteB)))] [3,4]}
[IFBTPL,Terminal]> subst [_q1 :: [MVAR] |->
                          X (G (~voteA)) && X (G (~voteB))] [1,2]
[IFBTPL,Terminal]> status +2
1.  voteA => (X (G (~ voteA))) && (X (G (~ voteB)))
2.  ((X (G (~voteA))) && (X (G (~voteB)))) => X (G ((~voteA) && (~voteB)))
```

The last sentence labeled with 2 above is a direct consequence of DISTX&, DISTG& and GV1 under rules GENG, MP and HS. In its turn, the last sentence labeled with 1 above is obtained from an ANDR application:

```
[IFBTPL,Terminal]> choices +1
1. ANDR [p :: [MVAR] |-> voteA, q :: [MVAR] |-> X G (~voteA),
         r :: [MVAR] |-> X (G (~voteB))] [1]
[IFBTPL,Terminal]> stepw 1
[IFBTPL,Terminal]> status +3
1.  voteA => X (G (~voteA))
2.  voteA => X (G (~voteB))
3.  voteA => (X (G (~voteA))) && (X (G (~voteB)))
    {ANDR [p :: [MVAR] |-> voteA, q :: [MVAR] |-> X G (~voteA),
           r :: [MVAR] |-> X (G (~voteB))] [1,2]}
```

Goal `1` above can formulated entirely in the language of CTerminal, by mapping *vote* into *voteA* using the `CTAtoST` ∘ `STtoT` morphism. If this were the only goal of our derivation, it would be possible to prove it using this presentation as a derivation context. In this way, the automated derivation could proceed by issuing the following commands, which would switch the derivation context to CTerminal using the respective morphisms and presentations:

