Introduction to Object-Oriented Mixed Prototyping with Algebraic Specifications

Jarle Hulaas

Swiss Federal Institute of Technology, Lausanne
Department of Computer Science
CH-1015 Lausanne
Switzerland

E-mail: hulaas@di.epfl.ch

Abstract

Automatic implementations of abstract data types (ADT) from algebraic specifications are currently all done in either functional, logical or purely imperative languages. In this paper we show how object-oriented languages naturally facilitate this job, and why they provide better flexibility in the process of progressively replacing the automatically generated code by more efficient hand-written code, a process we call *object-oriented mixed prototyping* (OOMP).

Keywords: Executable specifications, Mixed prototyping, Algebraic specifications, Abstract Data Types, Object-Oriented languages, Polymorphism.

1. Introduction

Formal specifications are taking an ever increasing importance in industrial applications. The advantage of formal methods resides by definition in their unambiguous nature. They can thus be supported by tools which automatically verify the properties, extract useful information or transform them into other representations. Moreover, they require a thorough analysis of the modelled system during the very first levels of the development process.

The benefits of formal specifications become even more evident when they can be used continuously through several stages of the software development cycle, like specification, implementation, integration or testing. Our purpose is to fill the gap that currently exists between the phase where a specification is simulated, generally using some logical inference tool, and the phase where the modelled system has an *efficient and maintainable* implementation in a main-stream object-oriented programming language. This objective is realized by application of a methodology called *mixed prototyping* [Ch 87].

Our work is done within the frame of the CO-OPN$_{1.5}$ (Concurrent Object-Oriented Petri Nets) specification language [BG 91][BBB+ 95] and the associated development tool called SANDS (Structured Algebraic Nets Development System) [BFR 93]. CO-OPN is directed towards specification of complex distributed systems. The specification language is based on structured algebraic Petri nets [Rei 91], where the Petri nets are used to model operational and concurrent aspects, and the algebraic specifications [EM 85] serve as a means of abstractly describing the data types involved. In this paper, we consider only the latter part of CO-OPN, and the results are thus directly applicable to all algebraic specification languages descended from Ehrig and Mahr's ACT-ONE [EM 85], such as PLUSS [Gau 85] and the data part of LOTOS

[ISO 88]. The basic principles can probably be generalized to any development method involving formal specifications: Our future work is to apply the methodology to concurrent systems.

In the following section we will first recall the basics of algebraic specifications, along with the semantics of the source and the target languages, as well as the compilation algorithm which translates the former language into the latter. Then we explain the principles of our methodology and its advantages.

2. Theoretical Framework

Most tools aimed at evaluation of ADTs are in fact theorem provers which can be, in the case of positive conditional axioms, evaluated using a Prolog interpreter. This is necessary because the expressive power of equational specifications is so high that it requires a very compute-intensive algorithm [KB 70] to implement them directly.

Therefore we have to restrict the specifyer's freedom as shown below (section 2.2) to obtain more efficient implementations.

2.1. Formal background

We will briefly remind the notions of algebraic specifications and rewrite systems, partly taken from [Sch 88]. For deeper explanations, we refer to [EM 85] and [DJ 90].

A *specification* of an ADT is a pair $SP = (\Sigma,E)$ where $\Sigma = (S,OP)$ is the *signature* of *SP*. *S* and *OP* are respectively a finite set of *sorts* and a finite set of *function names* with arities over *S*. *E* is a set of *conditional axioms* over Σ.

We don't define here the notions of terms and substitutions. $T_{OP}(X)$ is the set of finite terms having function names in *OP* and variables in *X*. $T_{OP}(X)$ is simply written T_{OP} when $X = \emptyset$. *Vars(t)* is the set of variables occurring in the term *t*. When $Vars(t) = \emptyset$, *t* is said *ground*, and when variables are present no more than once, *t* is said *linear.*

Inside OP we distinguish C and F, respectively a finite set of *constructors*, also called *generators*, and a finite set of *operations*, sometimes called *defined* or *derived operations*. The term algebra T_C constitutes the domain of computation. *F* is a set of function names such that $OP = C \cup F$.

From now on we consider the specifications as *conditional term rewriting systems*, which means that the equations are oriented.

A *n*-ary function $f \in F$ is defined by a list of *n*-ary *conditional rewrite rules* $\pi \Rightarrow \lambda \rightarrow \rho$, where:

1. λ is a linear tuple of $T_C(X)^n$
2. $\pi,\rho \in T_{OP}(X)$
3. π has the predefined sort *meta-boolean*.
4. $Vars(\lambda) \subseteq Vars(\rho) \cup Vars(\pi)$.

π is called the condition, which by default is equal to "`true`", and may be composed of conjunctions of sub-conditions which are in turn equalities of terms. We use the name "meta-boolean" because it is predefined and not to be confused with eventual user-defined boolean sorts.

Evaluating a term of T_{OP} means rewriting it into a term of T_C. If this operation is possible through a non-infinite sequence of applications of rewrite rules, then the rewrite system is said *finitely terminating*. If applying different sequences of rewrite rules to the same term always give the same resulting term, then the system is *confluent*. If a rewrite system is finitely terminating and confluent, then we can prove that every term admits a unique *normal form*, i.e. a term on which no more rewrite rules can be applied.

A term t_1 is said to match a term t_0 if there exists a substitution σ such that $t_1 = \sigma t_0$, and this is written:

$$t_0 \preccurlyeq t_1$$

We write σ_g the set of all *grounding substitutions*, i.e. substitutions from $T_{OP}(X)$ into T_{OP}. Given a term $t \in T_{OP}(X)$, we write *G(t)* the set of its ground instances, that is $\{\sigma t \mid \sigma \in \sigma_g\}$.

2.2. Semantics of the source language

We assume from now on that the rewrite systems to compile are confluent and finitely terminating.

Given the above definition of functions, the left-side of their equations are required to be linear (rule 1). This is a restriction due to the chosen compilation algorithm [Sch 88]. Several occurrences of the same variable must be manually transformed into an additional condition stating equality of the corresponding arguments. Thus,

$$c_1 \ \& \ \ldots \ \& \ c_n \Rightarrow f(x,x,y,z) = v(x,y,z);$$

is the same as:

```
c1 & ... & cn & (x = x') =>
        f(x,x',y,z) = v(x,y,z);
```

The evaluation is performed according to the very classical *strict* and *innermost* strategy, since it corresponds to the *call-by-value* parameter passing mechanism of most modern imperative programming languages.

It is not specified whether the terms are evaluated from the left or from the right, since it does not alter the final result in confluent rewrite systems[1].

We give below the semantics of the source language. $Rules\ (f) = (\pi_i \Rightarrow \lambda_i \to \rho_i)_{\ i=1...m}$ designates the list of rewrite rules defining the function f.

For all $c \in C, f \in F, op \in OP$ and $\sigma \in \sigma_g$:

<u>Definition:</u> $Rewr: T_{op} \to T_C \cup \{\mathtt{error}\}$

$$\text{(S1)}\ \frac{Rewr[t_i] = \mathtt{error}}{Rewr[op(t_1,...,t_n)] = \mathtt{error}}$$

$$\text{(S2)}\ \frac{\forall\ i=1...n,\ Rewr[t_i] \neq \mathtt{error}}{Rewr[c(t_1,...,t_n)] = c(Rewr[t_1],...,Rewr[t_n])}$$

$$\text{(S3)}\ \frac{\forall\ i=1...n,\ Rewr[t_i] \neq \mathtt{error}}{Rewr[f\,(t_1,...,t_n)] = Apply[Rules(f)]\,f\,(Rewr[t_1],...,Rewr[t_n])}$$

<u>Definition:</u> $Apply[R]: T^n{}_C \to T_C \cup \{\mathtt{error}\}$

$$\text{(S4)}\ Apply[\emptyset]\,f\,(t_1,...,t_n) = \mathtt{error}$$

$$\text{(S5)}\ \frac{\lambda_1 \not\preccurlyeq\ f\,(t_1,...,t_n)}{Apply[(\lambda_i \to \rho_i)_{\ i=1...m}]\,f\,(t_1,...,t_n) = Apply[(\lambda_i \to \rho_i)_{\ i=2...m}]\,f\,(t_1,...,t_n)}$$

$$\text{(S6)}\ \frac{\sigma\lambda_1 = f\,(t_1,...,t_n) \quad Rewr[\sigma\pi_1] = \mathtt{true}}{Apply[(\pi_i \Rightarrow \lambda_i \to \rho_i)_{\ i=1...m}]\,f\,(t_1,...,t_n) = Rewr[\sigma\rho_1]}$$

$$\text{(S7)}\ \frac{\sigma\lambda_1 = f\,(t_1,...,t_n) \quad Rewr[\sigma\pi_1] = \mathtt{false}}{Apply[(\pi_i \Rightarrow \lambda_i \to \rho_i)_{\ i=1...m}]\,f\,(t_1,...,t_n) = Apply[(\pi_i \Rightarrow \lambda_i \to \rho_i)_{\ i=2...m}]\,f\,(t_1,...,t_n)}$$

$$\text{(S8)}\ \frac{\sigma\lambda_1 = f\,(t_1,...,t_n) \quad Rewr[\sigma\pi_1] = \mathtt{error}}{Apply[(\pi_i \Rightarrow \lambda_i \to \rho_i)_{\ i=1...m}]\,f\,(t_1,...,t_n) = \mathtt{error}}$$

FIGURE 1. Semantics of conditional rewriting

Observe that the definitions of *Rewr* and *Apply* are mutually recursive.

When evaluating a term, it may happen, if the rewrite system does not respect *sufficient-completeness*, that none of the equations cover the particular case, leading to the rule S4. The result is then an error, or more concretely, an exception is raised.

From rule S5, we deduce that when several rules may apply, the first one - in the source text - is always selected. This enforces the procedural side of specification. If this effect is not desired, the user must take care that none of the equations overlap, or have the compiler issue appropriate warnings.

A final issue concerning the source language is that we do not allow equations between generators. There are two reasons for this. First, rewrite systems with such equations need to undergo some transformations to be compilable, and in mixed prototyping we do not want to alter the structure of the source specifications: This decision was taken to help the developer stay familiar with his model, the structure - i.e. the operation names - being the sole hook he has to orient himself inside the generated code. Second, as shown in [CKM 89], contructors with equations need very special inverses, programmed as iterators, which breaks the orthogonality of the development methodology.

2.3. Semantics of the target language

In this section we describe the semantics expected from the target language $\mathcal{TL}$.

1. The actual computation order is determined by the target language rules or compilers.

Concerning the correctness of mixed prototyping with respect to the semantics of the source specifications, we refer to [CK 90]. Although our implementations make heavy use of dynamic binding, we will not bother to define this property formally, because the selection of the pertinent functions for a given implementation is entirely determined at program startup and does not change during the whole execution. Hence we reduce the correctness proof to the case where function calls are resolved statically.

The target language $\mathcal{TL}$ has respectively Expressions E, Meta-Boolean expressions B and Terms Te:

$\mathcal{TL} = \{ E, B, Te \}$, where

$E ::= Te \mid \texttt{if}\ B\ \texttt{then}\ E1\ \texttt{else}\ E2 \mid \texttt{no\_match}$

$B ::= \texttt{Generator}\ (Te) = c \mid Te_1 = Te_2$

$Te ::= x \mid c^{-1}(Te)/p \mid op(Te_1,...,Te_n)$

Symbol x denotes any variable, c any constructor, op any n-ary function and p any position. *Generator* is a testor function which calculates the normal form of its argument and returns the most external symbol (at position ε) of the resulting term. The generator inverse is written c^{-1} and returns a tuple - the list of arguments of the given generator - whose member selection is noted "/".

The semantics is given by the fig. 2: This is in fact the output of the compilation algorithm given in section 2.4. $Eval_\gamma$ evaluates an expression E in an environment γ where γ is a grounding substitution that assigns values to the variables of X.

