ICC Final Paper Cover Sheet

Title:	**Formal Concepts and Tools** **for** **Object-Oriented Analysis and Design**
	Full paper

Author:	Pascal André & Jean-Claude Royer
Address:	IRIN Equipe de Recherche en Technologie Objet Université de Nantes - Faculté des Sciences et des Techniques 2 rue de Houssinière 44072 NANTES Cedex 03 FRANCE
E-mail:	(andre,royer)@unantes.univ-nantes.fr
Telephone: Fax:	(33) 40.37.30.22 (33) 40.29.32.51

Number of pages:	27
Keywords:	Object-Oriented Formal Specifications, Abstract Data Types, Formal Classes, Refinement, Specification Environments.
Category:	Research paper
Topics:	Object-Oriented Tools and Methods

Abstract:

Object-orientation and formal methods are promising approaches for software with quality. Previous works deal with a formal interpretation of the object model in current formal specification framework. We assess that three abstraction levels are required to enforce the separation of concerns : writing understandable formal specification and designing formal classes to be implemented in more than one OOP language. Tools for writing, proving and refining formal object-oriented specifications are also lacking. This paper presents some formalisms and tools to address these keypoints.

Formal Concepts and Tools for Object-Oriented Analysis and Design

1 Introduction

This paper is a contribution to fill up the gap between object-orientation and formal methods. Object-Oriented programming has proved its interest in software engineering. On one hand, current object-oriented analysis and design methods are helpful guidelines for engineers. On the other hand, formal methods are necessary to increase reliability and correctness in the object-oriented software development [12, 36, 26]: precise definitions, proof obligations, specification refinement, rapid prototyping, software validation, etc.

Previous works focus on the theoretical framework for object-oriented formal models using algebraic specifications [10, 18, 8] or Z-style (or VDM) specifications [1, 27, 14, 26, 15]. Many of the proposals quoted below are still under development. We still feel that formal specifications are currently far from the object-oriented software engineering because of the recondite notations, the lack of methods and tools for writing, proving and refining specifications. Furthermore, we need abstraction: abstraction towards implementation languages, abstraction towards any data structure representation.

So we propose a specification environment that focus on:

- a natural way of writing formal specification. We use graphical and textual notations and we introduce formalism step by step to do not disconcert software engineers.
- interactive tools to prove properties or to execute specifications,
- incremental refinement from specification to object-oriented programming (OOP). Three layers allow a separation of concerns: specification model (GAT stands for Graphic Abstract data Types), formal class model (FC) and object-oriented programming languages. We develop an heuristic method and interactive tools to extract formal classes from GAT [5] and to implement formal classes using object-oriented programming languages,
- an opened system to communicate with other specification or proof environments (automata, algebraic specifications, rewriting).

These concepts and tools can be adapted and included in CASE Tools of well-known methods such as [35, 34].
In this paper, we shortly present the GAT and FC models in section 2 and the transition between these models (section 3). Section 4 introduces some life cycle issues. Last we overview the specification environment in section 5.

2 The Models

To achieve reusability, readability and portability of software components, we need different levels of abstraction. The highest level must be easily understood by any client wanting to use a component in an application. The lowest level corresponds to object oriented programming, as in Eiffel, C++ or Smalltalk. We claim that an intermediate level is required to provide an adequate scheme to design portable components with strict software engineering principles (modularity, static type checking, inheritance rules, etc.).

Hence, we defined two abstraction models to meet these requirements:

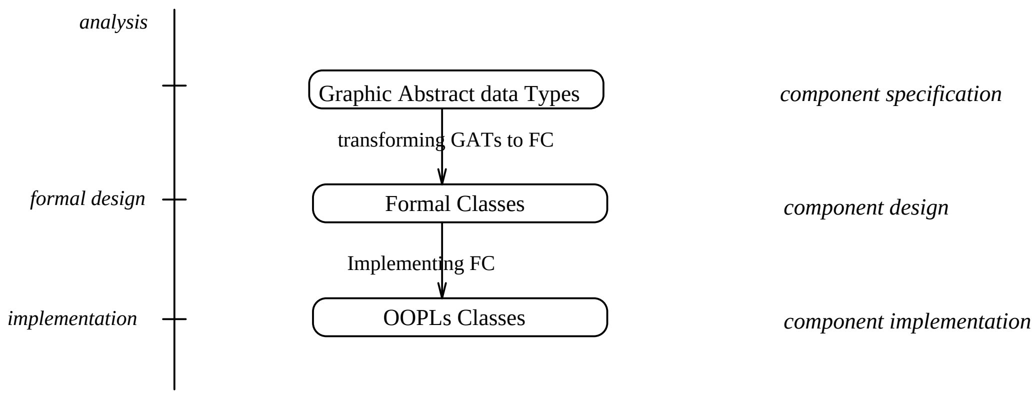

Figure 1 : Abstraction in the Component Life Cycle

These models are mainly based on concepts and notations of algebraic specifications [17, 13]. We illustrate this overview by the specification of a simple tank. This is not a demonstration of a good modeling but rather a trivial example to understand the interest of our models. The tank is filled up gradually with liquid. When filling up and the tank is becoming full, the rest of the liquid overflows. The tank is emptied from time to time. The capacity of the tank must be a non-null integer.

2.1 The Graphic Abstract Data Type Model

The main motivations of GATs are to smoothly introduce formal specifications of data types in object-oriented programming and to facilitates specification writing and reading. We present here the main features of the GAT model, see [4] for further informations.

2.1.1 Basic Model

A graphic abstract type is a specification of an abstract data type through a set of operations and a guarded finite state machine that models the conditions of operation invocations. The type specified is called the *type of interest*. The syntactic structure of a GAT is

```
GAT Tank is
    <header>
    <parameter declaration>
    <operation profile>
    <constraint>
    <dynamic behaviour specification>
ENDGAT
```

where *header* denotes specification documentation such as GAT name, authors, dates, comments. Constants and types parameterize GAT specifications. Operations are declared by their profile which corresponds to a signature with variables. The constraint is an invariant predicate. The dynamic behaviour specification describes an order on operation call and provides an object flavor to the model. Transitions are controlled by named guards, which are first order predicates limited to Horn clauses.

A graphic view of our example is given in figure 2. State name `ET` stands for empty tank, `IT` for intermediate tank and `FT` for full tank. The tank is parameterized by type `Liquid` and by constant `capacity`. Any effective type for `Liquid` must declare an operation `volume`.

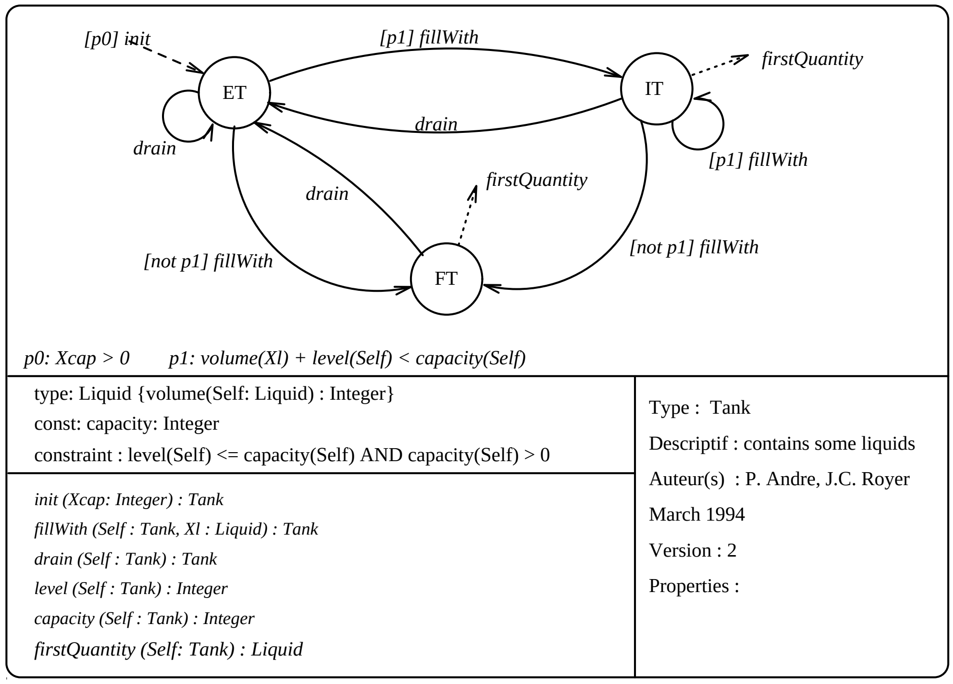

Figure 2 : GAT Specification of a Tank

GAT operations are functions, defined by their profiles (parameters and resulting type). The type of interest (`TI`) appears in every profile either in parameters or in result. When `TI` belongs to the parameters, the associated variable, noted `Self` is the first parameter. This is another object flavor for interpreting operation call as message send and `Self` as the receiver. The operations are classified according to their profile. Operations which resulting type is `TI` are called **constructors**. The operations create values of type `TI`. The other operations are called **observers**.

The finite state machine describes the conditions on partial operation call. Recall that "A type is characterized by one or more sets of values and the operations which are allowed on the values" [20]. A state is a subset of the type such that only some values and operations are available. A state corresponds to the notion of subsort in OBJ [19] or to algebras as in COLD [18]. Operations that do not appear on the finite state machine are said to be **neutral**, i.e. they are total operations and do not change the state. Observers are dotted-line arrows with no final state. Constructors are full-line arrows between two states. Basic constructors are dashed-line arrows arriving in an initial state. Guards are expressed from observers and parameters.

