A C++ Frame Library : User Manual and Implementation Notes

Sabine Moisan, Jean-Paul Rigault

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A C++ Frame Library: User Manual and Implementation Notes

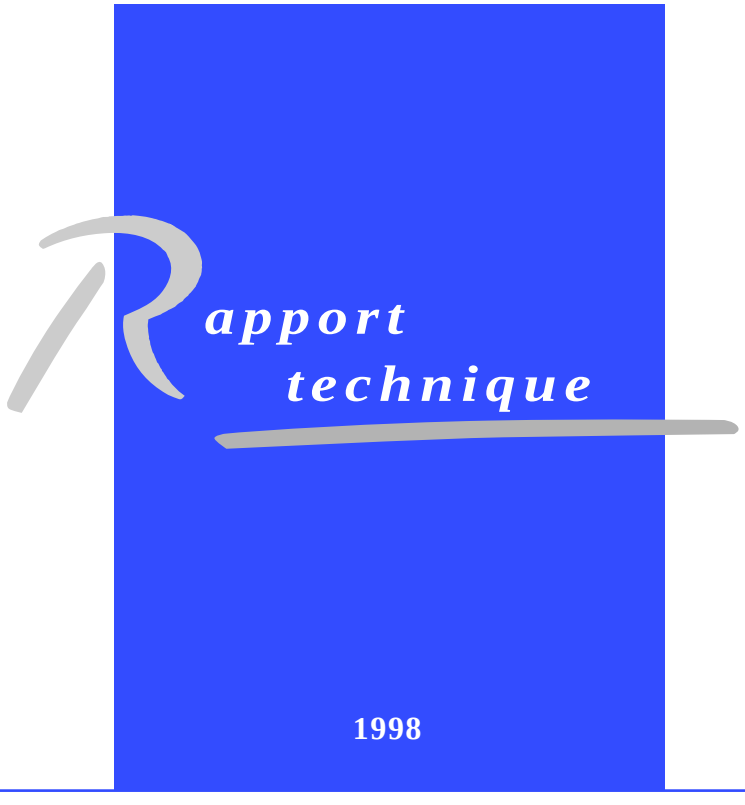
Sabine MOISAN - Jean-Paul RIGAULT

N° 0217

February 1998

PROGRAMME 3

Intelligence artificielle,
systèmes cognitifs
et interaction homme machine



ISSN 0249-6399

A C++ Frame Library User Manual and Implementation Notes

Sabine MOISAN*
Jean-Paul RIGAULT**

Programme 3 : Intelligenceartificielle, systèmes cognitifs
et interaction homme-machine

Projet Orion

Rapport technique n˚0217 - Février 1998

46 pages

Abstract: This document briefly describes FRAMELIB, a C++ library to manipulate "frames" as they are used in the Artificial Intelligence world. The library provides a general framework (!) for creating new frames (!!) by inheritance. The intended use is to generate new frames automatically, from a frame description language. But nothing prevents from using this library "manually". This document should be sufficient for directly using the library or for using it on an automatic generation basis. It also presents some design issues for those who are interested in the guts of the library. Note that FRAMELIB requires a C++ compiler supporting templates, ANSI exceptions, and RTTI (Run Time Type Information).

Key-words: knowledge-based systems, frames, C++.

(Résumé : tsvp)

* Email: moisan@sophia.inria.fr
** Email: jpr@essi.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS cedex (France)
Téléphone : 04 93 65 77 77 - international : +33 4 93 65 77 77
Télécopie : 04 93 65 77 65 - international : +33 4 93 65 77 65

Une bibliothèque de « frames » en C++

Résumé : Ce document décrit brièvement FRAMELIB, une bibliothèque écrite en C++ et destinée à manipuler des « frames », structures de représentation des connaissance utilisées en Intelligence artificielle. La bibliothèque fournit une cadre général permettant de créer de nouveau types de frames en utilisant l'héritage de classes. Typiquement, les frames seront générées ainsi automatiquement à partir d'un langage de description de frames. Mais rien n'empêche d'utiliser la bibliothèque « manuellement ». Ce document devrait être suffisant à la fois pour utiliser directement la bibliothèque et pour l'utiliser en génération automatique. Il discute également quelques décisions de conception pour ceux que l'implémentation intéresse. Il convient de noter que FRAMELIB requiert un compilateur C++ supportant les génériques (templates), les exception ANSI et l'identification dynamique de type (RTTI).

Mots-clé : systèmes à base de connaissance, frames, C++.

Table of Contents

1 Introduction

1.1 A brief introduction to “frames”

Frames are complex data structures used to represent “typical” objects or situations in Artificial Intelligence systems. Although there might be some religious issues in putting it so directly, a frame looks very much like a class in an object-oriented language such as SMALLTALK or C++. A frame describes a collection of objects; it has attributes that are called *slots*. It has also methods. Frames can inherit from each other or be combined together to form composite objects.

A main difference with regular classes, though, is that frames exhibit a two levels structure:

- first, slots (or attributes) represent roles, features involved in the typical object or situation represented by the frame;
- second, *facets* qualify these slots.

Thus, slots do not simply carry a value: their *facets* represent various features attached to the manipulation of the slot value. Some facets exhibit an *active behaviour*, that is they are associated with some function: such facets are called *daemons*. They correspond to “reflexes” that must be automatically triggered by the frame system. The circumstances when a daemon is executed depends on its type, but its execution is always transparent to the user. For example, an “if needed” daemon is triggered on a read access to the value of a slot, whereas an “if modified” daemon is triggered on a write access to the value of a slot. In any cases, the corresponding functions have one *implicit* parameter: the instance of the frame itself.