The empty list is noted ε. For all $x \in X$, $s \in S$, $p \in \aleph^*$, $Te, B, E1, E2 \in \mathcal{TL}$ and $\gamma \in \sigma_g$:

Definition: $Eval_\gamma$: $\mathcal{TL}_E \rightarrow T_{op} \cup \{\texttt{error}\}$

(T1) $Eval_\gamma\ \texttt{no\_match} = \texttt{error}$

(T2) $Eval_\gamma\ x = \gamma(x)$

(T3) $$\frac{(Eval_\gamma\ Te)\ [\varepsilon] = c}{Eval_\gamma\ c^{-1}(Te) = c^{-1}\ (Eval_\gamma\ Te)}$$

(T4) $$\frac{Eval_\gamma\ Te_i = \texttt{error}}{Eval_\gamma\ op\ (Te_1,...,Te_n) = \texttt{error}}$$

(T5) $$\frac{\forall\ i=1...n,\ Eval_\gamma\ Te_i \neq \texttt{error}}{Eval_\gamma\ op\ (Te_1,...,Te_n) = Rewr\ [op(Eval_\gamma\ Te_1,...,Eval_\gamma\ Te_n)]}$$

(T6) $$\frac{Eval_\gamma\ B = \texttt{true}}{Eval_\gamma\ \texttt{if}\ B\ \texttt{then}\ E1\ \texttt{else}\ E2 = Eval_\gamma\ E1}$$

(T7) $$\frac{Eval_\gamma\ B = \texttt{false}}{Eval_\gamma\ \texttt{if}\ B\ \texttt{then}\ E1\ \texttt{else}\ E2 = Eval_\gamma\ E2}$$

(T8) $$\frac{Eval_\gamma\ c(t_1,...,t_n) = Te}{Eval_\gamma\ c^{-1}(Te) = \{t_1,...,t_n\}}$$

(T9) $$\frac{\forall\ p=1...n,\ Eval_\gamma\ c(t_1,...,t_n) = Te}{Eval_\gamma\ (c^{-1}(Te))/p = t_p}$$

Definition: $Eval_\gamma$: $\mathcal{TL}_B \rightarrow \{\texttt{true},\texttt{false}\}$

(T10) $$\frac{Eval_\gamma\ Te = \texttt{error}}{Eval_\gamma\ (\texttt{Generator}\ (Te) = c) = \texttt{error}}$$

(T11) $Eval_\gamma\ (\texttt{Generator}\ (Te) = c) = (\texttt{Generator}\ (Eval_\gamma\ Te) = c)$

(T12) $$\frac{(Eval_\gamma\ Te)\ [\varepsilon] = c}{Eval_\gamma\ (\texttt{Generator}\ (Te) = c) = \texttt{true}}$$

(T13) $$\frac{(Eval_\gamma\ Te)\ [\varepsilon] \neq c}{Eval_\gamma\ (\texttt{Generator}\ (Te) = c) = \texttt{false}}$$

FIGURE 2. Semantics of the target language

The evaluation strategy of c^{-1} and `Generator` is call by value (rules T3 and T10), whereas the `if then else` instruction has its usual non-strict semantics (rules T6 and T7). The generator inverse is not defined for zeroadic constructors, since we will never want to decompose an atomic value.

The compilation algorithm generates `if-then-else` tree structures, with `no_match` in the leaves corresponding to cases left undefined by the source axioms.

2.3.1 Object-oriented features of the target language

Our purpose is to be as general as possible in the choice of the target language. We want to use a minimal set of mechanisms in order to be largely language-independent and to make the prototyping process as intuitive as possible.

What we need is an object-oriented language with the following characteristics (See for instance [Weg 87] for an explanation of object-oriented terminology):

- Abstract classes and deferred methods.
- Single inheritance, although multiple inheritance is no impediment.
- Inclusion polymorphism and dynamic binding.
- Polymorphic variables and heterogeneous data structures.
- Access to supermethod, for testing, as described in section 7 on page 17.
- Encapsulation.
- Strong and static typing.
- Class attributes, by opposition to class members, are necessary to implement the protocol object mechanism. This is sometimes available through metaclasses in pure object-oriented languages.
- Restricting the visibility of some features to descendant classes only. This is useful for the realization of various kinds of auxiliary functions, be it at the specification level - *hidden operations*, declared in the `body` part of a CO-OPN module - or at the implementation level, for instance when dealing with memory-management.
- Non-primitive operations, i.e. non-redefinable functions, are necessary in cases where the generated code is deemed too sensitive to be redefined by the user. This might concern e.g. complex filtering algorithms for ADTs, or acute synchronization sequences in concurrent contexts. Note that programming languages do generally not prevent people from *redeclaring* functions, but these redeclarations will never be taken into account in OOMP since they are not visible from the abstract class.
- Overloading, in the sense of ad-hoc polymorphism, is needed since specification languages also offer this facility. Object-oriented languages usually include this possibility, although Eiffel constitutes here a notable exception.
- Genericity is useful for direct implementation of generic modules of the specification language.
- Exception handling capacity is profitable to cleanly handle situations where sufficient-completeness is not respected.

Except as noted, all of these features are available in main-stream object-oriented languages like Eiffel, C++ and Ada95.

From now on we use interchangeably the terms *sub-class* and *derived class*, and the terms *redefine* and *override*.

2.4. The compilation algorithm

The compilation process consists mainly of a translation of pattern-matching into a procedural form. For this, we propose Schnoebelen's algorithm [Sch 88] adapted to OOMP. This choice is guided by the completeness of the algorithm: it is correct given the assumption that the rewrite system satisfies termination and confluence, and it is capable of compiling conditional axioms, as well as some other interesting extensions.

To tailor the compilation algorithm to OOMP, we only need modifying the code generation. We consider here that the abstract code is evaluated in the same environment as the concrete one, since they both implement the rewrite rules with identical semantics. Therefore we do not need any mixed rewriting process as the general case described in [CK 90].

Code generation for OOMP imposes the following distinctive conventions:

- For each ADT, provide a modular and hierarchical organization to the object code. There may be some problematic cases, such as modules defining either no new sorts, or, on the opposite, several sorts, thus complicating the task of attributing the operations to a specific class. The solutions to these problems must be solved in accordance with the possibilities of the target language.
- Supply dispatching calls instead of static calls when the target language needs this distinction.
- Express all functions in terms of a limited set of constructors, generator inverses and testors.

These principles will be detailed in the next sections. But before that, we will describe the rules of Schnoebelen's compilation algorithm.

The function *Compile_For* compiles a *n*-ary function *f* from its list *R* of defining rules, and from a non-empty set *T* of *n*-tuples of ground terms of $T^n{}_C$, which constitutes a description of its domain (or input arguments).

For all $x \in X$, $c \in C$, $k, p \in \aleph^*$, $B, E1, E2 \in \mathcal{TL}$:

<u>Definition:</u> *Compile R For:* $T^n{}_C \rightarrow \mathcal{TL}$

(C1) *Compile* $\emptyset$ *For* T = `no_match`

(C2) if $T \subseteq G(\lambda_1)$:

$Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{\ i=1...m}\ For\ T =$ `if` B `then` $E1$ `else` $E2$

where

$B = CompileRH_{\lambda 1}(\pi_1)$

$E1 = CompileRH_{\lambda 1}(\rho_1)$

$E2 = Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{\ i=2...m}\ For\ T$

(C3) if $T \cap G(\lambda_1) = \emptyset$:

$Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{\ i=1...m}\ For\ T = Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{\ i=2...m}\ For\ T$

(C4) if $T \not\subseteq G(\lambda_1)$ and $T \cap G(\lambda_1) \neq \emptyset$:

$Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{\ i=1...m}\ For\ T =$ `if` B `then` $E1$ `else` $E2$

where

$B =$ `Generator` $((c^{-1}(x_k))/p) = c$

$k.p = choose\ (NMP_{\lambda 1}\ (T))$

$c = \lambda_1[k.p]$

$T' = \{\ t \in T \mid t[k.p] = c\ \}$

$E1 = Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{i=1...m}\ For\ T'$

$E2 = Compile\ (\pi_i => \lambda_i \rightarrow \rho_i)_{i=2...m}\ For\ (T - T')$

FIGURE 3. Rules for Compile_For

The *choose* function of rule C4 is a function that chooses a minimal position inside any finite subset of $\aleph^*$. This function must be *regular*, which means that if $P' \subseteq P \wedge (choose(P) = p \in P') \Longrightarrow choose(P') = p$. The *choose* function can e.g. simply take the leftmost-outermost position, although better heuristics exist [Sch 88].

The function $NMP_{t0}\ (T)$, the non-matching positions of the set T w.r.t t_0, is defined as $NVP(\mathrm{t}_0)$ - $MP_{t0}\ (T)$. $NVP(t_0)$ is the non-variable positions of t_0, i.e. the set of symbols not representing variables, whereas $MP_{t0}\ (t)$, the matching positions of a ground term t w.r.t t_0, is defined inductively by

$\forall\ t_0 \in T_{op}(X)$, $\forall\ t \in T_{op}$:

- $\varepsilon \in MP_{t0}\ (t)$ iff $t[\varepsilon] = t_0[\varepsilon]$
- $p.i \in MP_{t0}\ (t)$ (with $i \in \aleph$) iff $p \in MP_{t0}\ (t)$ and $t[p.i] = t_0[p.i]$

Finally, $MP_{t0}\ (T)$ is defined as $\cap_{t \in T} MP_{t0}\ (t)$.

CompileRH is an auxiliary function which compiles the condition and the right hand of the current axiom into respectively the *B* and *E1* parts of rule C2.

<u>Definition:</u> $CompileRH_{\lambda} : T_{op}\ (X) \rightarrow \mathcal{TL}$

(C1') if t is some $\lambda/k.p$:

$CompileRH_{\lambda}\ (t) = (c^{-1}(x_k))/p$

(C2') if $op\ (t_1,...,t_n) \neq \lambda/p\ \forall\ p \in P(\lambda)$:

$CompileRH_{\lambda}\ op\ (t_1,...,t_n) = op(CompileRH_{\lambda}t_1,...,CompileRH_{\lambda}t_n)$

FIGURE 4. Rules for *CompileRH*

This concludes our presentation of the compilation algorithm. For its proof of correctness, termination and monotonicity, we refer directly to [Sch 88].

3. Principles of mixed prototyping

We place ourselves in a context where we not only have to execute efficiently the given specifications, but also require that the object code be maintainable. The goal is to allow the developer to modify incrementally the generated code to specialize it according to particular purposes. He could for instance need to add sophisticated input-output capabilities, or to optimize speed and memory use.

3.1. Related work

Several authors have already explored the process of compiling ADTs expressed by the means of algebraic specifications (see for instance [Kap 87] and [Sch 88]) and the algorithms are generally independent of the target implementation language. In actual tools however, the choice of a programming language is always dictated by some specific objective, and therefore they offer no continuity in the development process:

- When the purpose is to ensure a smooth transition from the specification formalism to the implementation language, *functional languages* offer the best alternative. For algebraic specifications to be executed efficiently, they must be *refined,* which means that operational information must be added. They tend by this process to become more functional than axiomatic in their nature. Therefore there is no paradigmatic leap between the source and the target language and it follows that the transition is easier to manage for humans. Another advantage is that the translation process is really straightforward, both environments offering the same level of abstraction. Functional languages also usually ensure safer semantics than other languages, and give better promises that the correctness of the formal specifications will not be violated by the implementation. This is the way followed by the KorSo project with the specification language Spectrum [BFG+ 93] that is to be translated into languages such as ML [Har 86] and Haskell [HJW 92]. The disadvantage of this method is that the developer who wants to perform modifications has to cope with machine-generated code. Besides that, functional languages still aren't as well-accepted as supposedly more efficient procedural languages.

- Another obvious ambition can be to directly produce very efficient object code. The chosen target language is usually a *procedural* one like C, which also offers good portability. In this case the generated code is less legible, since cluttered by low-level details like compilation of pattern-matching and memory-management. The source and the target language do not provide comparable levels of abstractness, which implies that the code cannot readily be modified by humans who want to employ more clever algorithms or representations. The solution proposed in tools that compile the abstract data types of LOTOS, like Lola [MM 88] or Ceasar [GT 93], is to incorporate compiler directives inside the specification modules in the form of comments, e.g. "`extern`" or "`implemented by`". This presents the inconvenience that the two environments (of automatically-generated and hand-written code) cannot cooperate in a way independent of the internal representation of a data type. For instance, a function generated by the tool cannot work on an externally defined representation of the same data type. A final remark can be made about the type-safety of this kind of languages: they often do not support the notions of ADT and code reuse. In order to overcome this problem, current tools resort to generic pointers, and thus lose the benefits of static type-checking.

- Another trend, represented by the *Larch project* [GHW 85], does not try to synthesize implementations of ADTs, but relies upon the formal description to check that hand-written code does not violate the specifications. To this purpose, two formalisms are used: the first one, the Larch Shared Language, is essentially the usual kind of algebraic specifications, and deals with the relationships between abstract data types and their associated operations. The second formalism is the abstract model-oriented family of Larch Interface Languages, which are specialized extensions of programming languages like Modula-2 or C++. An interface language provides invariants, abstract input-output assertions of the functions being implemented, and functions to map concrete values onto abstract ones. This approach explicits the fact that algebraic specifications currently do not support the notion of mutable object, although it is an active research field, and is therefore not well suited for extraction of implementation-level information such as invariants.

3.2. Advantages of mixed prototyping

Mixed prototyping is a methodology for incrementally replacing abstract modules, i.e. automatically generated, with concrete ones, i.e. hand-coded. This can be done while continuously preserving the possibility of testing the modified prototype: Figure 5 illustrates a hierarchy of modules under development where some modules are abstract, some completely concrete, and some others in an intermediate - or *semi-abstract* - state[1]. This situation still allows to execute the prototype, provided that it is in a minimally coherent state. The concept of mixed prototyping was first introduced by Choppy [Ch 87] using the Ada 83 programming language [Ada 83], primarily for its support for genericity. A similar approach was taken in [HS 85], using Prolog as implementation language, but this latter work is valuable essentially for proving properties of programs, since Prolog does not provide the attributes usually expected from a software engineering language.