Syntactic sugar reduces combinatorial explosion of states when describing complex data types. We defined a generalisation relation on states to group common transitions.

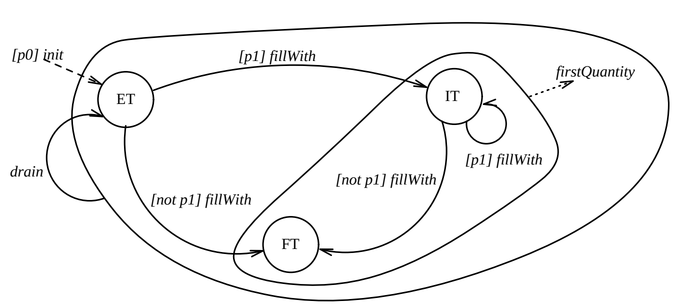

Figure 3 : Generalization of the transitions

Operation `drain` is a transition from a superstate containing all states into the initial state `ET`. Other constructions permit to group highly coupled states in an aggregate state, to combine operations with usual operators (sequence, parallel). Further extension may include state aggregation as in Statecharts [22].

2.1.2 Interpretation

A graphic abstract type is a mixing between object, abstract data types and state-transition systems. No representation like instance variables is given, but dynamic properties (except on real time features) are given. The automaton describes the life cycle. The semantics of GATs is given through algebraic specifications.

Algebraic specification of data type A GAT is a first step in the specification of one abstract data type. Its particular representation is used to build interactively the algebraic axioms which provide the meaning of the GAT. Note that the algebraic specification is optional since a more operational semantics can be given through formal classes refinement.

Parts of the algebraic specification are extracted from the GAT descriptions: clientship relation (USE), signature (operation profiles), preconditions. The constraint is a predicate that can be included in preconditions of operations or be informative only. For flat specifications langages (i.e. those which do not have a subsort construction), the dynamic behaviour is represented by a set of boolean total observers, one per state. A full specification of the tank is given in appendix A.

$$\left.\begin{array}{ll} init & : Integer \rightarrow Tank \\ fillWith & : Tank\ Liquid \rightarrow Tank \\ drain & : Tank \rightarrow Tank \end{array}\right\} constructors$$
$$\left.\begin{array}{ll} level & : Tank \rightarrow Integer \\ capacity & : Tank \rightarrow Integer \\ firstQuantity & : Tank \rightarrow Liquid \end{array}\right\} observers$$
$$\left.\begin{array}{ll} ET & : Tank \rightarrow Boolean \\ IT & : Tank \rightarrow Boolean \\ FT & : Tank \rightarrow Boolean \end{array}\right\} state\ observers$$

```
% constraint
    constraint(Self) == level(Self) <= capacity(Self) AND capacity(Self) > 0
% guards
    p0(Xcap) == Xcap > 0
    p1(Self, Xl) == volume(Xl) < capacity(Self) - level(Self)
precondition(init(Xcap)) == p0(Xcap)
precondition(fillWith(Self, Xl)) == ET(Self) OR IT(Self)
%   simplification of [ET(Self) AND p1(Self, Xl)] [ET(Self) AND NOT p1(Self, Xl)] OR
%        [IT(Self) AND p1(Self, Xl)] [IT(Self) AND NOT p1(Self, Xl)]
precondition(drain(Self)) == true
precondition(firstQuantity(Self)) == IT(Self) OR FT(Self)
precondition(capacity(Self)) == true
precondition(level(Self, Xw)) == true
```

Figure 4 : Signature of Type `Tank`

Algebraic axioms fulfill this description in giving an abstract semantics to operations. Axioms are positive conditional axioms, we note `cond ==> eq`, where `cond` is a conjunction of equations and `eq` is an equation `t == u` where `t` and `u` are algebraic terms.

As in algebraic specifications, we use an induction principle ([29]) in GATs. We choose a kernel set of primitive operations (resp. primitive methods) from the signature (resp. behaviour). Primitive constructors are called **generators**. They are sufficient to create all the values of the type. Secondary operations are defined on primitive ones. We assume that generators do not have circular dependent types (as in [31]).

We defined algorithms using paths in the automata to interactively build the axiomatic. Several steps have to be achieved :

- finding the primitive operations. An algorithm computes all the sets of constructors such that each state can be reached using only these constructors. We choose the smallest subset to be the family of generators (this heuristic is sufficient in practice). Primitive observers are those that cannot be defined by other observers (heuristic only). The generators are {`init, fillWith`} in the example.
- compute the axioms of the state predicates by the union of previous states (according to the generators) and guards. For example :
  ```
  IT1:  IT(init(Xcap)) == false % whenever init is defined
  IT2:  IT(fillWith(Self,Xl)) == [p1(Self, Xl) AND ET(Self)] OR [p1(Self, Xl)
  AND IT(Self)]
  ```
- define primitive observers from generators (i.e. finding the axioms) using equational theorems or induction. The domain of partial operations is defined using preconditions (guards and state predicates).
- define secondary constructors from generators.
- define secondary observers from primitive observers and constructors.

Our method is close to Guttag's safe axiomatization [21] and Bidoit's fair presentations [9] applied to partial operations.

To find the axiom is to compute paths in the automata such a way that the right term is given by the specifier as soon as he can. This axiom construction is what we call **G-derivation**. It consists in examining the effect of a generator on an operation, from a given state. G-derivation is a systematic axiom construction on definition domains.

For total operations (included neutral one), the semantics of an operation can be given through its effects on each generator or studied for each state. For example, the axiomatic of operations `capacity` and `level` is

```
capacity1- capacity(init(Xcap)) == Xcap
capacity2- capacity(fillWith(Self,Xl)) == capacity(Self)
level1:  ET(Self) == true ==> level(Self) == 0
level2:  ET(Self) == true & p1(Self, Xl) == true ==> level(fillWith(Self,Xl)) == volume(Xl)
level3:  IT(Self) == true & p1(Self, Xl) == true ==> level(fillWith(Self,Xl)) == volume(Xl)
+ level(Self)
level4:  FT(Self) == true ==> level(Self) == capacity(Self)
```

State-transition system A GAT can also be interpreted as a state-transition system. We defined a graphic notation for final states (in fact final transitions). It has no meaning in abstract data types since it means the no-value creation but corresponds to the `destructor` operator as in many object-oriented programming languages. A language is associated with the state-transition system to recognize well-formed terms.

No semantics of concurrent GAT is given now. Communications and concurrency between GAT has been evaluated through synchronized automata. But special provisions are needed to deal with internal/external events and input/output events together with the algebraic semantics.

2.1.3 Formal Reasoning

Syntactic verifications prove the consistency between the signature and the dynamic behaviour. Tools are needed to simplify the terms of the axioms and to merge similar axioms.

Formal reasoning is about general properties on the specification such as completeness, consistency or liveness, and application specific properties. These properties can be checked by traditional algebraic environments like LARCH, Asspegique, OBJ, etc [16].

An example of such proof is the proof of the invariant: `level X <= capacity X` in the algebraic specification (Appendix A).

This is an inductive theorem because a Tank is an inductive structure with two generators `init` and `fillWith`.

The automaton and rewriting system can help us to do the proof. We must prove the invariant on the three states of the automaton reduced to the generators (as in G-derivation).

Let `ET X = true` then we must prove
`level(init s(m:Integer)) <= capacity(init s(m:Integer))`, where `s(m)` means the successor of integer `m`. the symbolic evaluator of Asspegique (in graphic mode) produces:

```
expression :
inf level init s m:Entier  capacity init s m

The expression to be evaluated is ...

   __inf_____
  /         \
level_   capacity_
  |         |
init_      init_
  |         |
  s_        s_
  |         |
  m         m

    [ level (init (s m)) -->> 0 ] .

This rule let us get the expression :

  _inf___
 /       \
0    capacity_
         |
       init_
         |
         s_
         |
         m

    [ capacity (init (s m)) -->> s m ] .

This rule let us get the expression :

  inf__
  /   \
 0     s_
       |
       m

    [ inf 0 (s x) -->> true ] .

This rule let us get the expression :
```

`true`

Let `FT X = true` then `level X = capacity X` by axiom `level4`, hence the property is true.

Let `IT X = true` then there is two axioms (`level2, level3`) which reach this state. By axiom `level2`, `level X = level (fillWith Self Xl) = volume Xl` and `volume Xl < capacity Self, by axiom capacity2, capacity Self = capacity fillwith Self Xl = capacity X`. The same way applies with the other axiom. This is a simple proof where induction is hidden by the use of the G-derivation. In more complex proof a powerful system like Larch Prover is required.

2.1.4 Extensions

In the previous sections we focused on the basic model, based on algebraic data types with genericity and clientship. Other issues have been studied, some having a correspondence in algebraic specifications (inheritance, hidden sort and operations) other do not (concurrency).

The model has been extended to inheritance. The import part is computed from the operations profile while the inheritance declaration permits specification refinement and reuse. We separate signature inheritance and behavioural inheritance. We defined inheritance cases, close to subtyping, for signature inheritance :

- adding a new operation,
- strengthen the constraint,
- redefining an operation profile with co-variance rule on the receiver and the resulting types,

Modifying the semantics of operations, which corresponds to method redefinition is hardly verifiable.