This version of FRAMELIB (V 1.0) supports four kinds of facets (including one daemon) for each slot:

- A *value facet*, which simply represents the value of the slot; this value may be of any scalar type (boolean, integer, real) or of any class type with value semantics* (such as `String`, `Vector`, `Array, Set`...). It may also be (a reference to) another frame.
- A *range facet* which represents the domain of the value facet; this range may be nothing, an interval, an enumeration...
- An “if needed” daemon, which is a function computing the value of the slot whenever the value facet has not been initialized.

* A type has *value semantics* when each object carries its own value, and when this value may be copied through initialization or assignment. Consequently, there may be first class citizen *variables* (as well as constants) of the type.

- A *default facet* which is used as the value of the slot whenever the value facet has not been initialized and no “if needed” daemon has been defined. The type of the default facet is identical to the one of the value facet. The default value must also fall within the range defined by the range facet.

One can imagine other kinds of facets and daemons, and indeed many of them have been defined in the AI literature.

Three methods are associated with slots:

- *Set value* is used to store a value into the value facet;
- *Get value* either retrieves the value from the value facet, or calls the “if needed” daemon, or returns the default value, or throw an exception, in this order, depending on the initialization status of the facet;
- *Clear* removes the value (if any) set in the value facet (not yet implemented).

1.2 Examples of frames

To give an idea, here follows the description of some frames representing objects related to persons. The description language used is close to YAKL, ORION project own frame description language.

The first frame represents the basic attributes of a person. The `family_name` and `birth_date` slots have only a (non initialized) value facet. The `age` slot has a range and an “if needed” daemon which is used here as a sort of *active value*. The `marital` status slot is of enumerated type and has a default facet.

```
Frame {

     name: Person
     comment: "General description of a person"

Attributes

     String name: family_name

     Date name: birth_date

     Integer name: age
          default: 30
          range: [0,150]
          if_needed: return today.year - birth_date.year;

     Symbol name: marital
          comment: "marital status"
          range: [single,married,divorced,widow]
          default: single
```

```
        method: void get_married(Person partner) {
             marital = married;
        }
    }
```

Of course, one may define another frame, say `Woman`, inheriting from `Person`. The new frame `Woman` adds a new slot (`maiden_name`) with a default value and no daemon. It also refines the slot `age` which is inherited from `Person`, by giving a new definition of its "if needed" daemon*. Finally, it overrides the method `get_married`.

```
    Frame {

        name: Woman
        Subtype Of: Person

    Attributes

        String name: maiden_name
             default: family_name

        Override Integer name: age
             if_needed: throw exception dont_ask;

        Override method: void get_married(Person husband) {
             marital = married;
             family_name = husband.family_name
        }
    }
```

1.3 Implementing frames in C++

A frame system must provide the following minimal set of services:

- Create a new frame with various slots and facets;
- Create a new instance of an existing frame;
- Read/write the value of a slot;
- Clear a slot, that is remove its value (not yet implemented).

Usually frame systems are implemented with traditional AI languages such as LISP and its dialects, or even in languages specialized in knowledge representation.

* In general frame systems, it is possible to redefine entirely an inherited slot (a *virtual* slot so to speak). In FRAMELIB V 1.0 we restrict ourselves to the redefinition of the inherited *daemons*. See also §3.1.

However we decided to implement a (restricted) frame system in C++ at least for two reasons:

- *Portability*: the various dialects of LISP are not portable, and specialized languages are... special.
- *Efficiency*: clearly compiled languages like C++ generate much more efficient code than interpreted ones.

Of course efficiency has its cost. The basically static nature of C++ prevents from a natural expression of most *reflexive*[*] constructs. If one tries to relax the static constraints, which is certainly possible, the immediate consequence is losing strong typing which induces lower programming safety[**] but also impaired efficiency. Indeed, the static analysis performed by C++ constitutes one major factor of its efficiency.

So we have to face the challenge of building a system exhibiting a power of expression (with some metaconstructs) sufficient for our AI applications and still respecting the C++ philosophy (strong typing and maximum static analysis).

The FRAMELIB library is a set of classes which allows the definition of new classes representing frames. The main classes of FRAMELIB are `Frame` (the abstract base class of all frames), `Slot` (the abstract base class of all slots) and its three variations (`ScalarSlot`, `ClassSlot`, and `FrameSlot`). The `Slot` hierarchy is template-based. A new frame class may be defined by deriving a class from `Frame` (or from any of its descendants). Just to give the flavour of it, this is the sketch of the definition for class `Person` seen in §1.2:

```
class Person: public Frame {
public:
     // If needed daemon definitions
     virtual int ifn_age(void) {
          return today().year - birth_date.year;
     }
     // Other if_needed daemons (§2.5.1, §2.5.3)

     // Slots
     ClassSlot<String, Person> family_name;
     ClassSlot<Date, Person> birth_date;
     ScalarSlot<int, Person> age;
     ScalarSlot<Marital, Person> marital;
```

* In C++ classes are not objects. There are no metaclasses, although there are some metaobjects, especially those related to RTTI.

** In fact, programming safety is not our main concern, since in our project,the C++ code is automatically generated, as it will be explained later. Thus most static checks can be performed before generation.

```
        // Methods
        virtual void get_married(Person partner) {
             marital = married;
        }

        // Application constructors (§2.3.3, §2.5)
        // (automatically generated by YAKL)
        //In particular,these constructors have the
        //responsability to link the «if needed» daemon
        //(ifn-age) with its slots (age)

        // Some fix code automatically generated
        // by macro FRAME_FIX_PART (§2.5)
};
```

When defining class `Woman`, we just have to add the new slot and to redefine the virtual function associated with the daemon:

```
class Woman: public Person {
public:
     // Exceptions
     DCL_EXCEPTION(Woman, dont_ask, All);

     // If needed daemon (re)definitions
     virtual int ifn_age(void) {
          throw dont_ask();
     }

     // Slots
     FrameSlot<String, Woman> maiden_name;

     // Methods
     virtual void get_married(Person husband) {
          marital = married;
          family_name = husband.family_name;
     }
     // Application constructors (§2.3.3, §2.5)
     // (automatically generated by YAKL)

     // Some fix code automatically generated
     // by macro FRAME_FIX_PART (§2.5)
};
```

As will be discussed later (§3.1) the principal design decision in FRAMELIB was to figure out how to implement the daemon definition (and *redefinition*).