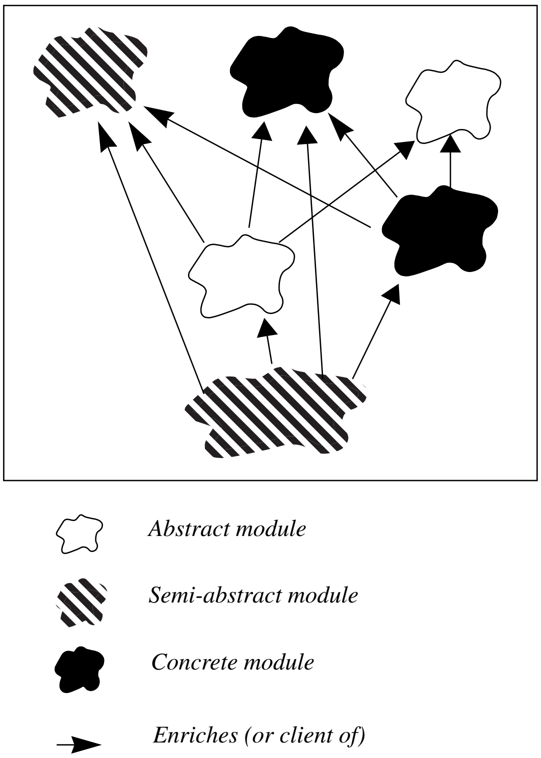

FIGURE 5. The concept of mixed prototyping

1. Note that the arrows are reversed w.r.t. the usual representation in algebraic specification.

Compared to the family of tools presented above, this idea is certainly closest to the kind which generates procedural languages, because the abstract code is not meant to be read or modified by humans. Rather, the intent is to give the opportunity to add pieces of software that are automatically and harmoniously incorporated into the environment. This means that the automatically generated modules can take advantage of all the low-level optimization techniques developed for compiling ADTs, provided that they are compatible with the general implementation protocol which ensures a clean interaction between the abstract and the concrete modules.

Other benefits of mixed prototyping are:

- No need for stubs or driver routines for the interconnection of abstract and concrete code, since they are written in the same language and are designed to cooperate following a unique protocol.
- The previous result also reduces the need for integration tests.
- Mixed prototyping supports both top-down and bottom-up development schemes.
- It encourages systematic refinement schemes.
- Reuse of concrete basic modules brings instant efficiency in new developments.

The correctness of partial implementation of models, which this methodology is based on, was proved by Choppy and Kaplan in [CK 90].

3.3. Benefits of object-orientation

We want to apply these results to the frame of object-oriented languages, because they offer a builtin paradigm, which is to allow several implementations for the same abstraction. To this purpose, we adapt the terminology to fit to the usual object-oriented vocabulary: From now on we will talk of *abstract* classes, *symbolic* classes, and *concrete* classes (Figure 6 depicts this organization). The former two are generated by the compiler, and the latter is hand-written. The abstract class provides only functions and no internal representation: that way the concrete class will not be littered with irrelevant data. The symbolic class, so called because it treats the values of the ADT as symbolic - or tagged - items, provides the minimal implementation required for

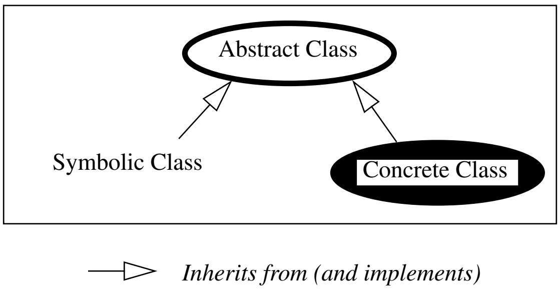

FIGURE 6. Object-oriented view of mixed prototyping

the algorithms of the abstract class to work properly.

Use of object-orientation adapts advantageously the principles of mixed prototyping to modern programming languages and paradigms. Compared to previous results, we can state the following additional improvements, which when needed will be detailed in the following sections:

- The programmer autonomously redefines and specializes operations of the abstract class. The prototyping tool does not need to know what changes have been done, only which class to use as implementation. Therefore it is not required to parse the concrete code and then merge it with an adequately regenerated abstract code: It becomes a pure code generator and is consequently easier to retarget to new programming languages. The tool is reduced to a simple compiler.
- There is no hidden mechanism: We exploit the natural resources of the object-oriented paradigm. The prototyping process gains transparency.
- Client classes need not be modified nor even recompiled when the developer switches forth - and even possibly back - between the symbolic and the concrete classes: the methodology reduces the impact of change.
- Memory management is transparent to the clients. The abstract class can even totally discharge its derived classes of this burden.
- Heterogeneous distributed systems are addressed by additional input-output facilities.
- The use of dynamic binding brings incremental loading of new implementations within reach. We can thus insert changes to a running prototype, provided that we keep track of all instances for eventual conversions. For more details we recommend [Cop 92].
- Finally, as a primitive way of checking the correctness of the concrete code w.r.t. the formal specifications, we show how to add assertion and comparison mechanisms through automatically generated sub-classes.

We also introduce some advances in the compilation of pattern-matching for mixed prototyping: *testors* and *syntactic equality* are generated as portable functions.

- The purpose of a testor is to support pattern-matching: it verifies the tag of a value, i.e. finds out which generator it was created by. In the field of mixed prototyping we want to perform this test while staying free from implementation details, since this tag information will usually not be directly available in concrete classes. This is achieved by having the derived classes define a function, call it "`Generator`", which returns an enumerated type corresponding to the different generators of the given ADT. The portable testor function has in fact the same role as the *generator inverse* w.r.t. selector functions in the jargon of ADTs.
- Syntactic equality is needed for compiling conditional rewrite systems. We let the abstract class export a redefinable equality operator, initially expressed as comparisons between normal forms, using the above mentioned portable testor. That way, we gain total transparency towards internal representation and memory management schemes.

By decoupling the client from the provider we obtain the dynamicity and the flexibility that is needed for a prototyping method to be attractive.

The superiority of OOMP rests on the use of specific mechanisms such as polymorphism and dynamic binding.

4. A small example

A complete example of Stack-of-Naturals is given in the appendices. Here, for brevity's sake, we concentrate on a smaller case.

Code extracts are in Ada95, or a pseudo-code inspired from it. Note that in Ada95, functions returning an abstract type must themselves be abstract, i.e. they can have no default implementation. To circumvent this, the return types are declared class-wide:

```
type Abstract_Nat is abstract new Root_ADT
   with private;
function "+" (X,Y: in Abstract_Nat) return
   Abstract_Nat'Class;
```

To enhance legibility, the class-wide type is renamed into the name of the sort it implements:

```
subtype Nat is Abstract_Nat'Class;
function "+" (X,Y: in Abstract_Nat) return
   Nat;
```

Suppose now we have specified and wanted to implement an ADT of natural numbers. The compiler would then directly generate an abstract class `ADT_Nat` and a derived class `Symbolic_Nat`.

4.1. Practicality of the abstract class

The abstract class is automatically generated by the compiler and is the interface of the ADT. It can be considered as a "container" inside which the developer hides the implementation of his choice, be it symbolic or concrete. The abstract class has four uses, which are described below.

4.1.1 The role of public interface

First of all, the abstract class constitutes the public interface of the class hierarchy, i.e. it defines the contract through which clients can use the ADT (or *enrich* it, in the philosophy of algebraic specifications): They do not know which of the derived classes (the symbolic or the concrete one) is functioning as the actual implementation provider. The abstract class uses language constructs, e.g. the keyword `abstract`, which prevent clients from trying to create direct instances of it. Those requests will eventually be delegated to the active sub-class.

4.1.2 The portable algorithm provider

The abstract class provides default algorithms for the operations specified on the ADT. They are coded in an implementation-independent way. These can later be redefined (*overridden* in the Ada terminology) in descendant classes by more efficient ones if necessary. For instance, natural numbers are usually defined in terms of a list having as length the number to represent. Given the following partial specification,

```
ADT Nat;
INTERFACE
   SORTS nat;
   GENERATORS
      zero : -> nat;
      succ : nat -> nat;
```

FIGURE 7. Partial specification of Naturals

the term `succ(succ(succ(zero)))` is then interpreted as the value `3`.

The specification of addition is defined by axioms deduced from the property of associativity, which, from an operational point of view, describes a recursive process where one of the arguments is progressively reduced to zero, while the resulting value is simultaneously incremented by one:

```
add-zero:   zero + y = y;
add-succ:   (succ x) + y = x + (succ y);
```

FIGURE 8. Specification of addition on Naturals

In the abstract class, this is roughly translated into the following (`Succ_Inverse(X)` being conceptually the same as taking the predecessor of `X`):

```
function "+" (X,Y: in ADT_Nat)
   return Nat is
begin
   if Generator(X)=zero then
      return Y;
   else
      return Succ_Inverse(X) + Succ(Y);
   end if;
end "+";
```

FIGURE 9. Pseudo-code for the abstract implementation of addition

This definition is as promised independent of the actual representation given to natural numbers: it works, provided that the derived classes implement the functions "`Generator`", "`Succ`" and "`Succ_Inverse`"[1].

1. A constructor function "`zero`" is also needed but it does not show in the given example.

4.1.3 The protocol class

The derived classes also must conform to the interface of the abstract class, which obliges them to provide an internal representation for the ADT, as well as to redefine the associated low-level operations. Adequate programming language keywords are employed to enforce this request - `abstract` in Ada95, `deferred` in Eiffel or pure `virtual` in C++ - so that the target language compiler will complain if necessary. On the other hand, if a derived class exports more operations than specified, this additional functionality is never used since the clients do not know of it.

4.1.4 The implementation container

Finally, the abstract class is where the information about the active derived class is stored and can be retrieved. OOMP does not only implement ADTs in terms of class hierarchies, but also provides mechanisms allowing more flexibility during the prototyping process, while keeping the advantages of strong and static typing. By using the abstract class as interface, and by resorting to polymorphism, we eliminate the need to modify or recompile client classes. We must however provide a means of designating the derived class to use as implementation, without mentioning its name in the client's code. That's the purpose of the *prototype object*.

The notion of prototype object stems from single-hierarchy languages such as Self [UR 87]. Single-hierarchy means that classes are not static entities, but real values regarded as first-class-citizens. In the case of OOMP, this helps selecting provider classes after compile-time. Thus we do not need to edit any source file except the configuration module when it is necessary to switch from one implementation to another.

The selection of an implementation with respect to the developer's indications is performed at the last moment before link-time, by generation of a *configuration routine*, which acts as a universal client in the client-supplier graph (see Figure 10). Appendix B gives a textual example. During the execution of this routine, at program startup, the initialization method of the abstract class is called to set the hierarchy's *prototype object*, which is a specimen of one of the implementation classes. The role of this object is to indicate, by its inherent type information, which implementation to use.

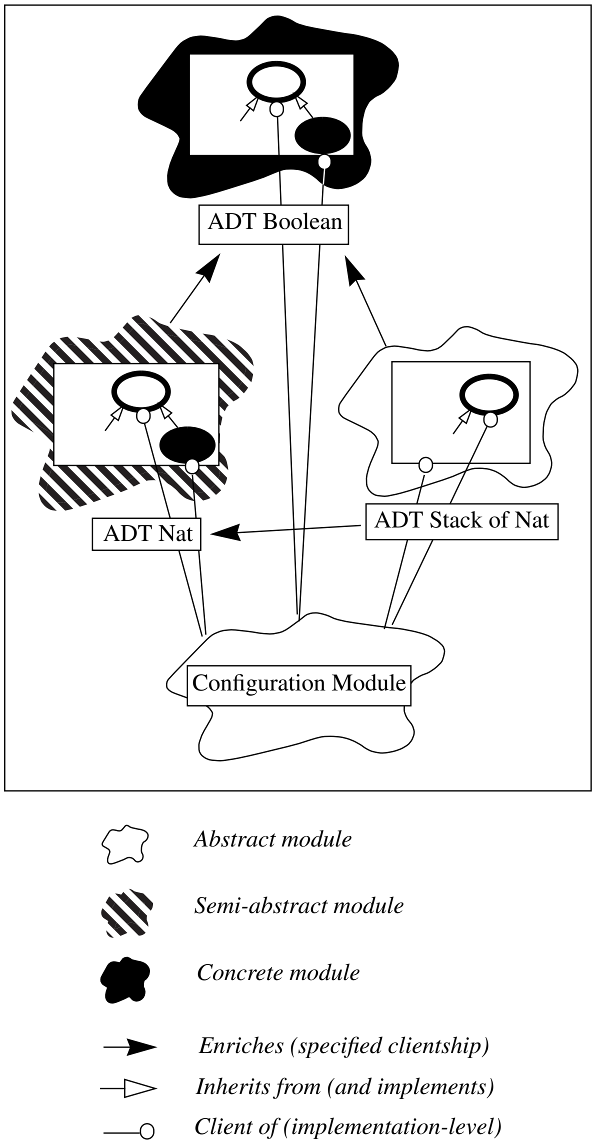

FIGURE 10. Role of the Configuration Module

Figure 10 shows a situation where the developer decides to try out his prototype, while the ADT `Boolean` is completely concrete, the ADT `Nat` semi-abstract, and the ADT `Stack` entirely abstract, i.e. treated symbolically.

The prototype object is typically used when it comes to creating *constants*, i.e. when constructing values from parameterless generators, such as `zero` for Nat and `empty` for Stack.

The general rule is that the prototype object serves as a *controlling argument* for functions that otherwise could not be a primitive of the given class because it has no parameter of that type. For instance, if we wanted to implement the constructor function `empty`, we would declare it like this:

```
function empty (Prototype: in ADT_Stack)
   return Stack is abstract;
```

The argument `Prototype` is needed in order to tell the target compiler that this function is a primitive. In other words, it can be redefined in derived classes, and the code that will be really executed is selected at run-time by inspection of the actual argument. A call to `empty` would look like this:

```
empty (Stack_Prototype) -- Ada95
```

or this

```
Stack_Prototype->empty() // C++
```

This may seem a bit awkward, but in reality this call can, in non-purely object-oriented languages, be wrapped inside a homonymous non-primitive function which hides the mechanism away from the clients. There is of course the cost of one supplemental function call to achieve this transparency.