In the dynamic part, inheritance is close to aggregation. Modifications are

- adding a new state : this corresponds to a superclass state decomposition,
- adding a new transition.

The dynamic behaviour inheritance is based on the observational equivalence. It means that forgetting the additional operations in the subclass leads to equivalency between the two automata. This semantics is supported by finite state systems algebra. Syntactic control are possible.

2.1.5 Benefits

On one hand, this is a natural and abstract way to build an algebraic specification, it is more abstract and user friendly than OS [10] or COLD-K/AS[18] or Z-style languages. The specification is easy readable. It makes incremental definition possible through state and transition addiction. Modeling error by special states (and terms) is also natural. Hence the two usual means for specifying exceptions and errors are possible and natural in our model : partial definition domain or special erroneous terms. On the other hand, the GAT model is an attempt to join dynamic and functional aspects of the specification in an unified scheme. Automata are more readable and more writable than temporal logics for engineers. The GAT model is an interesting means for documenting the use of components.

2.2 The Formal Class Model

The formal class model [2, 6] is an algebraic, abstract and minimal model common to OOP languages with classes. Its main motivations are the formal specification and design of reusable and portable classes. It does facilitate reverse engineering. Furthermore it does make proof, symbolic evaluation and simple implementation in OOP languages possible. The basic model is functional but an extension with side effects exists [2].

A formal class is an abstraction of concrete classes. It is a particular and simplified algebraic data type.

2.2.1 An Example of Formal Class

A class presentation is a "box". The information provided about a class consists of its name, inheritance links, exported and imported features, aspect and extensions

IntermediateTank	
inherits from `NONEMPTYTank NONFULLTank`	
comments: An intermediate tank is neither empty nor full	
features: `capacity, level`	
from: `Integer` **import:** `+, 0, <`	
from: `Liquid` **import:** `volume`	
aspect : tank	
field selectors	**constraint**
`oldState : IntermediateTank` ⟶ `NONFULLTank`	`0 < capacity(Self) AND`
`fluid : IntermediateTank` ⟶ `Liquid`	`level(Self) < capacity(Self)`
extensions	
;; `capacity` : maximum capacity `capacity` : `IntermediateTank` ⟶ `Integer` `capacity(Self) == capacity(oldState(Self))`	
;; `level` : current level `level` : `IntermediateTank` ⟶ `Integer` `level(Self) == level(oldState(Self)) + volume(fluid(Self))`	
;; `fluidAux` : auxillary method for firstQuantity `fluidAux` : `IntermediateTank Liquid` ⟶ `Liquid` **var:** `X1 : Liquid` `fluidAux(Self, X1) == fluidAux(oldState(Self),fluid(Self))`	

Figure 5 : A Formal Class Example

Such a class defines an algebraic specification. An instance is described by a term built on method profiles. To create an instance we use a `new` method with the class name and values. For instance:

```
cuve = newIntermediateTank(oldState= newEMPTYTank(5), fluid= newLiquid(3))
```

Keywords are allowed in primitive methods to clarify messages. Message sending is written as a functional call:

```
<selector>(<receiver> <, argument>*).
```

We only use single selection (single dispatch). Now we can send messages like `fillWith(drain(cuve), newLiquid(2))`.
A formal class has two main parts: an **aspect** and **extensions**.

2.2.2 Aspect

An aspect is an abstract and readable description for objects. It has two parts: an abstract structure and a constraint. The abstract structure is a set of **field selectors**. Each field selector is an observer with profile: $\texttt{fsel}_i : \texttt{CFC} \longrightarrow \texttt{T}_i$, where `CFC` means current formal class. The **constraint** is a predicate on the abstract structure. The constraint can be seen as a condition for creating or modifying an object. This is close to class invariants as in Eiffel. An aspect describes a family of concrete structures rather than one class implementation.

2.2.3 Methods

Methods model the behaviour of objects. Features are public methods of the behaviour. From the user point of view, features define useful and exportable functionalities of the class. But for the analyst or the designer of the class, the behaviour is split in two parts: **primitive methods** and **extensions**. We classify the methods as follows:

$$methods \begin{cases} primitive \begin{cases} observers \begin{cases} field\ selectors \\ semantic\ equality\ (equal?) \\ external\ description\ (describe) \end{cases} \\ \\ constructors \begin{cases} generator\ (new) \\ copy\ of\ object\ (copy) \end{cases} \end{cases} \\ \\ extensions \end{cases}$$

Instance methods and class methods are distinguished; `new<CFC>` is a class method.

Primitive methods Primitive methods are essential to describe and to handle class instances. They are implemented from the aspect into OOP languages. Other primitive methods are possible, we represent only the main in the above picture.

Field selectors are associated to the characteristics which describe objects. Semantic equality allows us to compare objects in an abstract way, independently of implementation. An object description is an external representation (a string) of the object.

Primitive methods `new` and `copy` have the usual semantics of OOP languages. Axioms of primitive methods are completely defined by the field selectors. They are left out in a formal class.

Extensions A programmer naturally defines methods as algorithms using instance variables, predefined entities and control structures. In the model an extension is a method whose semantics is defined relatively to primitive methods. An extension is a profile, a precondition and axioms. More precisely we use axioms like

$$\texttt{cond ==> oper(Self, ...) == u} \tag{2.2.3}$$

2.2.4 Relations

Structural dependency is a relation between formal classes. Structural dependency (`DPS`) is the set of links between a formal class and the resulting types of its field selectors. Class `IntermediateTank` depends on type `NONFULLTank` and type `Liquid`.

A formal class is a finitely generated algebraic type with the `new` generator. A well-structured class must contain at least one instance and there must be a well-found structural induction on these instances. A necessary condition for well-structuring is: $\forall\ (fsel_i\ :\ CFC \rightarrow T_i$ `requires:` $pred_i) \in class\ CFC$,

- if $pred_i \equiv true$ then T_i is well-structured and does not depend on CFC,
- else T_i is well-structured or $T_i\ dps\ CFC$.

In our model we use inheritance more rigorously than in concrete languages. This avoids some problems (inheritance of instance variables, confusing structural dependency with inheritance, type errors, and complexity of method lookup) and especially allows to define a minimal model for inheritance ([32] for details).

The inheritance graph must be without cycle. There is no inheritance of aspects. A class `S` inherits from `C` implies that every field selector of `C` exists in `S` with the same type or a subtype. The previous criterion is necessary but not sufficient. If they are constraints or field selector preconditions, the rule implies additional constraints and additional conditions in the subclass.

To avoid difficulties with method lookup we choose a general, simple and natural software engineering rule. Whenever there is a name clash, the designer must rename or, better redefine the method. Other strategies like depth-first, breadth-first or topological search are possible but they can add implementation problems [3].

2.2.5 Strong Typing

The model fits well to dynamically typed languages but also to strongly typed languages like Eiffel.

The type checking is based on usual principles:

- There are, may be, predefined types (which are not classes!).
- A class defines a type.
- Variables and methods are explicitly typed.
- Typing a message is like typing an operation application.

We add some rules specific to OOP :

- Inheritance implies subtyping. We do not handle functional objects this avoid the need of a contra-variant rule [11] which is not natural here.
- There is single dispatch of the method to apply on the receiver type.
- A public method is always public in its subclasses and a non abstract class has no abstract method in its complete behaviour.
- Methods are redefined according to covariant rules. These rules are consistent with the inheritance criterion.

In this context we define a type checking and prove the safety of the control.

2.2.6 Symbolic Evaluator and Proof

We define an abstract operational semantic to this model using conditional term rewriting [24]. This kind of semantics has practically two important benefits: it is user-friendly to object-oriented programmers and animation tools are easy to implement.

We transform each axiom of extensions into a conditional rewrite rule (left to right). The semantics of primitive methods is also defined through rewrite rules. A method is interpreted as a set of rules. A class is a set of method rules. So, the original program is viewed as a (inheritance) graph of set of set of rules.

A term in normal form is either a term built only on predefined types or a `new<CFC>` on normal form terms. The dynamic type (its class) of a term is given by its normal form. The basic strategy is evaluation by value as in concrete OOP languages. More precisely it first evaluates the receiver term, giving its normal form, and selects in the graph the method rules to apply.

The same scheme allows proofs in an algebraic style (equational deduction or term rewriting). Inductive proofs are also possible because we have an induction principle based on the structure of term in normal form (cf 2.2.4). With that scheme we proved in [33] a formal class design.

3 The model Transitions

In the previous description, we presented two model for software component description. The GAT model is an abstract but friendly way to document formally the components. The CF model is a much more constructive object-oriented approach. Object-oriented programming languages are the concrete representation level. Now we will overview the relations between these models.

3.1 From Graphic Abstract Data Types to Formal Classes

Formal classes refine graphic abstract data types by giving a particular, but still abstract, representation. Operations become methods and axioms are grouped in method definition.

Two main approaches are possible to transit from types to classes :

- the "flat" representation : a single class describes the full data type. If guards and the state-transition diagram are represented (boolean function, transition table, etc.) then the preconditions are computed immediately from the GAT preconditions. Otherwise, preconditions are expressed on the abstract structure itself. This requires a non trivial work from the designer. Now, suppose we have chosen an abstract structure. Since, observers may be partial, some field selectors may have no value. This absence of value can be expressed by an *undefined value* or $\perp$ (lifting), this is the usual approach in programming (e.g. `nil` for lists), or by conditional field selectors (e.g. selectors `liquid` and `tank` of class `FlatTank` of figure 12). The drawbacks of the flat approach are : an operational description, complex preconditions, and complex class description which lead bad expressiveness, readability and reuse.