Of course, frames are used as regular C++ classes, and slots may be used (most of the time) as regular members of the class (there are some restrictions, though: see §2.6):

```
Woman& w = *new Woman();
w.family_name = "Doe";
w.age = 42;

try {
     cout << “age of “ << w.family_name << " = " << w.age;
}
catch (Woman::dont_ask) {...}
```

1.4 Environment and compiler

FRAMELIB is template-based. It uses (non-template) classes nested within template classes. It does not require an automatic template instantiating mechanism, although that would be useful.

Frame and slot methods raise (throw) ANSI exceptions which must be descendants from class `FrameLib_Exception`.

When manipulating frames we need to be able to inquire for the type of a frame at run time. For this we rely on the C++ RTTI mechanism as it has been defined by the ANSI C++ committee. But we had to extend it a little (more on this in §3.2).

Finally we use some ANSI extensions: booleans (`bool`), explicit template instantiation, initialization of `const static` members...

Consequently we need a compiler close to the ANSI C++ level (as of April 95, at least). We implemented FRAMELIB with GNU **g++** (versions 2.7.0 and 2.7.2). However, the ANSI implementation in these compilers is not mature yet, and we had to fight with some compiler bugs (“internal compiler error” being the mildest!), especially in the area of exception handling.

FRAMELIB does not require any class foundation library, since it provides its own (and simple) one (see Appendix A). Anyhow, it requires the **iostream** library which comes with any C++ compiler.

2 The Frame Library

2.1 Support library

FRAMELIB provides its own library of support classes (fundamental data structures) which will be fully described in appendix A. This support library includes:

- Some common classes used throughout the library for handling exceptions (§A.1); all exceptions in FRAMELIB are derived from class `FrameLib_Exception`, which in turn derives from `exception`, the ANSI standard class;
- Class `String` (§A.2): a class encapsulating the usual (null-terminated) C character strings, with full value semantics and usual operations (concatenation with operator `+`, indexing, and also sub-stringing);
- Class `SList<Elem>` (§A.3.2) and class `SSet<Elem>` (§A.3.3): generic lists and sets; the first one is an ordered collection allowing redundancy; the second is unordered without redundancy; `Elem` represents any type with value semantics (copy operations); copy of objects are stored; both classes have full value semantics (i.e. copy operations duplicate all the elements in the collection);
- Class `SPList<Elem>` (§A.3.2) and `SPSet<Elem>` (§A.3.3): generic lists and sets of pointers; the first one is an ordered collection allowing redundancy (of pointer values), the second is unordered without redundancy (of pointer values); `Elem` represents any type; only pointers to objects are stored; both classes have pointer value semantics (i.e. copy operations duplicate only the pointers, thus the objects pointed to are shared after duplication);
- Class `Symbol_Table<Symbol>` (§A.3.4): a dictionary class; the type denoted by `Symbol` must have a method `name(void)` returning a `String` which is used as access key in the table; no redundancy is permitted; type `Symbol` must have also full value semantics, since *copies* of `Symbol`'s are stored in the table.

All classes, except `String`, have an associated iterator class (§A.3.1) which is also a generic class (`SList_Iter<Elem>`, `SPSet_Iter<Elem>`, `Symbol_Table_Iter<Elem>`...).

As usual in C++, template instantiation is a problem, since it differs from one compiler to the other. One can find how it is done for **g++** in appendix B.

2.2 Class `Frame`

Class `Frame` is the focal point of the frame library. It is the base class of all frame classes. The class maintains the set of its instances and of the instances of all its derivatives (the extension of the class). Methods are provided to extract subsets corresponding to specific types from this extension.

Note that `Frame` instances have no value semantics *by default* (no copy operators are defined). They may just be purposely cloned, that is duplicated.

2.2.1. Instance attributes of `Frame`

All the following members are *private* to class `Frame`.

`String _id`
: The unique identifier of this frame instance. It is of the form `frame_NNN` where `NNN` is some number.

`const String *_class_name`
: The name of the class this instance belongs to. It is a pointer since many instances may share the same class name. It is initialized through the `_this_class_name` static member (§2.2.2). This member helps in supplementing the RTTI mechanism (§3.2).

`String _alias`
: Another name for this instance. Contrarily to `_id`, it is not guaranteed to be unique, since it is user defined.

`SPSet<Slot> _slots_using`
: The set of (pointers to) slots referencing this frame. The cardinal of this set acts as a *reference count* of all slots referencing the frame.

`bool _valid`
: This flags indicates that the instance is currently valid (i.e. alive). This is reserved for future uses.

2.2.2. `Frame` class (static) members and methods

Static data members specific to `Frame` itself

All the following static members are *private* to class `Frame`.

`SPSet<Frame> _extension`
: The *extension* of class `Frame`, that is a set of (pointers to) all the instances of frames (that is instances of class Frame or of any class derived from it, at whatever level).

`int _cpt`
: A counter used to compute the unique identifier `_id`. It is incremented by one each time a new instance of frame is created.

`const String _ID_SEED`
: The base name for `_id`. Currently it is `"frame_"`.

`Symbol_Table<dict_entry> _exemplars_dict`
: The set of exemplars. There is one exemplar for each class derived from `Frame`. This helps to extend the RTTI mechanism (§3.2) and also to ensure persistency and virtual construction (§3.3). The structure `dict_entry` is defined as a class nested within class `Frame`.

Static data members that must be replicated in all derived class

All the following members are *private* to class `Frame`.