The fact that the configuration module is generated automatically ensures that there is created instances of *only one* of the classes implementing the ADT. Had we simultaneous instances of several sub-classes, we would need additional functions for all the possible combinations of parameters:

```
function "+"(X: in Symbolic_Nat;
             Y: in Concrete_Nat) return Nat;

function "+"(X: in Concrete_Nat;
             Y: in Symbolic_Nat) return Nat;
```

We do not consider this a viable solution.

The initialization process is realized in the target language via class methods and variables, or any other similar language construct, such as the `once` methods of Eiffel.

4.2. The symbolic class

Let's go back to the example of addition on naturals. The abstract class provides a portable implementation of the axioms, but has no internal representation to put it to work. This is the role of the symbolic class. The internal representation is very standard (see appendix F): We use a variant record, with one alternative for each generator.

Figure 9 shows a rather expensive way of implementing things, especially when we observe that the result is nothing more than the concatenation of the lists representing the numbers `X` and `Y`. We could thus have a new definition of addition exploiting this characteristic:

```
function "+" (X,Y: in Opt_Symbolic_Nat)
   return Nat is
   Res: Nat := Y;
   Cursor: Nat := X;
begin
   while Generator(Cursor)/=zero loop
      Res := succ(Res);
      Cursor := Succ_Inverse(Cursor);
   end loop;
   return Res;
end "+";
```

FIGURE 11. Pseudo-code for a better symbolic implementation of addition

Figure 11 shows a non-recursive version of symbolic addition. The previous function (Figure 9) is tail-recursive, so the compiler could actually have optimized that point.

Note that we might exploit in this redefinition the knowledge of the internal representation of "`Symbolic_Nat`" in order to avoid the calls to "`Generator`" and "`Succ_Inverse`". That would however "freeze" the internal structure of this branch of the class, disallowing any ulterior derivations.

Figure 12 illustrates the new situation:

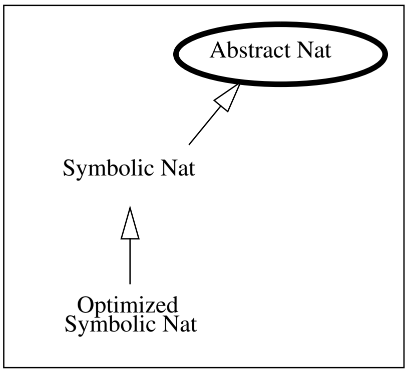

FIGURE 12. Specializing the symbolic class

Observe that we don't modify the text of the original symbolic class, because it is generated automatically, and as thus subject to accidental replacement by subsequent compilations. We create instead a sub-class having the same internal representation as the symbolic class.

4.3. The concrete class

Another, more frequent alternative to speed up the computations, is to choose an internal representation that takes advantage of a computer's hardwired treatment of natural numbers. We would then create a concrete class where the definition of addition would look like this:

```
type Concrete_Nat (Value: natural) is
   new Abstract_Nat with null record;

function "+" (X,Y: in Concrete_Nat)
   return Nat is
begin
   return Concrete_Nat(X.Value + Y.Value);
end "+";
```

FIGURE 13. Pseudo-code for a concrete implementation of addition

Figure 14 depicts the changes. Note that nothing prevents us from keeping the other implementations.

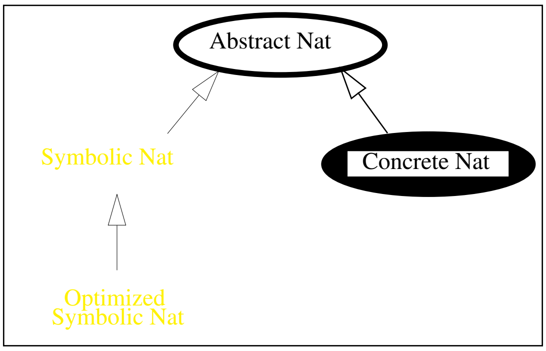

FIGURE 14. Addition of a concrete class

The interesting thing here is that the concrete class just has to provide a new internal representation, and define the associated low-level functions, such as a constructor for each generator. Non-generator operations are stored in the abstract class and coded in an implementation-independent way such that they are not required to be replaced at once. This means that new internal representations can be tested before adapting the derived operations. Then, for instance, addition can be redefined using the corresponding machine-level instruction, while subtraction would be left until later, if not valued as critical for the prototype.

4.3.1 Implementation of a concrete class

As mentioned before, there is a minimal amount of work to do before a concrete class can be used. The basic implementation is realized as follows:

- Provide an internal representation, i.e. instance variables such as the field `value` in figure 13.
- Implement the constructor functions, e.g. `zero` and `succ` in the case of `Nat`.

```
function Succ (Param: in Concrete_Nat)
   return Nat is
begin
   return Concrete_Nat(Param.Value + 1);
end Succ;
```

- For each non-constant generator, code the corresponding inverse:

```
function Succ_Inv (Self: in Concrete_Nat)
   return Nat is
begin
   if Self.Value = 0 then
```

```
      raise Succ_Inv_Error;
   end if;
   return Nat(Self.Value - 1);
end Succ_Inv;
```

- Give the `Generator` function an implementation conforming to its intended sense, for instance:

```
function Generator (Self: in Concrete_Nat)
   return Nat_Generator is
begin
   if Self.Value = 0 then
      return zero;
   else
      return succ;
   end if;
end Generator;
```

- Finally, each derived class must supply a dummy instance which will serve for the initialization of the prototype object. This is cleaner than having to provide a default constructor, which is the solution usually applied in object-oriented tools. For example:

```
Concrete_Nat_Prototype:
   constant Concrete_Nat_Ref :=
      new Concrete_Nat(0);
```

When the developer decides to use the concrete class as implementation, the configuration routine will tell the abstract class to use the constant `Concrete_Nat_Prototype` as prototype object instead of e.g. `Symbolic_Nat_Prototype`.

It is important to respect the naming scheme given in this example, otherwise the prototyping tool cannot generate correct configuration routines.

5. The development scheme

Given the previous example, we can formulate the development scheme advocated for mixed prototyping:

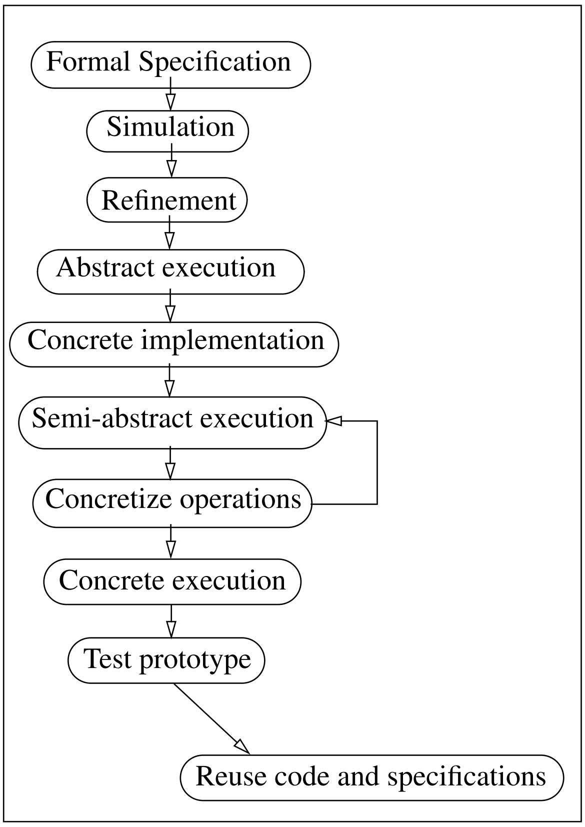

FIGURE 15. The mixed prototyping process

This illustrates the fact that the prototype can be executed at any moment. The benefits of object-orientation do not directly show on this figure: it provides a flexible and intuitive approach. Moreover, it gives better guarantees for correctness of the prototype w.r.t. the specifications, since validated modules are never modified even when the implementation of an imported ADT is changed.

6. About the generator inverse method

The flexibility of mixed prototyping rests on the systematic definition and application of generator inverses. It may sometimes be complicated to find appropriate semantics for them, as shown in [CKM 89], where a complete algorithm is presented to help their design. This is especially true in presence of equations between constructors (See 9.1.1 on page 20).

It is not strictly necessary to sort out generators from other operations: They could be automatically identified, and the specification transformed in order to remove all equations between them [Com 89]. Schnoebelen's compilation algorithm is also capable of doing this.

However, the need for generator inverses enforces the so-called *constructor discipline* [GH 78] during the specification, because generators must be considered as distinct from other operations in order to easily build the inverses. This gives place to a very imperative approach to specification, where constructors mainly serve as a means of selecting cases through pattern-matching, like in some functional programming languages, e.g. ML.

One could state that to distinguish generators from derived operations it is necessary to take premature decisions about the "representation" of the ADT. This is not really true, because the distinction is required only right before the compilation: higher-level simulation tools do not need this information. This fact is emphasized in the algebraic specification language PLUSS [Bid 89], which offers three explicit completion states according to the level of achievement of the specifications: *sketches*, the most abstract category, *drafts*, where constructors are identified, and *specs*, for the final stage, where all the properties of the software have been expressed.

Mixed prototyping can live without generator inverses, but the alternatives are more constraining. Implementation granularity can be exploited: If the derived operations which need pattern-matching are implemented at the same time as the generators, then inverses are not needed [CKM 89]. Another way is to impose a bottom-up approach to module implementation, and require the specification of an extensive set of observer and selector operations to compensate for the abolition of filtering [Roq 94].

Our purpose is however to take advantage of polymorphism to allow a finer implementation granularity, at the level of functions. Moreover, resorting to the systematic reuse of generator inverse is the most general technique, and also brings new improvements to the generation of abstract code, as shown below.

6.1. Further applications of generator inverse

The pair generator-inverse and portable selector form the building block for a language- and representation-independent implementation of algorithms. In the following subsections we describe new operations on ADTs that can be automatically generated. They can be considered as predefined, since they may be systematically synthesized by the specification compiler, at no programming cost.

Adding new predefined operations unfortunately requires modifying the specification compiler. In *reflective systems*, for instance TRPL [She 91], extensions, such as iterators, can be written in the same language as the source specifications. But that goes beyond our scope, besides that correctness of these user-defined additions is not formally addressed.

6.1.1 Defining syntactic equality

In [CK 90], equality tests directly reuse (Ada-) predefined equality, which, as the authors explained, only works on atomic values: In presence of pointers, two values are then considered equal only if they reference the same objects in memory. The correct and portable way to implement syntactic equality is by viewing an abstract data value as a composition of generators. This can be automatically implemented as a deep pattern-matching on the values to compare[1].

6.1.2 Building textual I-O facilities

An abstract data value is printed out by default as the textual representation of the generators it is made of. The dual input operation can be implemented by a little parser capable of reading and analysing the output from the printing routine. Figure 16 shows an example where the class `Concrete_Nat` uses the default behaviour of I-O routines as defined in the abstract class.

1. This is possible because we exclude equations between generators.

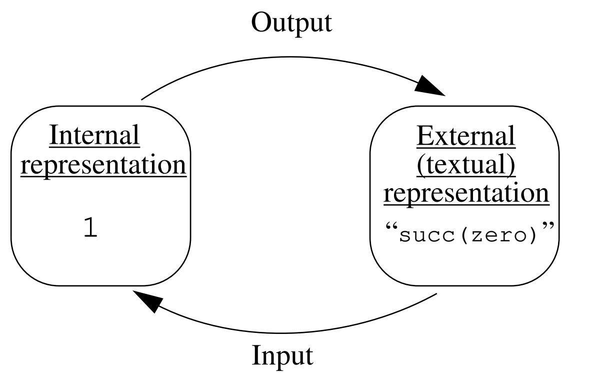

FIGURE 16. Default textual I-O routines used by `Concrete_Nat`

If these routines are redefined, it will usually be to produce more compact output, i.e. to write "1" instead of "succ(zero)".

6.1.3 Generating message-passing primitives

Following the same principle as in the previous section, we can generate binary message-passing operations (referred to as *flattening* or *marshalling* in the litterature), which allow communication of ADTs between two nodes in a distributed environment. One remarkable thing is that we can exchange *heterogeneous* data: the generators define a communication protocol valid for any implementation of a given ADT[1]. By heterogeneous, we mean data from computers with differing architectures *and* ADTs implemented by distinct sub-classes.

Each value is decomposed in terms of its generators, using the corresponding enumerated type defined in the abstract class (see appendix C for instance). The enumerated values are then converted to and from the usual integral representation of enumerated types for the actual transfer. Using for instance one byte for this tag information - to avoid issues such as endianness - allows a total of 256 different generators for each sort! The receiver reconstitutes the abstract data value by applying the generators in reverse order, the communication canal being considered as a fifo stream of bytes. Take for instance the stack-of-nat value

```
push succ(zero) on empty
```

which would have the following syntactic tree:

```
        push_on
        /     \
     succ    empty
     /
  zero
```

By simulating a traversal of this tree in a depth-first way, we find successively the tokens `push_on`, `succ`, `zero` and `empty`. By the definition given in the respective abstract classes of `Stack` and `Nat`, these tokens are unambiguously converted to the succession of bytes "`1 1 0 0`". Using tokens instead of the classical textual representation gives both more compact communication packets, and faster I-O operations, since the input needs no parsing.