- the "hierarchical" representation : one state is one class. There are no partial field selector to represent since each value of the class belongs to the algebra associated to the state. Preconditions are also simpler. Small parts are easier to understand and reuse. The main disadvantage is the combinatorial explosion of classes combined with the duplication of method definitions. Moreover it leads to a bad object representation since GAT transitions correspond to dynamic inheritance.

Looking at both approaches, several concrete representations with classes are possible to implement a given GAT specification. The aim of formal methods is to help the designer in his quest and not only to check that a class really implements an abstract data type as usually. In both cases, a method is needed to extract an abstract structure from a set of operations.

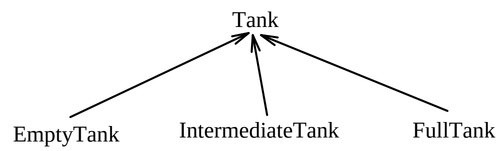

Figure 6 : Initial Specialization Hierarchy of the *Tank* Formal Classes

Each refinement case is possible. Experimentation shows that it is better to start from the second approach and to group commonalities and even classes in abstract nodes to increase module consistency and reduce their coupling.

We studied a method to get formal classes from GATs in [5]. The mains steps are:

- preliminary work : finding the primitive part of the GAT specification and group highly-coupled states in brother states (see section 2.1.2).
- extracting an abstraction hierarchy from operations profile and states. Operation profiles are split in terms of state and then join by merging states into superstates. These superstates become the abstract nodes of the hierarchies (one along the receiver and another along the result when of type `TI`). These states are also types. The USE hierarchy indicates which operation are available in each type. The CREATION hierarchy indicates how one creates values of that type.

$$\left.\begin{array}{l} fillWith : EmptyTank \rightarrow IntermediateTank \\ fillWith : EmptyTank \rightarrow FullTank \\ fillWith : IntermediateTank \rightarrow IntermediateTank \\ fillWith : IntermediateTank \rightarrow FullTank \end{array}\right\} NONFULLTank \rightarrow NONEMPTYTank$$

$$\left.\begin{array}{l} firstQuantity : IntermediateTank \rightarrow Liquid \\ firstQuantity : FullTank \rightarrow Liquid \end{array}\right\} NONEMPTYTank \rightarrow Liquid$$

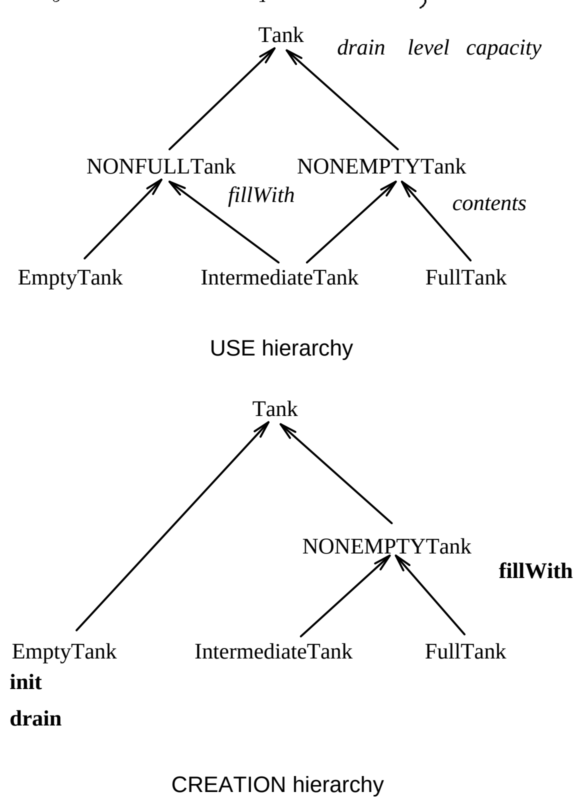

Figure 7 : Initial Specialization Hierarchy of the *Tank* Formal Classes

- now each node of the use hierarchy is supposed to be a formal class. We choose an abstract structure for each terminal class (with no subclass) from the generators that create values of that type-class. A first attempt is given by the union of parameters of the generators. In our simple example, there is only one generator for each state.

- restructuring is a refinement technique through abstraction functions that consists in finding a more appropriate structure and prove the isomorphism of one structure to the other. It occurs when simplifying a complex class or when grouping several subclasses in a unique superclass.

- the next step is writing extensions in respect with the abstract structure. This is a first checking of the validity of the abstract structure. Common methods are shared in common superclasses.

- at last, optimizations and hierarchy restructuring provide an inheritance hierarchy. The constraints can be strengthened. For example, in class `IntermediateTank`, the constraint `level(Self) <= capacity(Self)` becomes `level(Self) < capacity(Self)`. Hierarchy restructuring is gathering subclasses in a common superclass. A first structure is computed, where a boolean field selector represents each subclass. It contains all the field selectors of the subclasses conditioned by the corresponding boolean field selector. This non-optimized class will be restructured by another, more convenient abstract structure (see the above item on class restructuring). In our example, because of multiple inheritance, the only way to reorganize the hierarchy is to group all the subclasses in a unique class (similar to a flat design of appendix C).

Comparing with traditional refinement methods, our technique is interesting because it provides a fine-grain refinement and a step by step approach from types to classes with care to partial definitions and partitions of definition domains.

In our example, we get the inheritance and structural dependency graph of figure 9.

With these informations, we turn to the implementation a prototype.

3.2 From Formal Classes to OOP Languages

Formal classes are simple to implement in languages like CLOS, Smalltalk, Eiffel or C++. We implemented an automatic translator to Eiffel [2] and Smalltalk. Such translation produces the skeleton of the class (class structure, primitive methods and extension signatures). If the axioms are written in a functional style, a direct translation exists from axioms to method body. Otherwise, the designer writes the body of the methods according to the axioms (simple in general). An example of Eiffel class is given in appendix D.

When dealing with multiple inheritance in single inheritance object-oriented programming languages (Smalltalk), methods have to be copied. The closest common abstract class will be the superclass.

Experiments showed that classes are very small. But due to the functional background, the prototype was definitely not efficient. Optimizations are to be considered, either in the formal class modeling or in programs. The main techniques are:

- introducing side effects. Note that the FC model has been extended to side effects. The side effect model adds sequencing (;) and the primitive method `modify!`. This extension is used to optimize a first functional design. However we loose the simplicity of proof mechanism.

- modifying the representation by implementing an extension by an instance variable, implementing a field selector by an instance method.

- sharing some code of similar methods in the superclass and using method composition and polymorphism. The comparison of methods is more simple than in programming languages because of the axioms.

- refining the abstract structure into more concrete data structures using existing classes of the programming environment libraries (for example : use the class `List[Item]` in the flat representation of appendix C).

- grouping several classes in a common superclass to limit dynamic inheritance and enforce object identity.

Other optimizations are based on the special features of each language: dynamic or static typing, class libraries, private definitions, instance variable inheritance, single inheritance... Full description of implementation techniques are out of the scope of this paper.

4 Method

Formal methods can be used at several stages of the software life cycle. We distinguish two main applications :

- formal specification of the requirements that lead to rapid prototyping,

- formal description of design to prepare component documentation and incremental implementation.

The GAT formalism can be used to abstractly describe the objects of the analysis step. But other formalisms like OMT [34] or OOA of Shlaer and Mellor [35] are more friendly notations for practitioners. Unlike some attempt to derive formal specification from OMT specifications (as in [26]) we do not focus on the object model but the dynamic model because the object model overspecifies the relations between software components. Under some restrictions, especially for state aggregation and internal parallelism, we translated OMT dynamic behaviour into GATs. But one difficulty is the systematic use of asynchronous communications in OMT, conflicting with the implicit functional background of GATs.

Formal classes can be used independently from GATs for formal design. The abstract descriptions allow theorem proof and animation before implementation.

The GAT/FC models can also be use as a whole from analysis to implementation. A specification cycle is decribed by the following GAT:

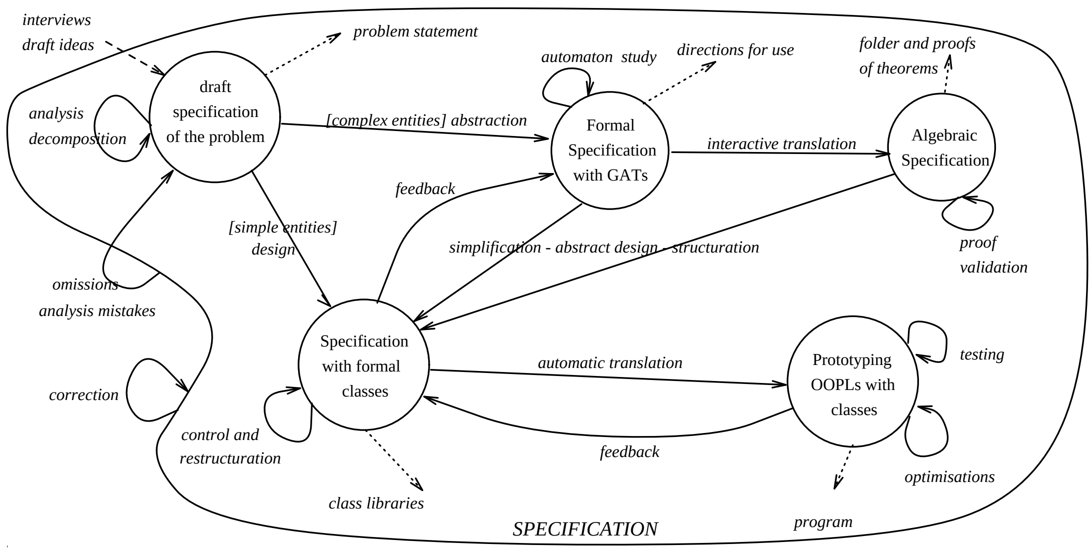

Figure 8 : Specification Process

A complete and validated development method is still missing.