`const String _this_class_name`
: The name of the current class. Each class derived from `Frame` must

define such a static member.

`Frame _exemplar`
: An exemplar of this frame. This is used to introduce a new type of frame to the extended RTTI mechanism. Each class derived from `Frame` must define such a static member.

These two fields must be initialized for any frame derived from class `Frame`. A C++ preprocessor macro, `FRAME_DEF_STATIC`, help initializing them (§2.5.2).

Static methods

All the following members are *public* to class `Frame`. They make it possible to extract from the extension of Frame all the instances corresponding to a specific subtype.

```
SPSet<Frame> strict_extension(const Frame& f)
static SPSet<Frame> strict_extension(const String& cln)
SPSet<Frame> strict_extension(const type_info& ti)
```
: *Strict extension*: given either a `Frame` instance `f`, its class name `cln`, or its type information `ti` as returned by `typeid`, return the set of (pointers to) all the instances with *exactly* the same type.

```
SPSet<Frame> full_extension(const Frame& f)
SPSet<Frame> full_extension(const String& cln)
```
: *Full extension*: given either a `Frame` instance `f` or its class name `cln`, return the set of (pointers to) all the instances with exactly the same type or with a type derived from it. One may wonder why there is not an SPSet <Frame> full-extension (const type-info&)-see §3.2 for the beginning of the answer.

The following method is protected, since it is to be used only within subclasses.

```
const Frame *exemplar_of(const String& cln)
```
: Return a pointer to the exemplar of class with name `cln`. If it cannot be found return the null pointer. Exemplar handling is described in §3.2 and §3.3.

2.2.3. `Frame` instance methods

Accessors

All these methods are public.

```
const String& id(void) const
const String& class_name(void) const
const String& alias(void) const}
bool valid(void) const
```
: Accessors to the private fields with identical names but prefixed with

an underscore.

`int n_slots_using(void) const`
: Number of slots referring to this `Frame`.

Type manipulation

The following functions are public.

`virtual bool is_kind_of(const String& cln) const`
: Return `true` when the current instance has the same type or a type derived from the class with class name `cln` (§3.2).

`virtual bool is_super_of(const Frame& f) const`
: Return `true` if the current frame has a type which is a super-type of the type of `f` (§3.2).

Constructors, destructors and similar functions

This function is public.

`virtual Frame& clone(const String& al = “”) const`
: Return a new instance (actually allocated by `new`) which is an absolute copy of the current frame. The facets are simply duplicated, member-wise. Any uninitialized facet remains so.

The two following constructors are *protected* since class `Frame` is logically abstract and cannot be constructed except from its derived classes.

`Frame(const String *cln, const String& al = "")`
: Construct a `Frame` with class name `cln` and alias `al`.

`Frame(Exemplar ex, const String *cln)`
: Construct an exemplar of the class, “introducing” it to the base class `Frame` and putting it into `_exemplars_dict` (§3.2 and §3.3).

Note that, since `Frame` has no public constructor, it may be constructed only by its derivatives.

IO operators

This is a friend function.

`friend ostream& operator<<(ostream& os, const Frame& f)`
: Output operator displaying the frame class name, its id, and its possible alias.

2.2.4. Exceptions of class `Frame`

All exceptions are public nested classes.

`class Frame::all : public FrameLib_Exception`
: The base class of all `Frame` exceptions.

`class Frame::unknown : public Frame::all`

An instance was searched for within the `Frame` extension and could not be found.

`class Frame::abstract : public Frame::all`

Class `Frame` is logically abstract, but cannot be such for C++ (for a number of technical reasons). This exception signals the fact that `Frame` is trying to be used as a concrete class (for example when one attempts to clone it).

2.3 Class `Slot`

Class `Frame` itself has no slot, but most of the classes derived from it will have some. They represents the attributes of the objects together with some specific behaviour when manipulating their value.

2.3.1. Types of slots

There are three kinds of slots federated through an unique base class `Slot`. This is mainly for *efficiency* reasons since one single class would have been *logically* enough.Whereas class Slot is a regular class, the three others are template classes, with two type parameters: `T` is the value type, and `F` is the type of the frame which the slot belongs to (see §3.1 for details on the use of `F`).

- Class `ScalarSlot<T, F>`: this class corresponds to slots the value of wh.ich is a *built-in type* with full copy semantics. Parameter passing and function returning is efficiently performed *by value*. Parameter `T` should designate a C++ built-in type (`char`, `int`, `long`, `float`, `double`...) but this cannot be enforced by testing.
- Class `ClassSlot<T, F>`: this class corresponds to slots the value of which is a type with *full copy semantics* but for which passing and returning by value is not efficient. So passing (and returning whenever possible) by reference (to a `const`) is preferred instead. Typical examples include `String`, `Vector`, `Array`, `Set`... Note that one may safely use class `ScalarSlot` for these types: the only risk is poor efficiency. Here again, we cannot enforce statically the existence of the full copy semantics.
- Class `FrameSlot<T, F>`: this class corresponds to slots the value of which is another frame. Remember that frames have no value semantics. Thus the same frame instance may be freely *shared* by several slots. The value and default facets are simply *pointers* to the frame. The cardinal of `_slots_using` (`n_slots_using`) represents the number of slots sharing the current frame (§2.2.1).

Class `Slot` has private copy constructor and private assignment operator to avoid any default value semantics for slots (and consequently for frame).

2.3.2. Facets and daemons

Each slot, whichever kind, has a (private) pointer to the frame instance it belongs to (`_context`). We call this instance its *context frame*. And it has also four facets including one daemon, as already mentioned: value, range, "if needed" daemon, and default. All facets are private.

The exact type of the value and default facets depends on the type of the slot. For `ScalarSlot` and `ClassSlot`, it is necessary to know whether the value has been initialized. Thus a boolean flag is associated with these facets (forming a nested private class `Facet`). This is not needed for `FrameSlot`, since the facets are simply a pointer to a frame, and the null pointer indicates "uninitialized" well enough.