These abstract routines may of course be redefined to better fit some concrete representation, but at the expense of losing the heterogeneous capability. To stay coherent, input and output routines must always be recoded together, otherwise the protocol cannot be respected.

On the other hand, if we consider a persistent store as a particular communication canal, then this kind of binary I-O routines can also provide support for software evolution in persistent systems.

As a final remark, we could say that this technique constitutes a particular case of heterogeneous hierarchical algebraic nets, as formalized by Guelfi in [Gue 94]. The specificity of our point of view is that there is only one communication sort, the byte, which is considered as predefined and as having a unique implementation on all the nodes.

6.1.4 Synthesizing conversion routines

The following establishes another application of the preceding section. As already explained, it is not allowed to have - on the same node - simultaneous instances of different derivations of a same abstract class. If under certain circumstances we need to alleviate this constraint, then conversion routines could be automatically generated as a

1. This can be compared to the idea behind the ASN-1 communication protocol.

composition of input-output routines, and that without knowing the internal representation of the different types!

Figure 17 shows how this is realized, `read` and `write` being the names of the predefined I-O routines:

```
function convert (From: in SourceType)
   return TargetType is
   To: TargetType;
   TmpStream: BinaryStream;
begin
--/ Heterogeneous I/O:
   write (TmpStream, From);
   read (TmpStream, To);
--/ Check that all data is consumed:
   if not End_of_Stream (TmpStream) then
      raise Conversion_Error;
   end if;
   return To;
end convert;
```

FIGURE 17. Pseudo-code for a conversion routine

7. Checking the correctness of concrete code

One problem arises when it comes to checking that the concrete classes correctly implement the specifications. OOMP provides a basic frame to enforce correctness:

- Low-level operations are initially declared abstract to oblige derived classes to implement them simultaneously with their internal representation. This preserves some coherence in the first generation, but ulterior derivations will not undergo the same constraints.
- The specified operations receive reusable implementations in the abstract class. The concrete class inherits these definitions and can gradually provide its own redefinitions. This allows a step-wise testing of the newly incorporated functions, which in turn permits immediate identification of accompanying errors.

Once a concrete class has completely specialized the operations on the ADT, how could we still perform automatic run-time checks?

As mentioned in section 3.1 on page 7, algebraic specifications do not provide automatically collectable information such as pre- and postconditions. Results of research in this field, see for instance [Lin 93], always involve some manual steps.

On the other hand, the code generated by the specification compiler is proved correct and can still be reused, since it remains permanently available in the abstract class. To exploit this we can have the prototyping tool generate on demand a special sub-class of the concrete class, call it the *testing class*, which has exactly the same internal representation, but which redefines the operations to make them compare the results of the user-defined functions with the ones of the abstract class.

This principle was described as early as 1972 by Hoare in [Hoa 72]. The key is to have a mapping function between the abstract values and the implementation values. In the case of OOMP we can engage the portable version of syntactic equality to this purpose. Given the following pre-existing definitions of addition:

```
--/ Previous definitions:
function "+" (X,Y: in Abstract_Nat) return
Nat;
function "+" (X,Y: in Concrete_Nat) return
Nat;
```

The prototyping tool can then generate this function on demand:

```
--/ Testing version:
function "+" (X,Y: in Tested_Nat) return Nat
is
   Result, Benchmark: Concrete_Nat;
begin
--/ Call to supermethod:
   Result := Concrete_Nat(X) +
             Concrete_Nat(Y);
--/ Call to abstract definition:
   Benchmark := Abstract_Nat(X) +
                Abstract_Nat(Y);
--/ Check if results are equal:
   if Abstract_Nat(Result) /=
         Abstract_Nat(Benchmark) then
      raise Concrete_Implementation_Error;
   end if;
   return Result;
end "+";
```

FIGURE 18. Pseudo-code for testing the concrete "+" operator

This scheme is independent of the internal representation, but necessitates a common object-oriented feature, namely *access to the super-*

method, i.e. calls to the overridden functions. This can be expressed either by renaming, as in Eiffel, or by a specific syntactic facility, as "`::`" in C++, or finally by some conversion mechanism such as the view conversions of Ada95.

The testing function is implemented as follows. First, we call the concrete function, in case the execution should raise some exception: Thus the function will have the same observable behaviour as the one it replaces. Then we call the abstract version, using the concrete internal representation to get the correct value to compare with. After that, we have to perform the comparison using the abstract version of the "=" operator. We cannot use the user-defined equality operator, since it is not guaranteed correct.

The method has the advantage of not necessitating editing or recompiling existing code: when the developer wants to test his implementation, he notifies the prototyping tool, which in turn synthesizes an adequate sub-class.

This technique is nevertheless rather inefficient in its current realization since most operations of the abstract class are defined recursively, and the compiler may not always manage to derecursify the algorithms. Each recursive call will dispatch back to the testing function, before being properly redirected. This side-effect actually provides a more thorough testing mechanism, since all intermediate values are also controlled. The intensive redispatching is a consequence of the requirement that only one implementation of an ADT may have active instances at any given moment.

If we relax this latter restriction, then the symbolic class of the ADT could be used to provide the benchmark, giving even better guarantees for the correctness of the concrete class. In this context, a new source of CPU cycle consumption would be the conversion of the concrete input parameters to their equivalent symbolic representation. This can fortunately be done automatically, with a composition of input-output routines as described in section 6.1.4 on page 16.

Other possibilities for testing the concrete code are:

- To use `theorems`[1] of the specifications to express some properties of the operations that should only be exploited in the testing functions. This could lead to a more economic execution of run-time tests. If for instance an axiom expresses the commutativity of addition, it could lead to infinite recursion if implemented by a naive compiler. But declaring this property as a theorem could serve for testing. The following specification:

```
AXIOMS
   add-zero:   zero + y = y;
   add-succ:   (succ x) + y = x + (succ y);

THEOREMS
   add-comm:   x + y = y + x;
```

  Would produce something like this:

```
function "+" (X,Y: in Tested_Nat)
   return Nat is
   Result, Theorem_Result: Concrete_Nat;
begin
--/ Perform intended operation:
   Result := Concrete_Nat(X) +
             Concrete_Nat(Y);
--/ Check theorems:
   Theorem_Result :=
      Concrete_Nat(Y) + Concrete_Nat(X);
--/ Check if results are equal:
   if Abstract_Nat(Result) /=
      Abstract_Nat(Theorem_Result) then
      raise Theorem_Violation_Error;
   end if;
   return Result;
end "+";
```

- Have the user write his own pre- and postconditions, which would then only be exploited in the testing class. To make this process systematic, the abstract class would have to define, for each operation, two initially empty boolean functions for expressing pre- and postconditions. The concrete class could afterwards redefine them as necessary. Note that the Eiffel language provides built-in constructs for invariant assertions and pre- and postconditions.

8. Other uses of object-orientation

OOMP resorts to inheritance and polymorphism as a means of switching between differing implementations of a same abstract data type. One can

1. In CO-OPN, `theorems` are formulas similar to `axioms` except that they express properties that are not intended to be interpreted.

wonder if object-orientation could also serve other purposes, or if the mechanism described in this paper excludes other possibly more meaningful applications of object-orientation.

8.1. Facilitating input-output operations

Until now specification compilers had a hard time when it came to implementing input and output of ADTs. With OOMP each value automatically 'knows' how to perform I-O operations on itself, even in the case of complex type compositions. Note that we need a reference to the protocol object to select the right input routine: just as with constant generators, new objects cannot be created out of nowhere!

8.2. Implementation of object-oriented specification languages

Object-oriented specification languages still constitute an active research field - see for instance [BB 95] - and it is hard to tell what kind of mechanisms they will offer. There is however a possibility for OOMP not to interfere with the specification language inheritance hierarchies. Take for instance a `Stack` specification, having a derived `Bounded-Stack` class specification.

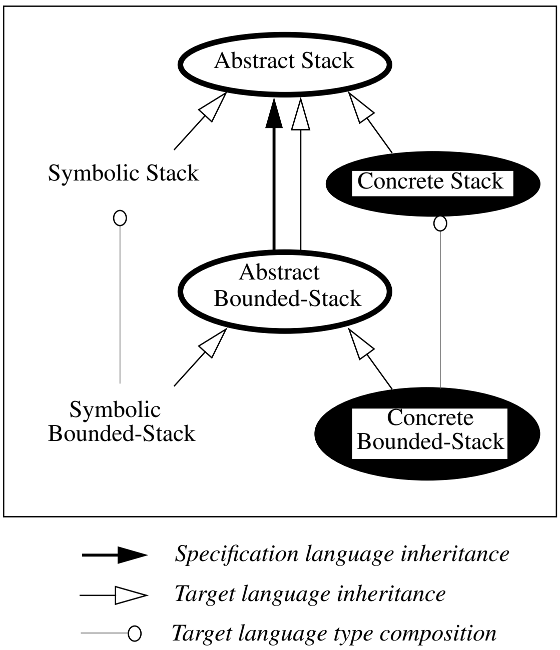

FIGURE 19. Representation of specified inheritance

Suppose that `Concrete Stack` was implemented as a bounded array. Then, if this is compatible with the specifications of `Abstract Bounded-Stack`, it would be natural to reuse this code for the `Concrete Bounded-Stack`. We do not need multiple inheritance to do this, but as indicated on figure 19, only type composition, inducing a simple client-furnisher relationship between those two classes. Therefore OOMP will never produce repeated inheritance schemes that did not originally appear inside the source specification hierarchy.

We do not further develop this discussion, because it will invariably lead to both source and target language-specific considerations. It is almost certain that object-oriented specification idioms will evolve to much more powerful mechanisms than what can be directly mapped onto current programming languages.

8.3. Subtyping

In algebraic specifications, subtypes are supported by the theory of order-sorted algebras [GJM 84]. Here, a subtype introduces *restrictions* w.r.t. its parent type, i.e. its carrier set is a *subset* of the one the parent is built on. This is useful for instance for cleanly expressing partial functions [GM 87].

This definition is contrary to the usual notion of subtype in object-oriented languages, where a derived class also is considered a subtype of its parent class. Current main-stream languages offer only extension inheritance, which means that one can only add, and not retract, operations or components to the parent class.

Some programming languages, such as Ada, offer some support for subtyping in the algebraic sense, however by a mechanism that is parallel to the one of inheritance.

Therefore the restrictive form for subtyping can only be simulated instead of directly represented - and enforced - in traditional object-oriented programming languages. A way to achieve this is to apply the *flattening* transformation described for LOTOS [ISO 88].

8.4. Coercion

Coercion is a means of reusing "code" in algebraic specifications. An ADT can thus import the properties of another type and merge them with the ones it defines for itself, without there being any subtype relationship. Implicit type conversions are often supplied by the language. Provided that the semantics of the imported operations are not violated by additional axioms[1], coercion can theoretically be implemented by code sharing. From an object-oriented point of view, this is feasible by the means of class derivation. But, considering that we should introduce no new hierarchical relationship in the implementation, coercion is only workable through the classical notion of type composition, or, in the worst case, by duplication of the original ADT.

8.5. Genericity

Does OOMP allow the same mechanism for the implementation of generic modules? In other words, can we implement generic modules of the source language as equivalent generic modules of the target language, and still keep the benefits of mixed prototyping? The answer depends greatly on how generic modules are instantiated in the source specification language. If instantiations can occur anywhere in the source text (call them *inline instantiations*), then it becomes harder to hide the inner workings for the support of mixed prototyping, such as the initialization of the prototype object.

In current versions of CO-OPN, inline-instantiations are not allowed: a generic module may only be instantiated in a separate ad-hoc module which can, in turn, be shared by client modules. This ensures that the instantiation will be made at program startup, and the initialization of the prototype object can be performed as usual by the configuration routine.

9. Compiler Optimizations

The optimizations that the specification compiler can perform are twofold: they are either textual, i.e. they enhance legibility but have little impact on performance, or they can be aimed at pure efficiency.

1. Proving the conformance of new axioms is unfortunately undecidable.

9.1. Syntactic simplifications

Let's mention some possible optimizations that contribute to rendering the abstract and the concrete code more elegant.

9.1.1 Recognition of generator inverses

The compiler could recognize certain operations as being the inverse of a generator. We might for instance have an operation "`pred`" defined on natural numbers:

```
GENERATORS
   zero : nat;
   succ : nat -> nat;
OPERATION
   pred : nat -> nat;
AXIOM
   pred (succ x) = x;
```

FIGURE 20. Definition of `pred` on `nat`

The operation `pred` would then be a specification of the inverse of `succ`, and the compiler would not have to rely upon unspecified functions like `succ_inverse` for the implementation of `nat`. This idea was developed by Roques in his thesis [Roq 94]. The drawback is that this rule must be systematically enforced: the compiler will either refuse to process the module, or have to supply the inverses when they are not explicitly specified. Mixing specified and unspecified inverses would rather tend to confuse the user: what happens if he specifies a generator inverse without being aware of it, or if he defines less inverses than needed for a given generator? To avoid these problems the specification language should be extended with a section devoted only to generator inverses, call it the `SELECTOR` part, the generic term used for this kind of functions. For instance:

```
GENERATORS
   zero: -> int;
   succ _ : nat -> nat;
SELECTOR
   pred _ : nat -> nat;
OPERATIONS
   _ + _ : nat nat -> nat;
```

FIGURE 21. Explicit declaration of generator inverses as selectors

We think that this requirement makes the prototyping process more complex, and adds more operational information than is really needed for an ADT to execute correctly: In fact, it does nothing more than *naming* the inverses.