5 The Environment

We developed an environment to support these concepts and methods. A part of the tools has been implemented in ParcPlace Smalltalk, release 4.

5.1 Tools for Organizing Specifications

A graphic interface has been developed to represent visually the links between GATs (subtyping, use), FCs (inheritance, structural dependency) (see figure 9), concrete classes (inheritance, aggregation, ...) but also between a GAT and its FCs or a formal class and its different implementations. This is a visual snapshot of the complexity of the system, through the component interaction.

Browsing through the graph make dynamic specification possible. Graphical views on the system are also useful to see the cycles in the definitions (before type and inheritance checking), to reorganize the inheritance or the clientship hierarchy (avoid repeated inheritance or exceptions).

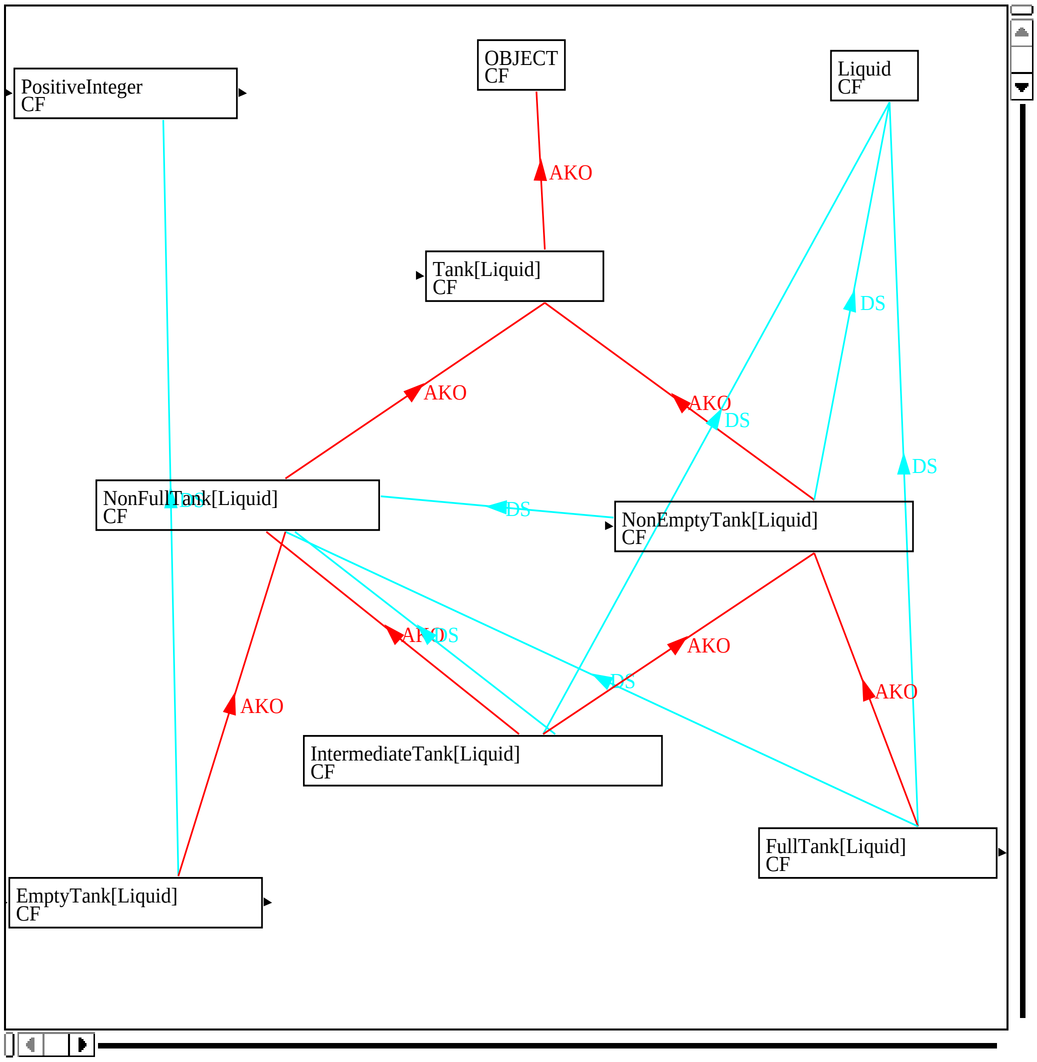

Figure 9 : DPS and AKO Hierarchies of the *Tank* Formal Classes

For complex applications, a larger grain of definition is necessary. This corresponds to the widespread idea of module. A module is a group of class working for a same goal. For example, a car-washing system is composed of tanks for water and cleaning products, washing brushes, a ventilator. Complex relations can rely modules (clientship, aggregation, inheritance). Integrate (application) modules and classes is under study.

5.2 Tools for Writing Specifications

Formal languages are implemented to manipulate GATs and FCs. Editing tools and library managers are developed with graphical interface. Graphic and textual editors are developed on the grammars to guide, capture and control specifications (e.g. figure 10 and figure 11).

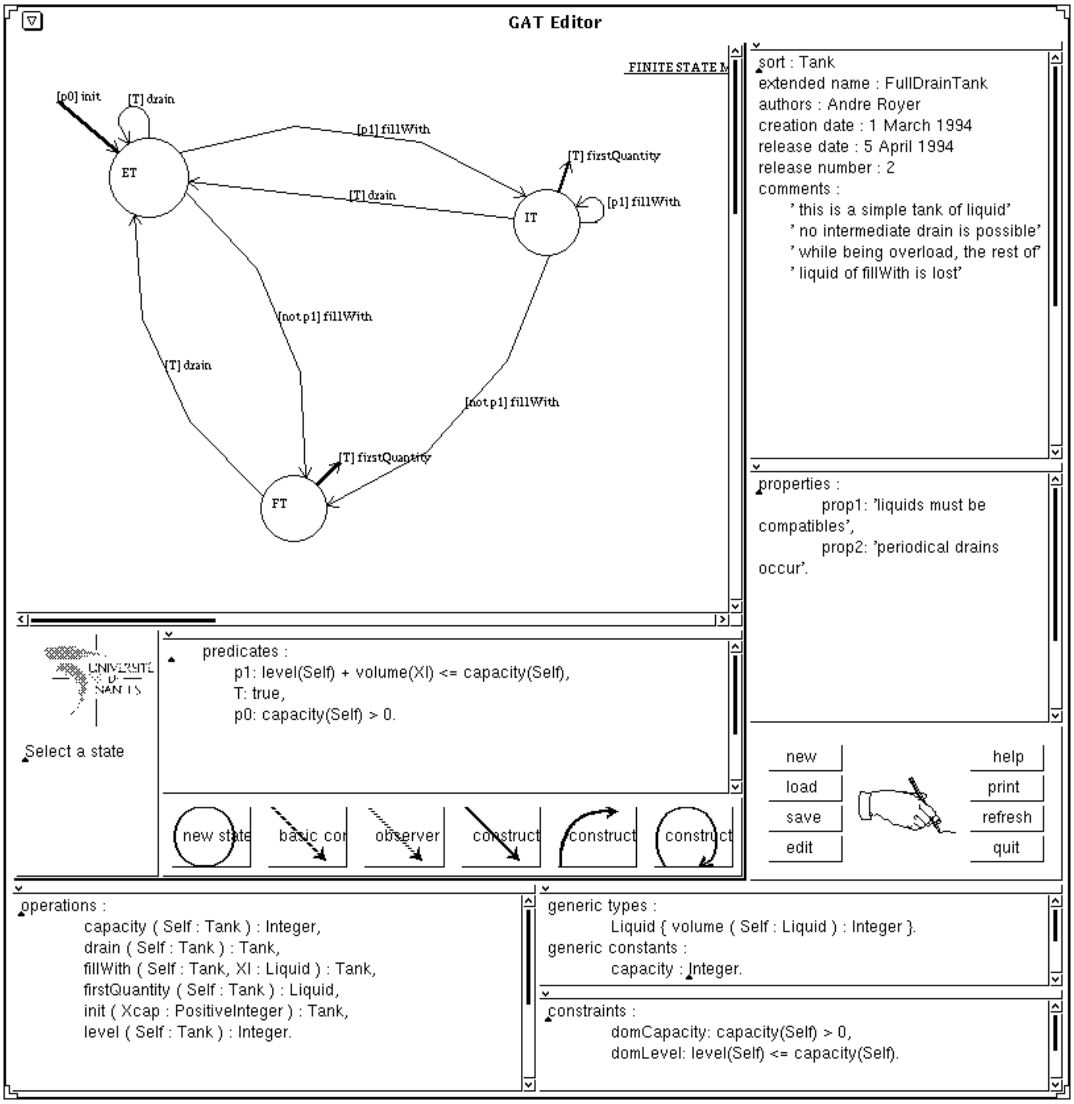

Figure 10 : The GAT Graphic Editor

Consistency checking is done inside the data types and between the data types.