The range slot is always a *pointer* to a `Range` object. Indeed, as we shall see in §2.4, `Range` is the root class of a full hierarchy.

Finally, the "if needed" facet is implemented as a private nested class `Daemon`. This class contains a pointer to a method of class `F` (the embedding frame) without argument and returning a `T` value (for `ScalarSlot` and `ClassSlot`) or a reference to `T` (for `FrameSlot`). Thus the type of this pointer (designated as `IfDaemon`) to member is either `T (F::*)(void)` or `T& (F::*)(void)`. In the latter case, it is the responsibility of the method itself to ensure that the returned object has been correctly allocated. Associated with this pointer, one can find a boolean flag indicating that the pointer to member has been initialized.

This means that for each facet, *there must exist at least one method of the right type in the embedding frame class*. If the corresponding method is *virtual*, a derived frame will be able to redefine it, changing the behaviour of the base class daemon. This is exactly what we wished. More on this in §3.1.

2.3.3. Methods of slots

Constructors

Class `Slot` itself has no other methods than private copy operators (§2.3.1) and a default constructor needed for internal reasons.

The three kinds of slots have a (public) default constructor (still for internal reasons) and a sort of *copy constructor*:

```
ScalarSlot(F *cxt, const ScalarSlot& sl)
ClassSlot(F *cxt, const ClassSlot& sl)
FrameSlot(F *cxt, const FrameSlot& sl)
```

> "Copy constructors" for slots, used during frame cloning. The context frame must be passed also, this is why they are not copy constructors in the usual sense.

They also present the following four different regular constructors the prototypes of which vary depending on the slot type:

```
ScalarSlot(F *cxt, If_Daemon ifn, bool ifn_valid = false)
ClassSlot(F *cxt, If_Daemon ifn,bool ifn_valid = false)
FrameSlot(F *cxt, If_Daemon ifn, bool ifn_valid = false)
```

Initialize the slot with context frame `cxt` and the "if needed" daemon pointing to method `ifn`. The parameter `ifn` must point to a virtual function of class `F`, returning either a `T` or a `T&` (see §2.3.2). The boolean `ifn_valid` indicates whether the "if needed" daemon is considered valid within the current frame class or that it is the responsibility of derived class to validate it.

```
ScalarSlot(F *cxt, If_Daemon ifn, T def,
       bool ifn_valid = false)
ClassSlot(F *cxt, If_Daemon ifn, const T& def,
       bool ifn_valid = false)
FrameSlot(F *cxt, If_Daemon ifn, T& def,
       bool ifn_valid = false)
```

Initialize the slot with context frame `cxt`, the "if needed" daemon pointing to method `ifn`, and the default facet being set to `def`. The parameter `ifn` must point to a virtual function of class `F`, returning either a `T` or a `T&` (§2.3.2). The boolean `ifn_valid` indicates whether the "if needed" daemon is considered valid within the current Frame class or that it is the responsibility of derived class to validate it.

```
ScalarSlot(F *cxt, If_Daemon ifn, Range<T> *pr,
       bool ifn_valid = false)
ClassSlot(F *cxt, If_Daemon ifn, Range<T> *pr,
       bool ifn_valid = false)
FrameSlot(F *cxt, If_Daemon ifn, Range<T> *pr,
       bool ifn_valid = false)
```

Initialize the slot with context frame `cxt`, the "if needed" daemon pointing to method `ifn`, and the range facet being pointed to by `pr`. The parameter `ifn` must point to a virtual function of class `F`, returning either a `T` or a `T&` (§2.3.2). The boolean `ifn_valid` indicates whether the "if needed" daemon is considered valid within the current Frame class or that it is the responsibility of derived class to validate it.

```
ScalarSlot(F *cxt, If_Daemon ifn, T def,
       Range<T> *pr,bool ifn_valid = false)
ClassSlot(F *cxt, If_Daemon ifn, const T& def,
       Range<T> *pr, bool ifn_valid = false)
FrameSlot(F *cxt, If_Daemon ifn, T& def,
       Range<T> *pr,bool ifn_valid = false)
```

Initialize the slot with context frame `cxt`, the "if needed" daemon pointing to method `ifn`, the default facet being set to `def`, and the range facet being pointed to by `pr`. The parameter `ifn` must point to a virtual function of class `F`, returning either a `T` or a `T&` (§2.3.2). The boolean `ifn_valid` indicates whether the "if needed" daemon is

considered valid within the current Frame class or that it is the responsibility of derived class to validate it.

Value manipulation

The three kinds of slots have all the following public methods:

`void set_valid(void)`
: Validate the if "needed" daemon of the slot. The pointer to member must have been previously initialized using one of the slot constructors (possibly in a base class). Of course this supersedes the value of `ifn_valid` given at construction time.

`void clear(void)`
: Clear the value facet of the slot (that is slot becomes uninitialized).

All three kinds of slot present also the following methods to set and retrieve the slot value. They also make it possible to manipulate a slot with type `T` as if it were a simple object of type `T` (more or less–see §2.6.2).

```
T ScalarSlot::operator=(T t)
const T& ClassSlot::operator=(const T& t)
T& FrameSlot::operator=(T& t)
```
: Set the value facet to `t`. The first two operators perform a full copy, according to `T` value semantics. This means that `T` must have copy operations (copy constructor and assignment) defined, at least by default. The last operator merely sets a pointer.

```
ScalarSlot::operator T(void)
ClassSlot::operator T(void)
FrameSlot::operator T&(void)
```
: Retrieves the slot value: return the value facet if it is valid; otherwise call the "if needed" daemon if it is valid; otherwise return the default value if it is valid; otherwise throw exception `uncomputable`. Each possible value is tested against the possible range facet. Should the test fail, exception `out_of_range` is thrown. Also, if the "if needed" daemon returns a valid result, this result becomes the new contents of the value facet*.