9.1.2 Simplification of unary generator inverses

Natural numbers are a typical example of a sort with a unary generator, namely `succ`. A naive compiler would produce this, as if `succ` took more than one argument:

```
--/ Tuple of arguments for Succ:
type Succ_Args is record
   N1: Nat;
end record;

function Succ_Inverse (N: in Abstract_Nat)
   return Succ_Args is abstract;
```

FIGURE 22. Naive implementation of `Succ_Inverse`

Whereas it could do the following, sparing the introduction of a new record type[1]:

```
function Succ_Inverse (N: in Abstract_Nat)
   return Nat is abstract;
```

FIGURE 23. Simpler formulation of `Succ_Inverse`

This kind of optimizations is however not always recommended since it weakens the orthogonality of the methodology, by affecting the interface of the functions. Although the result is more elegant, it may confuse the user.

9.2. Low-level compiler optimizations

As mentioned in section 3.2 on page 8, usual optimization techniques, taken among other from the field of functional languages, can be applied to the generation of abstract and symbolic classes:

- 'Remembering' the result of constant operations. By constant operations, we mean those without parameters. Since side-effects are not allowed, functions always return the same result. This idea is also applicable to the case of constant generators to implement structural sharing: this time it's not the value, but the reference to the dynamic instance that will be retained.
- Elimination of tail-recursion is possible because all the data structures are homogeneous (in fact they are declared heterogeneous, but at run-time they always will contain objects of the same class). This means that once a function has been selected through dynamic binding, it is not necessary to redispatch for recursive calls.
- Another application of the previous observation is that all recursive calls, not only the tail-recursive ones, can be transformed into static function calls, provided that the target language allows such distinctions. Mutual recursion (two functions calling each other) cannot profit from this optimization because it would lead to a coupling between these functions disallowing redefining them separately.
- As mentioned in [Gar 89], the compiler can recognize certain patterns of sorts, based on the profile of their generators. For instance, a sort having only constant generators can be translated into an enumerated type of the target language. Similarly, sorts with exactly one constant and one recursive generator can be directly mapped onto the predefined natural numbers of the target language.

When applying the mentioned optimizations, there remains the condition that the internal representation be packaged inside an appropriate structure to stay compatible with the object paradigm.

10. Memory management considerations

One major issue in the implementation of algebraic specifications is the intensive use of dynamic memory. This is a consequence of the functional style of programming, where objects most of the time survive the functions that created them. Moreover, objects are *immutable*, i.e. considered as constant values, so client modules must instantiate new ones to reflect a change of state, while the old values are conceptually good for destruction. Finally, algebraic specifications favour recursive data types, because it helps keeping

1. In fact, a good compiler would recognize from the signature that the sort can be directly implemented as builtin naturals.

abstractness, and because specification languages often have no predefined form of container structures. All these factors summed up explain that automatic implementations gladly resort to structural sharing.

Object-oriented languages help relieving client modules from the burden of destroying objects created in the heap. Some environments offer a general-purpose garbage collector, as in Eiffel [Mey 87]. This induces unfortunately a hidden distributed cost, which may reveal unsafe in real-time applications. In other languages, the root class can be used to implement memory reclamation, or can inherit this capability from other classes. Ada 95 provides two predefined classes, called the *controlled* classes, whereas C++ allows the programmer to redefine the *new* and the assignment operators. In the latter two cases, one basically has two options:

- Either let the root class take care of the complete memory management scheme,
- or just use the root class to define a global frame, and implement the details of memory-management inside the derived classes.

In the case of mixed prototyping, the second solution seems at first sight better adapted since the internal representation can change radically between the symbolic and the concrete versions of the same ADT. For instance, integer numbers use lists as supporting representation during their symbolic treatment. They will however receive a compact form, without pointers, in their corresponding concrete instances, justifying no structural sharing.

10.1.Reference-counting schemes

Reclamation of shared objects is classically performed through reference-counting. In object-oriented languages, this technique is often implemented using so-called *smart pointers*, These are defined as wrapper-classes for the actual information, and operate by intercepting the creation and deletion messages. We observe that the remedy may be worse than the problem we wanted to cure: the goal of structural sharing was to spend less time copying instances, but we end up in the worst case doubling the number of objects in memory, with one additional smart pointer for each useful object. For this method to be rewarding, the potential for structural sharing should be very high, as well as the cost of creating and copying the basic ADTs.

Another way to implement reference counting is through a global array containing all the useful references, but this imposes a static limit on the number of instances.

10.2.Considering objects as pure values

In algebraic specifications, there is no such notion as mutable objects: everything is a value. If we follow this philosophy, computing will consume more resources, but the programming model becomes much simpler and closer to the formal model. This can also be an important argument in the case of mixed prototyping, since the abstract class may impose, through the choice of function profiles, the way derived classes see memory management[1]. For instance, an abstract class may declare the following operation:

```
function "+" (X,Y: access Abstract_Int)
   return Int;
```

The **access** mode - by-reference mechanism for Ada95 objects - strongly suggests that the parameters `X` and `Y` may be used as more than sheer read-only arguments. Had the mode been by copy **in**, the interface would have been cleaner and the interpretation unquestionable.

Whatever strategy is adopted, object-oriented programming languages provide a potential for hiding away memory management from client classes, a comfort purely procedural languages do not offer.

11. Mixed prototyping with Ada 95

In this chapter we present some aspects of OOMP that give noteworthy results in Ada95.

11.1.Syntactical considerations

Ada95 allows the implementation to be syntactically closer to the source specifications than object-oriented languages with the distinguished

1. In languages where a function must conform to the profile of the function it redefines.

receiver[1] convention. Given for instance the following boolean term expressed in CO-OPN:

```
num is in set
```

In Ada95, it gives:

```
is_in (num,set)
```

Which is only a transformation from infix to functional notation, whereas in C++ we would have to write:

```
set->is_in(num)
```

To simplify the rules for dispatching, the Ada95 compiler performs static, and if necessary, dynamic checks to see that all controlling arguments of a primitive function are instances of the same derived class. This is coherent with, and even enforces, the OOMP requirement that only one implementation of an ADT may be used at any given moment.

Ada95 also allows a fixed set of user-defined operators, which is practical to keep resemblance with the source specifications. This is possible as well in C++ and Eiffel, with some more freedom in the choice of operator names.

11.2.Polymorphism

In Ada95, dynamic binding is enforced with the so-called *class-wide* type, which is the polymorphic type of the language. It is the type of every variable or field in the client modules. It also is the return type of all the functions of a class, primitive or not. This is to emphasize the fact that the value actually returned may be instance of any class derived from the abstract class.

In C++ the notion of polymorphic type is melted with the one of pointer, which means that objects necessarily have by-reference semantics to benefit from dynamic binding.

Finally, in pure object-oriented languages, there is no such distinction: Every type is polymorphic.

11.3.Name-space management

In CO-OPN, some operations can be hidden inside the body of a specification, meaning that client modules cannot 'see' nor use them.

1. The *receiver*, or target object, *of a message* is said *distinguished* because it stands before the name of the method, and because it is the only *controlling argument*, i.e. the one that determines the run-time polymorphism.

In the target language the abstract class must let the derived classes see and redefine these operations. In C++, this is a typical application of the *protected* visibility clause. In Ada95, we use private declarations and *child units* for this purpose.

In Ada the names of predefined types are not reserved, but considered as exported by the package Standard. This gives the neat possibility of directly reusing the names of the specified sorts, while still keeping the ability the employ the predefined types: integer, natural and boolean are typical examples of frequently used sort names.

11.4.Indefinite types

The new *indefinite* types of Ada95 are useful for enforcing information hiding[2]: private types can be exported as *indefinite* to prevent clients from seeing the discriminants needed for the instantiation. This means that clients are obliged to use the exported constructor functions.

In the case of OOMP, clients normally import the abstract class, which adds up to the same effect as using indefinite types: it is not allowed to create instances of an abstract type in Ada95. But if a client decides to import directly a derived class instead of the abstract ancestor, then the information hiding mechanism of indefinite types will be imposed by the Ada language.

11.5.Argument passing mechanism

Following the philosophy of the Ada95 language that objects have by-value semantics, and not by-reference, function profiles are declared in the abstract class to be with *copy-in* parameters. The language specifies however that the underlying mechanism is in fact by-reference, preserving thus both the identity of the objects and performance.

Using by-value semantics is also closer to the principles of algebraic specifications. For the symbolic class to implement structural sharing in these conditions is however tricky, because of Ada's rules for taking the address of an object:

- The language states that it is not allowed to create a reference to an object passed as argument by copy-in, even by the means of the overruling `Unchecked_Access` attribute.

2. They are similar to the opaque types of Modula-2.

- We could use the `Address` attribute and perform some `Unchecked_Conversions` to bypass the rules designed to prevent the programmer from getting a reference to a local variable. This is however unsafe under the general conditions of OOMP, where a client is not required to follow the same memory management policy as its provider classes. This method would forbid a client from storing an instance of an imported class in a local variable, which is rather unreasonable in a functional context !

Our solution is hybrid: we create a dynamic copy of the first element of the structure to share. The following nodes already reside on the heap, through the same technique. If we wanted, in the case of natural numbers, to build the symbolic value "5" on top of the value "4", we would finally only share a structure corresponding to the value "3". The situation is illustrated on fig. 24:

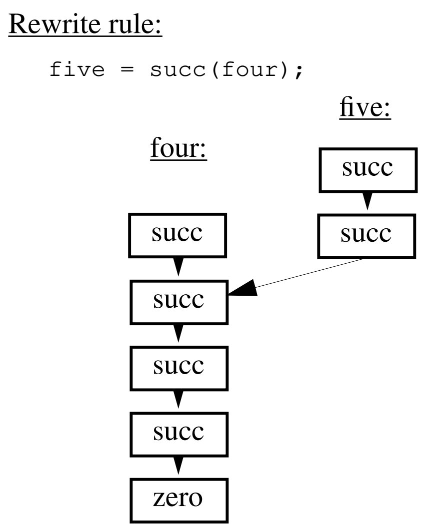

FIGURE 24. Partial structural sharing with by-value semantics

The memory-management semantics are localized inside the generator functions. See e.g. appendix F: Copying of the first element is done with an *initialized allocator*, Ada's equivalent to the copy constructor of C++. Thus the prototyping tool is not required to generate an ad-hoc copy function.

If we changed the memory management policy to perform deep copies instead, the `Adjust` function would have to be overridden. This function is inherited from the predefined `Controlled` class, and specifies the behaviour of assignment on derived instances.

12. Comparison with previous work on mixed prototyping

The most advanced tool for mixed prototyping [CK 90], namely ASSPEGIQUE+, merges the abstract and the concrete code through generic modules to avoid modifying the clients of an ADT. They will however need recompiling.

The notions of inheritance and genericity are very close, but not totally equivalent, as shown in [Mey 86] and confirmed by the fact that newer languages, such as Eiffel, Hybrid, C++ and Ada95 provide both tools.

In OOMP the abstract class is used as a generic type, since it has no internal representation and requires its descendants to implement related operations such as constructors and destructors. In other words, it cannot be instantiated. Moreover, the inheritance hierarchy formed by the abstract class and its concrete class is normally limited to two layers. It therefore is connected to the *horizontal* kind of genericity described in [Mey 87]: The concurrent instantiations of the generic class are all on the same level.

The difference resides in the use of inheritance: OOMP lets the user autonomously redefine and specialize operations of the abstract class. He can also reuse the abstract code when necessary, which allows interesting combinations. The prototyping tool does not need to know what changes have been done to the class hierarchy it generated.

On the other hand, in systems relying upon sheer genericity, when the user wants to reuse the abstract code, the tool has to complete the concrete code with dummy procedures, otherwise the generic instantiations cannot take place. This also means that the prototyping tool must know how to parse the concrete code, and then regenerate the generic module to merge adequately the abstract and the concrete parts. Unfortunately this technique is very language dependant and less intuitive. It also requires recompiling client modules each time a new function is implemented, even though the interface is not modified.

13. Comments on the method

In this section we will try preventively to discuss some potential charges concerning the prototyping technique we promote.

13.1. Disturbing the normal use of object-orientation

The technique exposed here obliges the user to introduce new types through the derived sub-classes. But the original type-hierarchy of the system being specified is luckily respected, since the client classes only know of the abstract class (See section 8.2 on page 19).

Moreover, implementing ADTs with an abstract class and differing implementations is a well accepted use of object-orientation. This replies to any criticism about using object-orientation target languages for not fully object-oriented specifications.