- syntactic controls (double definition of operations, parameters names, etc.),
- cross checking between the operations signature and the finite state machine,
- inheritance rules verification,
- existence of imported types,
- no circular dependencies in the inheritance graph nor in the clientship graph (according to the defined rules),
- type checking.

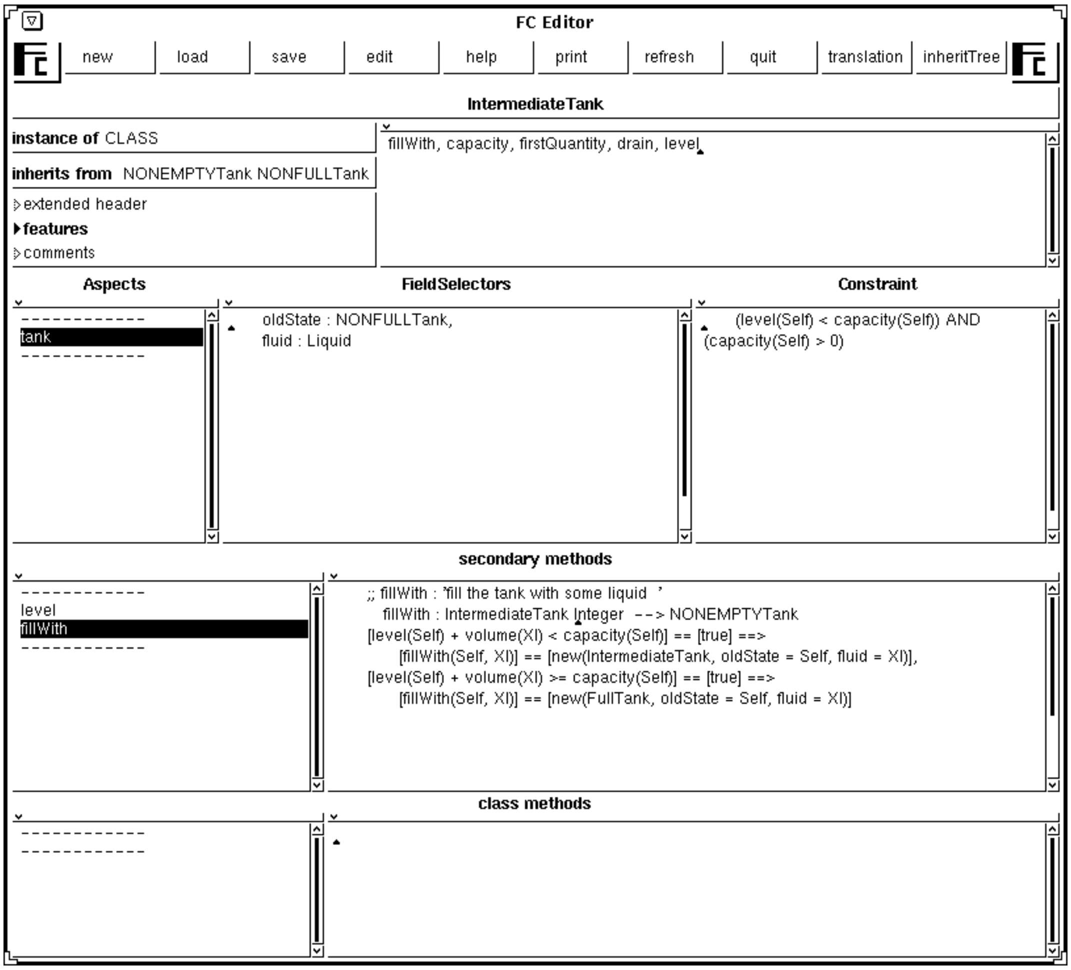

Figure 11 : The FC Graphic Editor

As for GATs, some control deal with formal classes

- signature and exported interface,
- inheritance rules verification,
- existence of imported types,
- consistency between importation and exportation declarations,
- no circular dependencies either in the inheritance graph or in the structural dependency graph,
- type checking.

A set of various tools is implemented like LaTeX and postscript formatting.

5.3 Tools for Testing Specifications

One of the interest of formal specifications is to set formal properties on the system. Interface with automata provers like AUTO [28] may be an answer to dynamic properties checking. Interface with logic prover like Muffin (Mural environment [23]) or the B Tool [25] are solid environments for logic theorems proof. We rather use animation for testing specifications.

5.4 Tools for Refining Specifications

A translation scheme has been approximate to extract GATs from OMT specifications. Automata structuring means are lacking to consider a direct translation. The OMT dynamic behavior model is actually more powerful. A deeper conversion mecanism needs formal grammars.

We described an algorithm to interactively translate a GAT specification into a pure algebraic specification in section 2.1.2. The current algebraic specification is actually a meta-language. A second translator, based on our algebraic specification meta-language provides specifications for the target algebraic environment (LARCH, Asspegique, OBJ).

The refinement process applied in section 3.1 needs human interaction to take design decisions. Melting the automatic parts and interactive parts is on progress.

As mentioned in section 3.2, we implemented algorithms to translate formal classes into Smalltalk and Eiffel, through compilers. Full interactive support is required to derive code from axioms.

5.5 Tools for Animation and Prototyping

At each level of the specification, tools are needed to animate the specification. For the GAT level, we mainly use the Asspegique or REVE environments. By orientation of axioms, we transform the equational specification into a rewrite system that can be executed. Actually, we use these environments as tools for proving parts of the specifications rather than overall consistency and completeness checking.

It would be interesting to animate automata during execution of the prototype. An abstract machine must be defined to support that user attractive animation. This tools is linked with the realization of a symbolic evaluator for GAT, which has not been studied yet.

A symbolic evaluator exists for formal classes (see section 2.2.6) using term rewriting [24].

5.6 Tools for Reusing Specifications

Reuse is not only an implementation issue. Having several representation level improves data type comprehension and then reuse. Abstract description of software components, through axioms make class or data type comparison possible. Tools are needed to extract similarity from a component to be specified and existing components.

6 Concluding Remarks and Future Work

In this abstract paper, we overviewed our general formal specification scheme. The three layered model adresses the problems of writing and refining formal specifications. It supports a smooth and clear transition from component specification to programming. The object orientation improve component specification reuse and modular proof obligation.

Other object-oriented formal specification languages exist. Some are based on Z (Object-Z [14], Z++ [27], MooZ [30]) or VDM (VDM++ [15]). They use the Z toolkit and environments to fulfill the formal requirements. The Z presentation is pleasant, but library and system manager are lacking. Some of these languages contains concurrent features (traces, temporal logic). OOZE [1] is a Z-style language based on the algebraic language OBJ and takes advantage from the rich specification environment. OS/OP [10] is a object-oriented specification language descended from the theory of partial algebra. Inheritance (refinement) is distinguished from subtyping. It seems a good theoretical frame, but no environment exists until now.

Implementing a full environment for formal development is surely a long and exacting task. We rather promote an opened environment to support the concepts and methods presented in this paper. Gates are needed to interface with other, more standard or advanced, specification environments, especially for proving the specifications. Customized tools deal with component management, edition and animation.

Editing and control tools have been implemented together with translators to programming languages. Current work is about the implementation of the gates, and the method to refine GATs into formal classes. We defined typing and inheritance rule for a kernel model of the graphic abstract types and the formal class models. A type checking algorithm has been defined for formal classes.

Future work is about modules (horizontal composition), and a concept of schema that groups together GAT and FC components (vertical refinement). We must extend the type checking rules to GAT and FC. Concurrency between finite state machines has been studied, through synchronization product [7]. But internal parallelism is not yet allowed in the model. Concurrency relates with system design and object asynchronous communication.

References

[1] A. J. Alencar and J. A. Goguen. OOZE: An Object-Oriented Z Environment. In P. America, editor, *Proceedings of ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199, Geneva, Switzerland, 1991. Springer Verlag.

[2] P. André, D. Chiorean, and J.-C. Royer. The Formal Class Model. In *Joint Modular Languages Conference*, pages 59–78, Ulm, Germany, 28-30 september 1994. GI, SIG and BCS.

[3] P. André and J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In A. Paepcke, editor, *OOPSLA'92 Proceedings*, pages 110–126, Vancouver, October 1992. ACM, ACM Press.

[4] P. André and J.-C. Royer. Aide à la spécification algébrique de types abstraits par l'utilisation d'automates : principes, algorithmes et exemples. Rapport ERTO, Université de Nantes, May 1994.

[5] P. André and J.-C. Royer. Building Formal Classes from Graphic Abstract Data Types. *Rapport ERTO*, 1994.

[6] P. André and J.-C. Royer. La modélisation des listes en programmation par objets. In Cointe, Pierre, Queinnec, Christian et Serpette, Bernard (éditeurs), editor, *Journées Francophones des Langages Applicatifs (JFLA'94)*, number 11 in Collection Didactique, pages 259–285, Noirmoutier, 31 janvier - 1 février 1994. INRIA.

[7] A. Arnold. *Systèmes de transitions finis et sémantique des processus communiquants*. Collection Etudes et recherches en informatique. Masson, 1992.

[8] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. The BCS Practitioner. Prentice Hall, 1991.

[9] M. Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l'informatique*, pages 347–357, Paris, Mar. 1982.

[10] R. Breu. *Algebraic Specification Techniques in Object-Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science*. Springer Verlag, June 1991. Thesis, Universität Passau, Germany.

[11] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.