These two operators may be respectively replaced by the following two methods (older pre-C++ style):

```
void ScalarSlot::set_value(T t)
void ClassSlot::set_value(const T& t)
void FrameSlot::set_value(T& t)
```
: Identical to `operator=`.

* This is clearly a choice. It might appear questionable in the future.

```
T ScalarSlot::get_value(void)
const T& ClassSlot::get_value(void)
T& FrameSlot::get_value(void)
```
Identical to `operator T`.

For convenience (§2.6.2) we also provide the following operator which makes it possible to use slots as methods (an accessor to itself, so to speak):

```
T ScalarSlot::operator()(void)
const T& ClassSlot::operator()(void)
T& FrameSlot::operator()(void)
```
Identical to `operator T&`

2.3.4. Exceptions of class `Slot`

Class `Slot` defines the following (nested) public exception classes:

`class Slot::all : public FrameLib_Exception`
The base class of all `Frame` exceptions.

`class Slot::bad_copy : public Slot::all`
Attempt to perform a C++ default copy of a slot.

`class Slot::uncomputable : public Slot::all`
Thrown when the value of a slot cannot be computed.

`class Slot::out_of_range : public Slot::all`
Thrown when the value of a slot does not fall within its range.

2.4 Class `Range`

2.4.1. Types of ranges

`Range` instances are objects representing domain of values. In fact there is a full hierarchy of domains federated under the base class `Range<T>`. All classes in this hierarchy are template classes depending on the type `T` of the value. `T` may be any type (scalar, built-in, user class, frame).

Class `Range<T>` itself is an abstract class with no members but a pure virtual function:

`virtual bool contains(const T& t) const = 0`
This function returns `true` if the object `t` "falls within the range". Its precise definition is under the responsibility of the derived classes.

Since this interface is the only thing that classes `Frame` and `Slot` know about `Range`, the system is completely open, and new subtypes of `Range` may be freely added.

At this time (version V 1.0), we have defined only two subtypes of `Range`:

`class Interval<T> : public Range<T>`
: An interval is defined by two values of type `T`, `tmin` and `tmax`. Method `contains` returns `true` when `tmin <= t <= tmax`. Of course this supposes that type `T` has got `operator<=`.

`class Collection<T> : public Range<T>`
: A (finite) collection of `T` objects. In other words an enumerated type. Methods `contains` returns `true` when t is equal to one of the members of the collection. This supposes that `T` defines `T::operator==`.

Both classes have a display operator (`operator<<`), mainly for debugging purposes.

2.4.2. Class `Interval`

Class `Interval` is very simple indeed. An `Interval` object is constructed from two values of type `T`:

`Interval(const T& mi, const T& mx)`
: Construct the interval with lower* bound `mi` and upper bound `mx`. The bounds are part of the interval itself.

Recall that `T::operator<=` must be defined.

2.4.3. Class `Collection`

Class `Collection` is not much more complicated than `Interval`. It simply has a richer variety of constructors:

`Collection(void)`
: Default constructor: the collection is empty (`contains` always return `false`).

`Collection(const T& t)`
: A singleton collection.

`Collection(const T tab[], int n)`
: Initialize the collection from a *copy* of the components in array `tab`, of dimension `n`. This supposes that `T` has value semantics.

`Collection(const SSet<T>& s)`
: Initialize the collection from a *copy* of the components in the (simple) set `s`. This supposes that `T` has value semantics (this is also needed by `SSet`–see §A.3.3).

`bool is_empty(void) const`
: What do you think?

`int card(void) const`
: The *cardinality* of the collection, that is its number of elements.

* In fact it does not matter which is the largest.

2.5 Generating new frames

Creating frame classes consists simply in deriving new classes from the base class `Frame` or from one of its descendants. The frame classes will have slots and daemons. However, the derivation must obey a systematic process in order to maintain the consistency of the frame system.

In the Orion project at INRIA, we generate automatically the code of the new frame classes from the analysis of a *frame description language*. This section explains how to perform this generation.

2.5.1. General structure of a frame class

There are two main parts in the definition of a new frame class:

- An application-*dependent part*, which contains the definitions related to slots and daemons; this includes also the constructors needed to initialize correctly a frame object.
- A *fixed part*, present in all frames, which contains fields and methods indispensable to maintain the frame system consistent. This part is absolutely independent of the particular frame or of its slot.