13.2. Introducing incoherencies during the concretization

In some cases, it may be necessary to concretize several derived operations simultaneously. As shown in 6.1.3 on page 16, input-output routines are an example of such tightly connected operations. To our knowledge, no object-oriented programming language provides means of forcing the collective redefinition of a group of methods, unless by declaring them as abstract. In other words, we would like to have a notion of *cluster of non-abstract functions* such that the programmer cannot redefine only one of them without adapting the others. There is no implementation granularity defined at this level.

We can however generate tests to be performed at prototype startup to see that the programmer is not trying to use a concrete input routine with a corresponding abstract output routine. This can be done by a low-level comparison of the addresses of the methods involved.

13.3. Inefficiency of the implementation

One could argue that the systematic use of run-time polymorphism is a cause of inefficiency. This is no longer relevant in languages where the dispatch table is completely determined at compile-time[1]. And there already exists possible remedies: some Eiffel compilers are able to transform dynamic into static binding. Work on eliminating unreachable procedures in object-oriented programs, such as [Sr 92], focus on lowering the size of the resulting binary code. The ambitious BRUNEL project [SKM 94] aims at several object-oriented specific optimizations such as link-time detection of class hierarchies where only one derived class provides instances. This allows collapsing the inheritance hierarchy and performing static procedure calls or even inlining, reducing consequently both the code size and the alleged slowness.

14. Conclusion

In this paper we have shown how object-orientation gives a new dimension to mixed prototyping: It is more intuitive because it uses the natural resources of object-oriented programming. It brings more flexibility, allows a faster modify-compile-execute cycle, and gives better guarantees for the correctness of the concrete code. We have demonstrated some openings towards dynamic, distributed and persistent systems. If ADTs constitute a privileged domain for object-orientation, it remains to be seen how far OOMP can be applied in other directions. Our current work actually shows encouraging results in the field of prototyping of concurrent systems.

1. The additional indirections could however cause some trouble in demanding real-time applications.

15. References

[Ada 83] *Reference Manual for the Ada Programming Language.* ANSI/MIL-Std-1815a, 1983

[Ada 95] *Ada Reference Manual, Language and Standard Libraries,* ISO/IEC 8652:1995

[BB 95] O. Biberstein, D. Buchs, *Concurrency and Object-Orientation with Structured Algebraic Nets,* G. Bernot & M. Aiguier editors, Working papers of the International Workshop on Information Systems - Correctness and Reusability - IS-CORE'95, Research Report, Université d'Evry, France, September 1995.

[BFG+ 93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen, *The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0,* Technical Report TUM-I9311-2, Technische Universität München. Institut für Informatik, May 1993.

[BBB+ 95] D. Buchs, O. Biberstein, M. Buffo, C. Buffard, J. Flumet, J. Hulaas, G. Di Marzo, P. Racloz, *SANDS*$_{1.5}$*/CO-OPN*$_{1.5}$*: An Overview of the Language and its Supporting Tools,* Technical Report #95/133, DI-EPFL, 1995

[BG 91] D. Buchs, N. Guelfi, *CO-OPN: A Concurrent Object Oriented Petri Net Approach*, 12th International Conference on theory and application of Petri Nets, Aarhus, pp. 432-454, 1991

[BFR 93] D. Buchs, J. Flumet, P. Racloz, *SANDS Structured Algebraic Net Development System,* Research report no 71, CUI 1993, also in 14th Int. Conf. Th. Petri Nets, Tool Presentation, Chicago, 1993

[Bid 89] M. Bidoit, *PLUSS, un langage pour le développement de spécifications algébriques modulaires,* Thèse d'Etat, Université de Paris-Sud, Orsay, 1989

[Ch 87] C. Choppy, *Formal Specifications, Prototyping and Integration Tests*, Proc. of the 1st European Software Engineering Conference, 1987

[CK 90] C. Choppy, S.Kaplan, *Mixing Abstract and Concrete Modules: Specification, Development and Prototyping,* Proc. 12th Int. Conf. on Software Engineering, Nice, 1990

[CKM 89] C. Choppy, S. Kaplan, V. Meissonnier, *Mixing executable specifications and code evaluation: An approach for prototyping and integration testing,* Proc. International Congress on New Technologies for Software and Supercomputers Development, Caracas, November 1989

[Com 89] H. Comon, *Inductive Proofs by Specifications Transformation.* In N. Dershowitz, editor, Procs of the 3rd Intl Conf on Rewriting Techniques and Applications, vol 355 of LNCS, pp 76-91, Springer Verlag, Berlin, 1989

[Cop 92] J.O. Coplien, *Advanced C++ programming styles and idioms*, Addison-Wesley, 1992

[DJ 90] N. Dershowitz, J.-P. Jouannaud, *Rewriting systems,* Handbook of Theoretical Computer Science, Van Leuwen editor, North Holland, 1990

[EM 85] H. Ehrig and B. Mahr , *Fundamentals of algebraic specification 1: equations and initial semantics,* EATCS Monographs, Springler Verlag, 1985

[Gar 89] H. Garavel, *Compilation of LOTOS Abstract Data Types,* Proceedings of 2nd FORTE, Vancouver, Canada, 1989

[Gau 85] M.-C. Gaudel, *Towards structured algebraic specifications,* Proc. ESPRIT Technical Week, Bruxelles, Springer Verlag, 1985

[GH 78] J.V. Guttag, J.J. Horning, *The Algebraic Specification of Abstract Data Types,* Acta Informatica, 10:27-52, 1978

[GHW 85] J. Guttag, J.J. Horning, J.M. Wing, *The Larch family of specification languages*, IEEE Software, vol. 2, no. 5, pp. 24-36, 1985

[GJM 84] J. Goguen, J.-P. Jouannaud, J. Meseguer, *Operational Semantics for Order-Sorted Algebra*, Research Report 84-R-101, CRIN, 1984

[GM 87] J. Goguen, J. Meseguer, Order-sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems, in IEEE Symp. on Logic in Computer Science, Ithaca-NY, 1987

[GT 93] H. Garavel, Ph. Turlier, CAESAR.ADT: un compilateur pour les types abstraits algébriques du langage LOTOS, Actes du Colloque Francophone pour l'ingénérie des protocoles, CFIP '93, Montréal, Canada, 1993

[Gue 94] N. Guelfi, *Les réseaux algébriques hiérarchiques: un formalisme de spécifications structurées pour le développement de systèmes concurrents*, Thèse de doctorat, No d'ordre 3313, Université ParisXI Orsay, 1994,

[Har 86] R. Harper, *Introduction to Standard ML.* Technical Report ECS-LFCS-86-14, University of Edinburgh, Dept. of Computer Science, November 1986

[Hoa 72] C.A.R. Hoare, *Proof of correctness of data representations,* Acta Informatica 1, pp 271-281, 1972

[HJW 92] P. Hudak, S. Peyton Jones, P. Wadler, eds. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*, ACM SIGPLAN Notices, May 1992

[HS 85] J. Hsiang, M.K. Srivas, *A PROLOG Environment Developing and Reasoning about Data Types,* TAPSOFT, March 1985, pp 276-293, Berlin, 1985

[ISO 88] ISO IS 8807, Information Processing Systems, Open System Interconnection, *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO, July 1988

[Kap 87] S. Kaplan, *A Compiler for conditional term rewriting systems,* Proc. Rewriting Techniques and Applications 87, Bordeaux, Lecture Notes Computer Science 256, Springer, Berlin, 1987

[KB 70] D.E. Knuth, P. Bendix, *Simple Word Problem in Universal Algebras.* In J. Leech, editor, Computational Problems in Abstract Algebra, pp. 263-297, Pergamon Press, Oxford, 1970.

[Lin 93] H. Lin, *Procedural Implementation of Algebraic Specification,* ACM TOPLAS, Vol 15 Number 5, 1993

[Mey 86] B. Meyer, *Genericity versus Inheritance,* In Proceedings of the 1st OOPSLA, pp 391-405, Portland, Oregon, 1986

[Mey 87] B. Meyer, *Eiffel: A Language and Environment for Software Engineering*, Interactive Software Engineering, Inc, 1987

[MM 88] J.A. Mañas, T. de Miguel, *From LOTOS to C*, K.J. Turner, editor, Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland), pp. 79-84, North-Holland, 1988

[Rei 91] W. Reisig, Petri Nets and Algebraic Specifications, Theoretical Computer Science 80, pp. 1-34, 1991

[Roq 94] C. Roques, *Modularité dans les spécifications algébriques: Théorie et applications,* Thèse de doctorat, No d'ordre 3308, Université Paris XI Orsay, 1994

[Sch 88] Ph. Schnoebelen, *Refined Compilation of Pattern-Matching for Functional Languages,* Science of Computer Programming, 11:133-159, 1988

[She 91] T. Sheard, *Automatic Generation and Use of Abstract Structure Operators,* ACM TOPLAS, Vol 13, pp 531-557, 1991

[SKM 94] A.J.H. Simons, L.E. Kwang, N.Y. Mei, *An Optimizing Delivery System for Object-Oriented Software,* in Object Oriented Systems, Vol I, pp 21-44, 1994

[Sr 92] A. Srivastava, *Unreachable Procedures in Object-Oriented Programming,* in LOPLAS, Vol I, pp 355-364, 1992

[Str 91] B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison Wesley, 1991

[UR 87] D. Ungar, B. Smith Randall, *Self: The Power of Simplicity,* SIGPLAN Notices 22,12, December 1987

[Weg 87] P. Wegner, *Dimensions of Object-Based Language Design,* in OOPSLA '87 Conference Proceedings, pp. 168-182

Appendix A The Stack of Naturals example

Suppose we wanted to implement a stack of natural numbers, with some observer operations “`max`”, “`min`” and “`is in`” in addition to the usual “`top`”, “`pop`” and “`push on`”. The Stack specification would then enrich the Natural and the Boolean specifications. The Naturals would have to provide comparison operations, which means that they also would enrich Booleans.

The CO-OPN Stack specification would look like this:

```
ADT Stack;
INTERFACE
   USE Nat, Bool;
   SORTS stack;

   GENERATORS
      empty          : -> stack;
      push _ on _    : nat stack -> stack;

   OPERATIONS
      top         : stack -> nat;
      pop         : stack -> stack;
      _ is in _   : nat stack -> boolean;
      max         : stack -> nat;
      min         : stack -> nat;

BODY
   OPERATIONS ;; Hidden
      auxmax      : nat stack -> nat;
      auxmin      : nat stack -> nat;

   AXIOMS
      top(push x on s) = x;
      pop(push x on s) = s;

      n is in empty = false;
      n=m => n is in (push m on s) = true;
      n is in (push m on s) = n is in s;

      max(push x on empty) = x;
      max(push x on s) = auxmax(x,s);
      auxmax(x,empty) = x;
      (x < y) = true =>
         auxmax(x,push y on s) =
         auxmax(y,s);
      (x < y) = false =>
         auxmax(x,push y on s) =
         auxmax(x,s)

;; axioms for 'min' and 'auxmin' omitted

   WHERE
      x, y : nat;
      s : stack;
END;
```

FIGURE 25. The stack of naturals specification

Appendix B Body of the configuration module

```
with ADT_Boolean; use ADT_Boolean;
with ADT_Boolean.Concrete;
use ADT_Boolean.Concrete;

with ADT_Nat; use ADT_Nat;
with ADT_Nat.Concrete;
use ADT_Nat.Concrete;

with ADT_Stack; use ADT_Stack;
with ADT_Stack.Symbolic;
use ADT_Stack.Symbolic;

package body Configuration_Module is

   procedure Do_Config is
   begin
      Initialize(Boolean_Ref(
         Concrete_Boolean_Prototype));
      Initialize(Nat_Ref(
         Concrete_Nat_Prototype));
      Initialize(Stack_Ref(
         Symbolic_Stack_Prototype));
   end Do_Config;
end Configuration_Module;
```

Appendix C The abstract class specification

```
with Root_ADT; use Root_ADT;
with ADT_Nat; use ADT_Nat;
with ADT_Boolean; use ADT_Boolean;

package ADT_Stack is

type Abstract_Stack is abstract new
   Root_ADT.Root_ADT with private;
type Abstract_Stack_Ref is access all
   Abstract_Stack;

subtype Stack is Abstract_Stack'Class;
type Stack_Ref is access all Stack;

-------------------
-- Class methods --
-------------------

--/ Initialize the class hierarchy:
procedure Initialize
   (With_Prototype: in Stack_Ref);

--/ Reference to the prototype object:
function Stack_Prototype return Stack;
pragma inline(Stack_Prototype);

-------------------------------------------
-- Constructors, destructors and testers --
-------------------------------------------

--/ Generators of Stack:
type Stack_Generator is (empty, push_on);
```

```
--/ Create a value "empty":
function empty return Stack;

--/ Create a value "push on":
function push_on
   (P1: in Nat; P2: in Abstract_Stack)
   return Stack is abstract;

--/ Type returned by "push on" inverse:
type push_on_Arg (Nat1: Nat_Ref;
                  Stack1: Stack_Ref)
   is null record;

--/ Potentially raised by push_on_inv:
push_on_inv_error: exception;

--/ Inverse of "push on" (selector)
function push_on_Inv
   (Self: in Abstract_Stack)
   return push_on_Arg is abstract;

--/ Tell which generator created me:
function Generator (Self: in Abstract_Stack)
   return Stack_Generator is abstract;

--/ Test for equality:
function "=" (Left,Right: in Abstract_Stack)
   return Standard.Boolean;

--------------------------
-- Specified operations --
--------------------------

--/ Operation "top"
function top (P: in Abstract_Stack)return Nat;

--/ Operation "pop"
function pop (P: in Abstract_Stack)
   return Stack;

--/ Operation "is in"
function is_in (N: Nat; S: in Abstract_Stack)
   return ADT_Boolean.Boolean;

--/ Operation "max"
function max (P: in Abstract_Stack)
   return Nat;

--/ Operation "min"
function min (P: in Abstract_Stack)
   return Nat;

-------------------------
-- Standard operations --
-------------------------

--/ Textual I/O (Not Ada95 conformant)
procedure Write (Stream: access TextStream;
                 Self: in Abstract_Stack)
procedure Read (Stream: access TextStream;
                 Self: out Abstract_Stack);

--/ Binary I/O (Not Ada95 conformant)
procedure Write (Stream: access ByteStream;
                 Self: in Abstract_Stack);
procedure Read (Stream: access ByteStream;
               Self: out Abstract_Stack);

private

--/ Auxiliary function to create a 'empty':
function empty (Prototype: in Abstract_Stack)
   return Stack is abstract;

--/ Operation "auxmax"
function auxmax (OI: in Nat;
                 P: in Abstract_Stack)
   return Nat;

--/ Operation "auxmin"
function auxmin (OI: in Nat;
                 P: in Abstract_Stack)
   return Nat;

--/ Internal representation:
type Abstract_Stack is abstract new
   Root_ADT.Root_ADT with null record;

end ADT_Stack;
```