[12] E. Casais, C. Lewerentz, T. Lindner, and W. Franz. Formal Methods and Object-Orientation. In B. Magnusson, B. Meyer, and J.-F. Perrot, editors, *TOOLS EUROPE '93*, Versailles, Mar. 1993. International Conference & Exibition, Prentice Hall. Tutorial T7.

[13] C. Choppy, D. Bert, M. Bidoit, R. Echahed, J. Reynaux, C. Roques, and F. Voisin. Rapid Prototyping with Algebraic Specifications: a Case Study. Research 844, LRI - Orsay, Université Paris Sud, May 1993.

[14] D. Duke and R. Duke. Towards a Semantics for Object-Z. In D. Bjorner, C. Hoare, and H. Langsmaack, editors, *Proceedings of VDM'90*, pages 244–261. Springer Verlag, 1990.

[15] E. Dürr and N. Plat. VDM++ Language Reference Manual. Technical report afro/cg/ed/lrm/v11, Cap Volmac - Utrecht University, Aug. 1995. Esprit III AFRODITE project.

[16] H. Ehrig and I. Classen. Overview of Algebraic Specification Languages, Environment and Tools, and Algebraic Specifications of Software Systems. *EATCS Bulletin*, 39, 40, 41, 1990. Part 1 in number 39, Part 2 in number 40, Part 3 in number 41.

[17] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1 : Equations and Initial Semantics*, volume 6 of *EATCS Monograph on Theorical Computer Science*. Springer-verlag, 1985.

[18] L. M. G. Feijs and F. B. M. Jonkers. *Formal Specification and Design*, volume 35 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[19] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Jan. 1985. SIGACT, SIGPLAN.

[20] M.-C. Gaudel. *Algebraic Specifications*, chapter 22. Software Engineers Reference Book. Butterworths, Mc Dermid, J. (Ed.), Mar. 1990.

[21] J. V. Guttag and J. J. Horning. The algebraic Specification of Abstract Data Types. *Acta informatica*, 10:27–52, 1978.

[22] D. Harel, H. Lachover, A. Naomad, and A. Pnuelli. Statemate: a working environment for the development of complex reactive systems. *IEEE transactions on software engineering*, 16(4), Apr. 1990.

[23] C. B. Jones. Does the OO-Community Need Formal Methods. In B. Meyer and J. Bezivin, editors, *TOOLS'91*, pages 15–18, Paris, Mar. 1991. Invited Lectures.

[24] J.-P. Jouannaud and P. Lescanne. La réécriture. *Technique et Science Informatique*, 5(6):433–452, 1986.

[25] C. Lafontaine, Y. Ledru, and P.-Y. Schobbens. An Experiment in Formal Software Development : using the B Theorem Prover on a VDM Case Study. *Communications of the ACM*, 34(5):62–71, May 1991.

[26] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. Object Oriented Series. Prentice Hall, 1993.

[27] K. Lano and H. Haughton. The Z++ Manual. Technical report, Imperial College, London, Oct. 1994.

[28] V. Lecompte, E. Madelaine, and D. Vergamini. AUTO : un système de vérification de processus parallèles et communicants. Technical Report 83, INRIA, Mar. 1987. french.

[29] B. Liskov and J. Guttag. *La Maîtrise du Développement de Logiciel : abstraction et spécification*. Collection "Ingénierie des systèmes d'information". Les Editions d'Organisation, 1990.

[30] R. L. Meira, Silvio and A. L. C. Cavalcanti. The MooZ Specification Language. Technical Report ES/1.92, Universidade Federal de Pernambuco, Janeiro 1992. version 0.4.

[31] D. R. Musser. On proving Inductive Properties of Abstract Data Types. In *Proceedings of the Seventh Annual ACM Symposium on PoPL*, pages 154–162, Las Vegas, Jan. 1980.

[32] J.-C. Royer. A New Set Interpretation for the Inheritance Relation and its Checking. *ACM OOPS MESSENGER*, 3(3):22–40, July 1992. Rapport N^o 91-10 LIST, Université de Nantes.

[33] J.-C. Royer. Un exercice de spécification formelle de preuve et de conception à objets. Rapport de recherche 30, IRIN, Faculté des Sciences et des Techniques, Université de Nantes, Oct. 1993.

[34] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall International, 1991.

[35] S. Shlaer and S. J. Mellor. *Object Lifecycles*. Yourdon Press Computing. Prentice Hall, 1992. Modeling the World in States.

[36] I. Sommerville. *Le génie logiciel*. Addison-Wesley France, 4e edition, Oct. 1992.

7 Appendix A: Algebraic Specification of Data Type `Tank`

- Signature of the `Tank` ADT:

$$\left.\begin{array}{ll} init & : Integer \rightarrow Tank \\ fillWith & : Tank\ Liquid\ \rightarrow\ Tank \\ drain & : Tank\ \rightarrow\ Tank \end{array}\right\} constructors$$

$$\left.\begin{array}{ll} level & : Tank\ \rightarrow\ Integer \\ capacity & : Tank\ \rightarrow\ Integer \\ firstQuantity & : Tank\ \rightarrow\ Liquid \end{array}\right\} observers$$

$$\left.\begin{array}{ll} ET & : Tank\ \rightarrow\ Boolean \\ IT & : Tank\ \rightarrow\ Boolean \\ FT & : Tank\ \rightarrow\ Boolean \end{array}\right\} state\ observers$$

- General predicates :

```
% invariant
invariant(Self) == level(Self) <= capacity(Self) AND capacity(Self) > 0
% guards
    p0(Xcap) == Xcap > 0
    p1(Self, Xl) == volume(Xl) < capacity(Self) - level(Self)
```

- Preconditions of operations

```
precondition(init(Xcap)) == p0(Xcap)
precondition(fillWith(Self, Xl)) == ET(Self) OR IT(Self)
precondition(drain(Self)) == true
precondition(firstQuantity(Self)) == IT(Self) OR FT(Self)
precondition(capacity(Self)) == true
precondition(level(Self, Xw)) == true
```

- State observers on generators

```
ET1:  ET(init(Xcap)) == true
ET2:  ET(fillWith(Self,Xl)) == false
IT1:  IT(init(Xcap)) == false
IT2:  IT(fillWith(Self,Xl)) == [p1(Self, Xl) AND ET(Self)] OR [p1(Self, Xl) AND IT(Self)]
FT1:  FT(init(Xcap)) == false
FT2:  FT(fillWith(Self,Xl)) == [NOT p1(Self, Xl) AND ET(Self)] OR [NOT p1(Self, Xl) AND IT(Self)]
```

- primitive observers on generators

```
capacity1- capacity(init(Xcap)) == Xcap
capacity2- capacity(fillWith(Self,Xl)) == capacity(Self)

level1:  ET(Self) == true ==> level(Self) == 0
level2:  ET(Self) == true & p1(Self, Xl) == true ==> level(fillWith(Self,Xl)) == volume(Xl)
level3:  IT(Self) == true & p1(Self, Xl) == true ==> level(fillWith(Self,Xl)) ==
         volume(Xl) + level(Self)
level4:  FT(Self) == true ==> level(Self) == capacity(Self)
% G-derivation on state IT
firstQuantity1:  ET(Self) == true & p1(Self, Xl) == true ==> firstQuantity(fillWith(Self,Xl)) == Xl
firstQuantity2:  IT(Self) == true & p1(Self, Xl) == true ==> firstQuantity(fillWith(Self,Xl)) == firstQuantity(Self)
% G-derivation on state FT
firstQuantity3:  ET(Self) == true & NOT p1(Self, Xl) == true ==>
         firstQuantity(fillWith(Self,Xl)) == Xl
firstQuantity4:  IT(Self) == true & NOT p1(Self, Xl) == true ==>
         firstQuantity(fillWith(Self,Xl)) == firstQuantity(Self)
```

- secondary constructors on generators

```
drain1:  drain(Self) == init(capacity(Self))
```

8 Appendix B: Formal Classes for `Tank`

The classes below are those of the hierarchy of figure 7. The constraints are adapted in each class.

Tank	
ABSTRACT	
inherits from OBJECT	
comments:	this is a simple tank of liquid no intermediate drain is possible the first tank overload is lost
features: capacity, level, drain	
aspect : tank	
field selectors	**constraint**
	level(Self) ¡= capacity(Self) AND capacity(Self) ¿ 0
extensions	
;; capacity : capacity of the tank capacity : Tank ⟶ Integer ABSTRACT	
;; level : current level of the tank level : Tank ⟶ Integer ABSTRACT	
;; drain : drain the tank drain : Tank ⟶ EmptyTank drain(Self) == newEmptyTank(capacity= capacity(Self))	

NONEMPTYTank		
ABSTRACT		
inherits from Tank		
comments: the tank is not empty		
features: capacity, firstQuantity		
aspect : tank		
field selectors		**constraint**
oldState	: NONEMPTYTank ⟶ NONFULLTank	capacity(Self) > 0 AND
fluid	: NONEMPTYTank ⟶ Liquid	level(Self) <= capacity(Self)
extensions		
;; capacity : answer the capacity of the tank capacity : NONEMPTYTank ⟶ Integer capacity(Self) == capacity(oldState(Self))		
;; firstQuantity : answer the type of liquid of the tank firstQuantity : NONEMPTYTank ⟶ Liquid firstQuantity(Self) == fluidAux(oldState(Self),fluid(Self))		