Let us revisit the first example given in §1.2 and §1.3. Here follows the full definition of the frame class `Person` in our system:

```
1    class Person: public BaseClass {
2    public:
3
4         //-----------------------------------------------
5         // Application dependent part: slots and methods
6         //-----------------------------------------------
7
8         enum Marital {SINGLE, MARRIED, DIVORCED, WIDOW};
9
10        // Functions for daemons (all slots)
11        virtual String ifn_family_name(void);
12        virtual Date ifn_birth_date(void);
13        virtual int ifn_age(void);
14        virtual Marital ifn_Marital(void);
15
16        // The slots themselves
17        ClassSlot<String, Person> family_name;
18        ClassSlot<Date, Person> birth_date;
19        ScalarSlot<int, Person> age;
20        ScalarSlot<Marital, Person> marital;
21
22        // Methods
23        virtual void get_married(Person partner) {
24             marital = married;
25        }
```

```
26
27        //-----------------------------------------------
28        // Application dependent part: constructors
29        //-----------------------------------------------
30
31        // Regular (user) constructor(s)
32        Person(const String& al = "")
33              : BaseClass(&Person::_this_class_name, al),
34                family_name(this, &ifn_family_name),
35                birth_date(this, &ifn_birth_date),
36                age(this, &ifn_age, &age_range, true),
37                marital(this, &ifn_marital, SINGLE)
38        {}
39
40    protected:
41
42       // Construction relay
43        Person(const String *cln, const String& al = "")
44              : BaseClass(cln, al),
45                family_name(this, &ifn_family_name),
46                birth_date(this, &ifn_birth_date),
47                age(this, &ifn_age, &age_range, true),
48                marital(this, &ifn_marital, SINGLE)
49        {}
50
51       // Copy constructors (for clone())
52        Person(const Person& p, const String& al = "")
53              : BaseClass(&Person::_this_class_name, al),
54                family_name(this, p.family_name),
55                birth_date(this, p.birth_date),
56                age(this, p.age),
57                marital(this, p.marital)
58       {}
59
60        Person(const Person& p, const String *cln,
61                        const String& al = "")
62              : BaseClass(cln, al),
63                family_name(this, p.family_name),
64                birth_date(this, p.birth_date),
65                age(this, p.age),
66                marital(this, p.marital)
67        {}
68
69        //-----------------------------------------------
70        // Fixed part (independent of the slots)
71        // generated by the macro call
72        //         FRAME_FIX_PART(Person, BaseClass);
73        //-----------------------------------------------
74
75    private:
76
```

```
77        static const String Person::_this_class_name;
78        static Person Person::_exemplar;
79
80   protected:
81
82       Person(Exemplar ex)
83             : BaseClass(ex, &Person::_this_class_name)
84       {}
85
86       Person(Exemplar ex, const String *cln)
87             : BaseClass(ex, cln)
88       {}
89
90   public:
91
92       virtual bool is_super_of(const Frame& f) const
93       {
94             return is_kind_of(f, *this);
95       }
96
97        virtual bool is_kind_of(const String& cln) const
98       {
99             Frame *pf = exemplar_of(cln);
100            return pf == 0 ? false: pf->is_super_of(*this);
101      }
102
103      virtual Person& clone(const String& al = "") const
104      {
105            return *new Person(*this, al);
106      }
107
108  };
```

Even in the slot dependent part, not all the code is dependent on the particular frame. The dependent code is indicated in `slanted Courier font`, like `family_name` on line 63.

2.5.2. Fixed (slot independent) part

This is the simplest part. It starts from line 69. It is entirely parametrable by the name of the frame class, here `Person`, and the name of the base class, here `BaseClass`. Thus, we defined a preprocessor macro, `FRAME_FIX_PART`, with two parameters, to generate this part.

Consequently all lines from 69 to 107 can be replaced by the single line:

```
FRAME_FIX_PART(Person, BaseClass);
```

In the fixed part one can find the declaration of two static members: the frame own class name (`_this_class_name`) and an "exemplar" for this frame (lines 77 and 78). As usual in C++, these static members *must* be defined (and initialized) *ex-*

actly once in the whole program (usually in a `.C` file). To help this initialization, we defined the macro `FRAME_DEF_STATIC(cl)` where `cl` is the frame class (here `Person`). So it is enough to put somewhere at the beginning of the `Person.C` file the line

```
FRAME_DEF_STATIC(Person);
```

to get these two statics properly initialized.

Note that the constructors come by pair. This pattern in consistently used throughout the frame class. For instance, at line 82 and 86, one can find two constructors taking an `Exemplar` as first parameter. The first one

```
Person::Person(Exemplar ex)
```

is used to construct an object of *exact type* `Person`. The second one,

```
Person(Exemplar ex, const String *cln)
```

is just a *relay constructor*. The classes derived from `Person` will invoke it, with their own class name as the second parameter. And it will just propagate this information upwards, to its base class. (Note how the first constructor calls the second one, but for its base class.)

The role and handling of exemplars will be explained in 3.2 and §3.3.

2.5.3. Slot dependent part

In the slot dependent part, one can find the definitions of frame and of everything related to them (slots, methods, daemons...). This is of course highly dependent on the frame itself and on its application.

Slots and daemons

At the beginning (line 8) we find the definition of an enumeration type which is used as the value type for slot `marital`. This is of course application dependent. Then we find (lines 11–14) one declaration of a virtual function for every slot that will be defined in this frame. This function is to be called when the ”if needed” daemon of the slot is invoked. Since it is virtual, it will be redefinable in classes derived from `Person`, changing the behaviour of the `Person`’s slot.

Note that here, only `ifn_age` is likely to have a *valid* definition in `Person` itself. This explains why line 36 includes `true` as last parameter for the constructor of slot `age` (2.3.3). The other daemon functions must also have a definition. They cannot be pure virtual unless we want to turn `Person` into an abstract class.

Then (lines 17–20) comes the slots definition. No mystery here. Note that the second template parameter must be the embedding frame name (here `Person`).

If the frame has methods (other than daemon related ones), they could be declared at this point (line 23).

Constructors

After the slots definition, the slot dependent part conclude with a bunch of four constructors. As already indicated (§2.5.2) they come by pair: one for constructing and object of exact type `Person`, the other for relaying a construction of a derived class object.

The first pair of constructors (lines 32 and 43) may be termed "regular" constructors. Only the first one is supposed to be called from outside the frame system, hence its public status. The arguments indicated are minimal. They can be supplemented by (trailing) application-dependent parameters.

Note that both constructors have the same structure:

- A *member initialization list* consisting of
  - one line for the base class, calling the relay constructor;
  - one line for each defined slot, choosing one of the slot constructors (§2.3.3).
- An empty body, although a application may fill it with some specific code.

A note on lines 36 and 47: we suppose here that `age_range` is a global object of type `Range` defined somewhere in the program; for instance

```
Interval<int> age_range(0, 150);
```

The second pair of constructors take a first parameter of type `Person`. So they look like copy constructors. Indeed, they are needed to implement clone (line 103). They are very similar to the first pair, with one important difference: they no longer depend on the application will and wish, but only on the list of slots.