Appendix D The abstract class body

```
with ADA.INTEGER_TEXT_IO;
use ADA.INTEGER_TEXT_IO;
with TEXT_IO; use TEXT_IO;
with Ada.Exceptions; use Ada.Exceptions;
--/ For parsing input:
with Scanner;

package body ADT_Stack is

---------------------------------
-- Class methods and variables --
---------------------------------

--/ Reference to prototype object:
Prototype_Object: Stack_Ref := null;

--/ Initialize the class hierarchy:
procedure Initialize
   (With_Prototype: in Stack_Ref) is
begin
   if Prototype_Object = null then
      Prototype_Object := With_Prototype;
   end if;
end Initialize;

--/ Reference to the prototype object:
function Stack_Prototype return Stack is
begin
   return Prototype_Object.all;
end Stack_Prototype;

-------------------------------------------
-- Constructors, destructors and testers --
-------------------------------------------

--/ Create a value "empty":
```

```
function empty return Stack is
begin
   return empty (Stack_Prototype);
end empty;

--/ Test for equality:
function "=" (Left,Right: in Abstract_Stack)
   return Standard.Boolean is
   Left_Cursor : Stack := Left;
   Right_Cursor : Stack := Right;
   Left_Gen : Stack_Generator;
begin
<<recursion>>
   Left_Gen := Generator(Left_Cursor);
   if Left_Gen /= Generator(Right_Cursor) then
      return false;
   elsif Left_Gen = empty then return true;
   elsif Left_Gen = push_on then
      declare
         Left_Inv : push_on_Arg :=
            push_on_Inv(Left_Cursor);
         Right_Inv : push_on_Arg :=
            push_on_Inv(Right_Cursor);
      begin
         if Left_Inv.Nat1.all /=
            Right_Inv.Nat1.all then
            return false;
         else
            Left_Cursor :=
               Left_Inv.Stack1.all;
            Right_Cursor :=
               Right_Inv.Stack1.all;
            goto recursion;
         end if;
      end;
   end if;
end "=";

-------------------------
-- Specified operations --
-------------------------

--/ Operation "top"
function top (P: in Abstract_Stack)
   return Nat is
begin
-- axiom: top(push x on s) = x
   if Generator(Stack(P))=push_on then
      declare
         inv: push_on_Arg :=
               push_on_Inv(Stack(P));
         x : Nat := inv.Nat1.all;
      begin
         return x;
      end;
   else
      raise_exception
         (Incomplete_Definition'identity,
         "abstract top");
   end if;
end top;

-- Idem for operations "pop", "is in",
-- "max" and "min"
```

```
--/ Private operations:
--  -------------------

--/ Operation "auxmax"
function auxmax (OI: in Nat;
                 P: in Abstract_Stack)
   return Nat is
begin
-- axiom: auxmax(x,empty) = x
   if Generator(Stack(P))=empty then
      declare
         x: Nat := OI;
      begin
         return x;
      end;
   else
-- axiom: (x < y) = true =>
--    auxmax(x,push_on(y,s)) = auxmax(y,s)
      declare
         inv: push_on_Arg :=
               push_on_Inv(Stack(P));
         y: Nat := inv.Nat1.all;
         s: Stack := inv.Stack1.all;
         x: Nat := OI;
      begin
         if (x < y) = true then
            return auxmax(y,s);
-- axiom: (x < y) = false =>
--    auxmax(x,push_on(y,s)) = auxmax(x,s)
         elsif (x < y) = false then
            return auxmax(x,s);
         end if;
      end;
   end if;
end auxmax;

--/ Idem for operation "auxmin"

-------------------------
-- Standard operations --
-------------------------

--/ Textual I/O
-- ------------

-- NB: For generality's sake, we take here
-- some liberty with the standards of Ada95


procedure Write (Stream: access TextStream;
                 Self: in Abstract_Stack) is
begin
   case Generator(Stack(Self)) is
      when empty =>
            put(Stream,
               Stack_Generator'Image(empty));
      when push_on =>
         declare
            inv: push_on_Arg :=
                  push_on_Inv(Stack(Self));
         begin
            put(Stream,
             Stack_Generator'Image(push_on));
            put(Stream,"(");
            put(Stream,inv.Nat1.all);
```

```
            put(Stream,",");
            put(Stream,inv.Stack1.all);
            put(Stream,")");
         end;
   end case;
end Write;

--/ These declarations are local to "Read"
empty_tok: aliased constant Token :=
               Stack_Generator'Image(empty);
push_on_tok: aliased constant Token :=
               Stack_Generator'Image(push_on);

procedure Read (Stream: access TextStream;
                  Self: out Abstract_Stack) is

   subtype Token_Range is integer range
   Stack_Generator'Pos(Stack_Generator'First)
   ..
   Stack_Generator'Pos(Stack_Generator'Last);

   Curr_Token: Token_Range;
   Generators: Token_Array (Token_Range) :=
      (empty_tok'Access,push_on_tok'Access);

begin
   Scan_Tokens(Stream,Generators,Curr_Token);
   case Stack_Generator'Val (Current_Token) is
      when empty =>
         Self := empty;
      when push_on =>
         declare
            Stack1: Stack := Stack_Prototype;
            Nat1: Nat := Nat_Prototype;
         begin
            if Next(Stream) /= `(` then
               raise Parse_Error;
            end if;
            Read (Stream,Nat1);
            if Next(Stream) /= `,' then
               raise Parse_Error;
            end if;
            Read (Stream, Stack1);
            if Next(Stream) /= `)' then
               raise Parse_Error;
            end if;
            Self := push_on(Nat1,Stack1);
         end;
   end case;
end Read;

-- Idem for binary I/O routines

end ADT_Stack;
```

Appendix E Specification of the symbolic class

```
package ADT_Stack.Symbolic is

type Symbolic_Stack (<>) is
   new Abstract_Stack with private;

type Symbolic_Stack_Ref is access all
   Symbolic_Stack;

-------------------------------------------
-- Constructors, destructors and testers --
-------------------------------------------

--/ Constant view of the prototype object:
Symbolic_Stack_Prototype:
   constant Symbolic_Stack_Ref;

--/ Create a "push_on":
function push_on  (P1: in Nat;
                   P2: in Symbolic_Stack)
   return Stack;

--/ Inverse of push_on (selector)
function push_on_Inv (Self: in Symbolic_Stack)
   return push_on_Arg;

--/ Tell which generator created me:
function Generator (Self: in Symbolic_Stack)
   return Stack_Generator;

private

--/ Auxiliary function to create a "empty":
function empty (Prototype: in Symbolic_Stack)
   return Stack;

type Symbolic_Stack (Tag: Stack_Generator) is
   new Abstract_Stack with
      record
         case Tag is
            when empty =>
               null;
            when push_on =>
               push_on:
               aliased push_on_Arg(null,null);
         end case;
      end record;

Symbolic_Stack_Prototype:
   constant Symbolic_Stack_Ref :=
      new Symbolic_Stack(empty);

end ADT_Stack.Symbolic;
```

Appendix F Body of the symbolic class

```
package body ADT_Stack.Symbolic is

-------------------------------------------
-- Constructors, destructors and testers --
-------------------------------------------

--/ Constant objects are shared:
Shared_Empty : constant Symbolic_Stack_Ref :=
   new Symbolic_Stack(empty);

--/ Create a "empty":
function empty (Prototype: in Symbolic_Stack)
   return Stack is
begin
```

```
   return Stack(Shared_Empty.all);
end empty;

--/ Create a "push_on":
function push_on (P1: in Nat;
                  P2: in Symbolic_Stack)
   return Stack is
begin
   return Stack(Symbolic_Stack'
      (Abstract_Stack with
         Tag => push_on,
         push_on => (new Nat'(P1),
                     new Stack'(P2))));
end push_on;

--/ Inverse of push_on (selector)
function push_on_Inv(Self: in Symbolic_Stack)
   return push_on_Arg is
begin
   if Self.Tag /= push_on then
      raise push_on_inv_error;
   end if;
   return Self.push_on;
end push_on_inv;

--/ Tell which generator created me:
function Generator (Self: in Symbolic_Stack)
   return Stack_Generator is
begin
   return Self.Tag;
end Generator;

end ADT_Stack.Symbolic;
```

Appendix G The concrete class spec

```
package ADT_Stack.Concrete is

type Concrete_Stack (<>) is
   new Abstract_Stack with private;

type Concrete_Stack_Ref is access all
   Concrete_Stack;

-------------------------------------------
-- Constructors, destructors and testers --
-------------------------------------------

--/ Constant view of the prototype object:
Concrete_Stack_Prototype:
   constant Concrete_Stack_Ref;

--/ Create a "push_on":
function push_on (P1: in Nat;
                  P2: in Concrete_Stack)
   return Stack;

--/ Inverse of push_on (selector)
function push_on_Inv (Self: in Concrete_Stack)
   return push_on_Arg;

--/ Tell which generator created me:
function Generator (Self: in Concrete_Stack)
   return Stack_Generator;

-------------------------
-- Optimized operations --
-------------------------

--/ Operation "top"
function top (P: in Concrete_Stack)return Nat;

private

--/ Auxiliary function to create a "empty":
function empty (Prototype: in Concrete_Stack)
   return Stack;

type Nat_Array is array (integer range <>)
   of Nat_Ref;

type Concrete_Stack (Size: natural) is
   new Abstract_Stack with
      record
         Tab: Nat_Array(1..Size);
      end record;

Concrete_Stack_Prototype:
   constant Concrete_Stack_Ref :=
      new Concrete_Stack(0);

end ADT_Stack.Concrete;
```

Appendix H The concrete class body

```
with ADA.INTEGER_TEXT_IO;
use ADA.INTEGER_TEXT_IO;
with TEXT_IO; use TEXT_IO;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;

package body ADT_Stack.Concrete is

-------------------------------------------
-- Constructors, destructors and testers --
-------------------------------------------

--/ Share constant objects:
Shared_Empty : constant Concrete_Stack_Ref :=
   new Concrete_Stack(0);

--/ Create a "empty":
function empty (Prototype: in Concrete_Stack)
   return Stack is
begin
   return Stack(Shared_Empty.all);
end empty;

--/ Create a "push_on":
function push_on (P1: in Nat;
                  P2: in Concrete_Stack)
   return Stack is
   Aux: Nat_Array(1 .. P2.Size+1);
begin
   Aux(1..P2.Size) := P2.Tab(1..P2.Size);
   Aux(P2.Size+1) := new Nat'(P1);
```

```
   return Stack(Concrete_Stack'
      (Abstract_Stack with
         Size => P2.Size + 1,
         Tab => Aux));
end push_on;

--/ Inverse of push_on (selector)
function push_on_Inv(Self: in Concrete_Stack)
   return push_on_Arg is
begin
   if Self.Size = 0 then
      raise push_on_inv_error;
   elsif Self.Size = 1 then
      return   (Self.Tab(1),
               Stack_Ref(Shared_Empty));
   else
      declare
         New_Size : natural := Self.Size-1;
         Res : push_on_Arg(
            Self.Tab(Self.Size),
            Stack_Ref'
            (new Concrete_Stack(New_Size)));
      begin
         Concrete_Stack_Ref(Res.Stack1)
            .Tab(1..New_Size) :=
               Self.Tab(1..New_Size);
         return Res;
      end;
   end if;
end push_on_Inv;

--/ Tell which generator created me:
function Generator (Self: in Concrete_Stack)
   return Stack_Generator is
begin
   if Self.Size = 0 then
      return empty;
   else
      return push_on;
   end if;
end Generator;

--------------------------
-- Optimized operations --
--------------------------

--/ Operation "top"
function top (P: in Concrete_Stack)
   return Nat is
begin
   if P.Size = 0 then
      raise_exception
         (Incomplete_Definition'identity,
         "concrete top");
   end if;
   return P.Tab(P.Size).all;
end top;

end ADT_Stack.Concrete;
```