```
[IFBTPL,Terminal] retracw STtoT STerminal
[IFBTPL,STerminal] retracw CTAtoST CTerminal
```

The proof of `voteA => X (G (~voteA))` can be outlined by first arguing that *vote* can only happen if a choice has already been *confirmed* (1.13) and, in this circumstance, another vote cannot be issued in the next moment (1.14). Moreover, in view of (1.9) and (1.11), the logical value of this attribute never changes again. Therefore, after the moment when *confirmed* becomes true, *vote* is forbidden to happen forever. These two axioms also entail `voteA => X (G (~voteB))` in the Terminal context. The derivation of these facts is omitted here.

The supported forward proof checking process is analogous to the above, but having as a goal to validate or find in an automatic manner the set of unifiers for each derivation step.

## 6 Related Work

Our work is related at least to two distinct research subjects: the development of automated reasoning tools and the development of integrated environments for rigorous software development.

An example of the first family of tools is the Alfa proof editor [12], which supports direct manipulation of derivations in a logical framework based on Martin-Löf's Type Theory. The tool is interactive and allows one to define meta-theories (axioms and inference rules), formulate theorems and construct derivations. Alfa uses the Agda proof checker and both are implemented using Haskell.

Another example of automated reasoning tool is Isabelle [21], which is a generic theorem prover strongly influenced by the design of LCF and consequently based on the functional programming language ML. A LCF theory is presented in terms of types and functional symbols of some signature. Moreover, LCF adopts the proofs as types paradigm and represents object level proof rules by functions. Proof tactics are represented as terms that are provided as input to an interactive proof engine. Isabelle has been used to automate a considerable number of proof calculi.

Despite the fact that Isabelle was developed to be a generic theorem prover, there is substantial evidence that it is also an adequate tool for supporting rigorous software development. Many formal methods, such as VDM [1] and Z [15], have been automated using this tool. The development of concurrent and distributed systems has also been studied using Isabelle, as show in the mechanisation of UNITY [22] and TLA [17].

The main distinction between these tools and the toolkit described here is the role attributed to signatures and theory presentations. These tools both support signature and presentation definition, but do not treat them explicitly as first class objects in the reasoning process. In turn, our work has been influenced by the axiomatic and the ADT schools of software development, which focus on logical presentation independently from any implementation platform [13]. The categorical treatment of these and related notions is particular to our work.

The use of categorical notions allows us to give a more systematic treatment to presentation operations such as extension and merging, by using constructions like inclusion morphisms and co-limits. We also foresee an improved integration with so called oracles, which are external tools that may be used for supporting additional automated reasoning functionalities, since the implemented general logical structures already define the interfaces for supporting this kind of integration. We formalise the integration itself through some consequence relation, which can be defined semantically or proof-theoretically.

Substantial research on providing automated reasoning support for rigorous software development has also been performed using Maude [3], which is based on rewriting logic as a general logical framework. An inductive theorem prover [2] and a model checker for linear time logic [7] are among the available tools.

Some attempts have been made to provide support for rigorous software development approaches based on category theory using IDEs. Among these, KIDS [24] and more recently SPECWARE [25] appear to consist in the most elaborated tools. Both support a transformational process which begins with the formulation of a set of specifications and morphisms to represent the application domain using a specific higher order logical system and proceeds by categorical stepwise refinement. This process targets the generation of code based on variants of Lisp. The correctness of the process is justified in terms of the source and the programming logical systems. We believe that future versions of our toolkit could benefit from the reported results on categorical design tactics, data type refinement and program optimization implemented by these environments.

Finally, concerning the manipulation of categorical notions using software tools, [23] presents a full implementation of these notions using the programming language ML, which is not lazy and consequently does not provide faithful support to infinitary data objects. Isabelle was adopted in [19] instead, but the considerable distinction between the produced automated proofs and those developed manually was stressed. This has been a major concern in our research.

## 7 Final Remarks

In this paper, we have presented a toolkit devoted to support rigorous software development activities based on Category Theory and General Logics.

Currently, apart from providing an implementation of the underlying foundational definitions aiming to support toolkit extensibility, the following functionalities for propositional logics presented in Hilbert style are supported:

1. Theory presentation type inference;
2. Categorical theory presentation manipulation;
3. Proof checking;
4. Interactive theorem proving;
5. Persistence and pretty-printing;

The decision of beginning the development of the toolkit with a focus on propositional logics was due to their simpler structure, facilitating the implementation of the notions described here. The choice of Hilbert style proof calculi was due to the fact that almost all logics can (also) be defined in this way. All the propositional proof calculi listed in [4] have been implemented using this toolkit.

These decisions do not constrain the future development of our toolkit. In particular, we are currently developing the implementation of a higher order resolution algorithm that will support theorem proving in first and higher order logics. In addition, we do not foresee any difficulty in providing support for natural deduction and sequent calculi for logics whose derivations are presented in this way. Yet another goal is to formalize the meta-level of our toolkit.

The real challenge ahead is concerning how to provide systematic and formalised support for model theory. This subject has been much less explored in the literature and requires substantially more research. Another research direction is providing support for derivable proof rules, as adopted in rely-guarantee disciplines of software development [5], which require specific derivation styles.

## References

1. Sten Agerholm and Jacob Frost. An Isabelle-based theorem prover for VDM-SL. In E. Gunter and A. Felty, editors, *Proceedings of the 10th Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, August 1997.
2. Manuel Clavel, Francisco Durán, Steven Eker, and José Meseguer. Building equational proving tools by reflection. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.
3. Manuel Clavel et al. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
4. Carlos Henrique C. Duarte. *Proof-Theoretic Foundations for the Design of Extensible Software Systems*. PhD thesis, Department of Computing, Imperial College, London, UK, 1998.
5. Carlos Henrique C. Duarte and Tom Maibaum. A rely-guarantee discipline for open distributed systems design. *Information Processing Letters*, 74(1–2):55–63, April 2000.

6. Carlos Henrique C. Duarte and Tom Maibaum. A branching-time logical system for open distributed systems development. *Electronic Notes on Theoretical Computer Science*, 67, 2002.
7. Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:143–168, 2002.
8. José Fiadeiro and Amilcar Sernadas. Structuring theories on consequence. In Donald Sannella and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 332 of *Lecture Notes in Computer Science*, pages 44–72. Springer-Verlag, 1988.
9. Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.
10. Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.
11. Armando Haeberer and Tom Maibaum. The very idea of software development environments: A conceptual architecture for the ARTS environment paradigm. In *Proc. 13th Conference on Automated Software Engineering (ASE'98)*, pages 260–271. IEEE Computer Society Press, October 1998.
12. Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR'2000)*, number 1955 in Lecture Notes in Computer Science, pages 70–84. Springer Verlag, November 2000.
13. Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
14. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.
15. Kolyang, Thomas Santen, and Burkhart Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proc. 9th Conference on Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 283–298. Springer Verlag, 1996.
16. Tom Maibaum and Wladyslaw Turski. On what exactly is going on when software is developed step-by-step. In *Proc. 7th Conference on Software Engineering (ICSE'84)*, pages 525–533. IEEE Computer Society Press, March 1984.
17. Sthepan Merz. Mechanizing TLA in Isabelle. Technical report, University of Maribor, Slovenia, 1995.
18. José Meseguer. General logics. In Hans Dieter Ebbinghaus et al., editors, *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
19. Greg O'Keefe. Towards a readable formalisation of category theory. *Electronic Notes in Theoretical Computer*, 91:212–228, 2004.
20. Lawrence C. Paulson. Designing a theorem prover. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 415–475. Oxford University Press, 1992.
21. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
22. Lawrence C. Paulson. Mechanizing a theory of program composition of UNITY. *ACM Transactions on Programming Languages and Systems*, 23(6):1–30, November 2001.
23. David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice-Hall, 1988.
24. Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

25. Yellamraju V. Srinivas and Richard Jüllig. SPECWARE$^{TM}$: Formal support for composing software. In B. Moller, editor, *Proc. Mathematics of Program Construction (MPC'95)*, volume 947 of *Lecture Notes in Computer Science*, pages 399–422, 1995.