NONFULLTank

ABSTRACT

inherits from Tank

comments: the tank is not full

features: fillWith

aspect : tank

field selectors	**constraint**
	capacity(Self) ¿ 0 AND level(Self) ¡= capacity(Self)

extensions

;; fillWith : fill the tank with some liquid
  fillWith : NONFULLTank Liquid ⟶ NONEMPTYTank
var: Xl : Liquid
level(Self) + volume(Xl) < capacity(Self) == true ==> fillWith(Self, Xl) ==
    newIntermediateTank(oldState= Self, fluid= Xl)
level(Self) + volume(Xl) >= capacity(Self) == true ==> fillWith(Self, Xl) ==
    newFullTank(oldState= Self, fluid = Xl)

;; fluidAux : auxillary method for firstQuantity
  fluidAux : NONFULLTank Liquid ⟶ Liquid
var: Xl : Liquid
ABSTRACT

EmptyTank

inherits from NONFULLTank

comments: the tank is empty

features: level

aspect : tank

field selectors	**constraint**
capacity : EmptyTank ⟶ Integer	capacity(Self) > 0 AND
	level(Self) ¡= capacity(Self)

extensions

;; level : provides the level of liquid of the tank
  level : EmptyTank ⟶ Integer
level(Self) == 0

;; fluidAux : auxillary method for firstQuantity
  fluidAux : EmptyTank Liquid ⟶ Liquid
var: Xl : Liquid
fluidAux(Self, Xl) == Xl

class methods

;; init : creates a new tank with capacity cap
  init : EmptyTank Integer ⟶ EmptyTank
var: Xcap : Integer;
requires: Xcap > 0
init(Xcap) == new(EmptyTank, capacity = Xcap)

IntermediateTank	
inherits from NONEMPTYTank NONFULLTank	
comments: An intermediate tank is neither empty nor full	
features: capacity, level	
from: Integer **import:** +, 0, <	
from: Liquid **import:** volume	
aspect : tank	
field selectors	**constraint**
oldState : IntermediateTank ⟶ NONFULLTank	0 < capacity(Self) AND
fluid : IntermediateTank ⟶ Liquid	level(Self) < capacity(Self)
extensions	
;; capacity : maximum capacity capacity : IntermediateTank ⟶ Integer capacity(Self) == capacity(oldState(Self))	
;; level : current level level : IntermediateTank ⟶ Integer level(Self) == level(oldState(Self)) + volume(fluid(Self))	
;; fluidAux : auxillary method for firstQuantity fluidAux : IntermediateTank Liquid ⟶ Liquid **var:** X1 : Liquid fluidAux(Self, X1) == fluidAux(oldState(Self),fluid(Self))	

FullTank	
inherits from NONEMPTYTank	
comments: the tank is full	
features: level	
aspect : tank	
field selectors	**constraint**
oldState : FullTank ⟶ NONFULLTank	capacity(Self) > 0 AND
fluid : FullTank ⟶ Liquid	level(Self) = capacity(Self)
extensions	
;; level : provides the level of liquid of the tank level : FullTank ⟶ Integer level(Self) == capacity(Self)	

9 Appendix C: Flat Formal Class for Tank

A recursive design for a single formal class **Tank** is :

FlatTank	
inherits from OBJECT	
comments: a flat implementation of the GAT Tank	
features: fillWith, capacity, firstQuantity, drain, level	
aspect : tank	
field selectors	**constraint**
capacity : FlatTank ⟶ Integer	capacity(Self) > 0 AND level(Self) <= capacity(Self)
isEmpty? : FlatTank ⟶ Boolean	
liquid : FlatTank ⟶ Liquid **requires:** isEmpty?(Self) == false	
tank : FlatTank ⟶ FlatTank **requires:** isEmpty?(Self) == false	

secondary methods

;; isFull? : answer whether the tank is full or not
isFull? : FlatTank ⟶ Boolean
isEmpty?(Self) == false ==> isFull?(Self) == volume(liquid) + level(tank) >= capacity(Self)
isEmpty?(Self) == true ==> isFull?(Self) == false

;; level : provides the level of liquid of the tank
level : FlatTank ⟶ Integer
isEmpty?(Self) == true ==> level(Self) == 0
isFull?(Self) == true ==> level(Self) == capacity(Self)
isEmpty?(Self) == false ^ isFull?(Self) == false ==>
level(Self) == volume(liquid) + level(tank)

;; firstQuantity : answer the firstQuantity of the tank
firstQuantity : FlatTank ⟶ Liquid
requires: NOT isEmpty(Self)
isEmpty?(tank(Self)) == true ==> firstQuantity(Self) == liquid(Self)
isEmpty?(tank(Self)) == false ==> firstQuantity(Self) == firstQuantity(tank(Self))

;; fillWith : fill the tank with some liquid
fillWith : FlatTank Integer ⟶ FlatTank
var: Xl : Liquid;
requires: NOT isFull?(Self)
fillWith(Self, Xl) == newFlatTank(capacity = capacity(Self),
isEmpty? = false, liquid = Xl, tank = Self)

;; drain : empties the tank
drain : FlatTank ⟶ FlatTank
drain(Self) == newFlatTank(capacity = capacity(Self), isEmpty? = true)

class methods

;; init : creates a new tank with capacity cap
init : Integer ⟶ FlatTank
var: Xcap : Integer;
requires: Xcap > 0
init(Xcap) == newFlatTank(capacity = Xcap, isEmpty? = true)

Figure 12 : The **FlatTank** Formal Class

A flat design, obtained from the merging of the subclasses into one single and non-abstract class **FlatTankList** is as follow.

We assume the existence of a generic formal class specifying the list. Such a modelling is given in [6]. We add an hidden method to iterate on the list **mapVolume**.

`FlatTankList`

instance of `CLASS`

inherits from `OBJECT`

comments: a flat implementation of the GAT Tank

features: `fillWith, capacity, firstQuantity, drain, level`

aspect : tank

field selectors	**constraint**
`capacity : FlatTankList ⟶ Integer`	`capacity(Self) > 0 AND`
`contents : FlatTankList ⟶ List`	`level(Self) <= capacity(Self)`

secondary methods

;; `isEmpty?` : answer whether the tank is empty or not
`isEmpty? : FlatTankList ⟶ Boolean`
`isEmpty?(Self) == isEmpty?(contents(Self))`

;; `mapVolume` : apply an operation volume on every item of the list
`mapVolume : List ⟶ Integer`
var: `ll : List; li : Liquid;`
`isEmpty?(ll) == true ==> mapVolume(ll) == 0`
`isEmpty?(ll) == false ==> mapVolume(ll) == volume(head(ll)) + mapVolume(tail(ll))`

;; `isFull?` : answer whether the tank is full or not
`isFull? : FlatTankList ⟶ Boolean`
`isFull?(Self) == mapVolume(contents(Self)) > capacity(Self)`

;; `level` : provides the level of contents of the tank
`level : FlatTankList ⟶ Integer`
`isFull?(Self) == true ==> level(Self) == mapVolume(contents(Self))`
`isFull?(Self) == false ==> level(Self) == capacity(Self)`

;; `firstQuantity` : answer the firstQuantity of the tank
`firstQuantity : FlatTankList ⟶ Liquid`
requires: `NOT isEmpty(Self)`
`firstQuantity(Self) == last(contents(Self))`

;; `fillWith` : fill the tank with some contents
`fillWith : FlatTankList Integer ⟶ FlatTank`
var: `Xl : Liquid;`
requires: `NOT isFull?(Self)`
`fillWith(Self, Xl) == copy(Self, contents= cons(contents(Self),Xl))`

;; `drain` : empties the tank
`drain : FlatTankList ⟶ FlatTank`
`drain(Self) == copy(Self, contents= newEmptyList)`

class methods

;; `init` : creates a new tank with capacity cap
`init : Integer ⟶ FlatTank`
var: `Xcap : Integer;`
requires: `Xcap > 0`
`init(Xcap) == newFlatTank(capacity= Xcap, contents= newEmptyList)`

Figure 13 : The `FlatTank` Formal Class

Note that it could be a intuitive design for the tank, which is less reusable than the hierarchic one of appendix B.

10 Appendix D: Eiffel Translation for Class `IntermediateTank`

```
-- direct Eiffel translation V 2.3
   CLASS IntermediateTank

   EXPORT  capacity, level, fluid, oldState, fillWith, firstQuantity, drain

   INHERIT NONFULLTank
           DEFINE capacity, level, fluid, oldState;
           NONEMPTYTank
           DEFINE capacity, level, fluid, oldState;

   FEATURE

           fluid_private : Liquid;
           oldState_private : NONFULLTank;

           -- fields selectors
           fluid : Liquid IS
                   DO
                   RESULT := fluid_private;
                   END; -- fluid

           oldState : NONFULLTank IS
                   DO
                   RESULT := oldState_private;
                   END; -- oldState

           -- Create
           Create (XoldState : NONFULLTank; Xfluid : Liquid) IS
                   DO
                   fluid_private := Xfluid;
                   oldState_private := XoldState;
                   END; -- create

           -- capacity
           capacity : Integer IS
                   DO
                   RESULT := oldState.capacity
                   END; -- capacity

           -- level
           level : Integer IS
                   DO
                   RESULT := oldState.level + fluid.volume;
                   END; -- level

           -- fluidAux
           fluidAux (X1 : Liquid) : Liquid IS
                   DO
                      RESULT := oldState.fluidAux(fluid);
                   END; -- fluidAux

   -- invariant
   INVARIANT level < capacity;

   END --class IntermediateTank
```