```
({ :: IFBTPLFormula [SORT],
   ~ :: IFBTPLFormula -> IFBTPLFormula [CONN],
   => :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],
   || :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],
   && :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],
   <=> :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],

   Beg :: () -> IFBTPLFormula [CONN],
   V :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],
   X :: IFBTPLFormula -> IFBTPLFormula [CONN],
   F :: IFBTPLFormula -> IFBTPLFormula [CONN],
   G :: IFBTPLFormula -> IFBTPLFormula [CONN],
   U :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],
   W :: (IFBTPLFormula, IFBTPLFormula) -> IFBTPLFormula [CONN],

   A :: IFBTPLFormula -> IFBTPLFormula [CONN],
   E :: IFBTPLFormula -> IFBTPLFormula [CONN]},

 {{} |- :: [ORDEF-IFBTPL] {(p || q) <=> ((~p) => q)},
  {} |- :: [ANDDEF-IFBTPL] {(p && q) <=> ~(p => ~q)},
  {} |- :: [IFFDEF1-IFBTPL] {(p <=> q) => (p => q)},
  {} |- :: [IFFDEF2-IFBTPL] {(p <=> q) => (q => p)},

  {} |- :: [WEAK-IFBTPL] {p => (q => p)},
  {} |- :: [DIST-IFTLPL] {(p => (q => r)) => ((p => q) => (p => r))},
  {} |- :: [CONT-IFBTPL] {((~p) => ~q) => (q => p)},

  {p, p => q} |- :: [MP-IFBTPL] {q},

  {} |- :: [XDEF-IFBTPL] {(X p) <=> (p) V (~ (p => p))},
  {} |- :: [FDEF-IFBTPL] {(F p) <=> (p || (p V (p => p)))},
  {} |- :: [GDEF-IFBTPL] {(G p) <=> ~(F (~p))},
  {} |- :: [UDEF-IFBTPL] {(p U q) <=> (q || (p && (q V p)))},
  {} |- :: [WDEF-IFBTPL] {(p W q) <=> ((G p) || (p U q))},

  {} |- :: [GV1-IFBTPL] {(G (p => q)) => ((p V r) => (q V r))},
  {} |- :: [GV2-IFBTPL] {(G (p => q)) => ((r V p) => (r V q))},
  {} |- :: [VVR-IFBTPL] {(p V q) => (p V (q && (p V q)))},
  {} |- :: [VVL-IFBTPL] {((p && (q V p)) V p) => (q V p)},
  {} |- :: [LIN-IFBTPL] {((p V q) && (r V s)) =>
                         (((p && r) V (q && s)) ||
                         ((p && s) V (q && s)) ||
                         ((q && r) V (q && s)))},
  {} |- :: [DISTV-IFBTPL] {((p || q) V r) => ((p V r) || (q V r))},
  {} |- :: [INFT-IFBTPL] {X (p => p)},
  {} |- :: [INIT-IFBTPL] {~ (X Beg)},

  {p} |- :: [GENG-IFBTPL] {G p},
  {Beg => G p} |- :: [INDG-IFBTPL] {p},

  {} |- :: [EDEF-IFBTPL] {(E p) <=> ~(A (~p))},

  {} |- :: [AMON-IFBTPL] {(A (p => q)) => ((A p) => (A q))},
  {} |- :: [AREF-IFBTPL] {(A p) => p},
  {} |- :: [AS5-IFBTPL] {(E p) => A (E p)},
  {} |- :: [EV-IFBTPL] {((E p) V q) => E (p V q)},
  {} |- :: [AFIX-IFBTPL] {(A (p => (X (q U p)))) => (p => X (A (q U p)))},
  {} |- :: [EBEG-IFBTPL] {(E Beg) => Beg},

  {p} |- :: [GENA-IFBTPL] {A p}})
```

**Fig. 6.** Proof Calculus Presentation for an Initialised Future Time Propositional Logic.