2.6 Using frames

2.6.1. Creating and manipulating frames globally

Frame classes may be used as any C++ class. One can define frame objects, apply methods to them, pass them as parameters, return them from functions, and so on.

However recall that frames are not supposed to have value semantics. The only copy operation which is allowed is cloning (`clone`). Of course an application may define copy operators for frames (copy constructor, assignment). This is under its own responsibility.

The absence of (default) copy operations implies that frame are not first class citizens for C++. *In particular, it is unwise to manipulate frames through anything else than a pointer or a reference. As it is unwise to create a frame without using the* `new` *operator*. Class `Frame` itself will certainly assume in the future that all frames are allocated onto the heap.

2.6.2. Accessing slots

As already mentioned (§1.3), frame slots can be manipulated almost as if they were regular C++ class data members. For instance, the following is correct and convenient:

```
Person somebody;
somebody.age = 17;      // Slot<int, Person>::operator=
cout << somebody.age;   // conversion to int: operator T
```

Using `get_value` and `set_value` is much more awkward:

```
somebody.age.set_value(17);
cout << somebody.age.get_value();
```

However, the world is not perfect! Nor is the identification between type `T` and type `Slot<T, F>`. The problem arises for `ClassSlot<T, F>` and `FrameSlot<T, F>` when one tries to call a `T` method. For instance:

```
int n = somebody.family_name.length();
```

This does not work! The error message from **g++** is something like

```
no member function `ClassSlot<String,Person>::length()'
defined
```

The reason is simple: for various reasons, C++ refuses to perform implicit conversions onto the target parameter of a method call. So slot `family_name` is not automatically converted to `String`. Of course, the user may use `get_value`

```
int n = somebody.family_name.get_value().length();
```

or even uglier use cast, like

```
int n = ((const String&)somebody.family_name).length()
```

or, to make it "modern" C++

```
int n = static_cast<const
String&>(somebody.family_name).length();
```

The following syntaxes are also acceptable but, since they involve a full construction of a `String` object, they are less efficient:

```
int n =
static_cast<String>(somebody.family_name).length();
int n = ((String)somebody.family_name).length();
```

Of course, all these forms are horrible. This is why we provide `operator()` for slots (§2.3.3). The syntax is much nicer, since a slot can now be considered as a method, an accessor to its value:

```
int n = somebody.family_name().length();
```

3 Implementation Notes

3.1 Implementation of daemons (`if_needed`)

The implementation of" if needed" daemon is one interesting point in FRAMELIB. The constraints where the following:

- The daemon is triggered within a slot, thus it has to be a slot attribute;
- The daemon must be executed within the context of the frame the daemon belongs to, thus it has to be a member-function of this frame;
- The daemon may be specialized in classes derived from this frame, thus it has to be a virtual function.

The first two points may seem somewhat contradictory, at first glance. In fact using a pointer to member as explained in §2.3.2 helps solving the contradiction. Of course, we need to have a frame to invoke the function pointed to; but each slot knows its embedding frame (its `_context`, §2.3.2). The third point is satisfied also, owing to the fact that, in C++, access through a pointer to member always implies *dynamic binding*.

3.2 Implementation of `is_kind_of`

Another annoying problem was the absence of a general `is_kind_of` operator in C++. Of course, we have weak forms of this operator using `dynamic_cast`; the expression

```
dynamic_cast<A *>(p)
```

is non-null if, and only if, `p` points to an object of type `A` or derived from `A`, that is iff `p` points to *a kind of* `A`, or iff the type of `*p` is a subtype of `A`. Unfortunately, class `A` must be *statically* defined; even with RTTI `enabled`, it is impossible to write (although it would be simple to implement!):

```
dynamic_cast<typeid(*p1)>(p)
```

to check whether `p` points to the sort of object pointed to by `p1`. To circumvent the difficulty, we proceed in two steps.

First a general template function `is_kind_of` is defined in `common_defs.H`:

```
template <class Deriv, class Base>
inline bool is_kind_of(const Deriv& d, const Base& b)
{
     return dynamic_cast<const Base *>(&d) != 0;
}
```

This function is automatically instantiated when it is called; for instance in

```
A& a = ...;
B& b = ...;
if (is_kind_of(a, b))...
```

the function `is_kind_of<A, B>` gets instantiated; the result is `true` iff `a` is a kind of `B` that is iff the class of `a` (which is `A` or derived from `A`) is a subtype of `B`.

Second class Frame and each of its descendants (represented by F in the following), defines the following two virtual functions:

```
virtual bool F::is_super_of(const Frame& f) const
{
     return ::is_kind_of(f, *this);
}

virtual bool F::is_kind_of(const String& cln) const
{
     const Frame *pf = exemplar_of(cln); \
     return pf == 0 ? false : pf->is_super_of(*this); \
}
```

The first one, when called, provokes the instantiation of `is_kind_of<F, Frame>` which returns `true` iff parameter `f` has a type which is a subtype of `F`, or say the other way round, iff the current frame `F` is a supertype of the type of `f`.

To really implement `is_kind_of` as a method of `F`, we need to reverse the process, hence the second function. Parameter `cln` is a frame class name (§2.2.2). The function `exemplar_of` retrieves an object of the corresponding type to which we may apply the `is_super_of` method. So the `is_kind_of` method returns `true` iff the current frame `F` is a subtype of the one the class name of which is `cln`.

Note that the code of these two functions is absolutely the same, whichever frame it is part of. However, it *must* be replicated in any class derived from `Frame`. Indeed the type of `*this` triggers the automatic template instantiation (at compile-time) of the (two parameters) function `is_kind_of`.

Of course these are really tricks. Although we handle type expressions, we have to use objects as a relay. The flaw is that RTTI is not powerful enough as a meta-mechanism to deal dynamically with types.

3.3 Virtual construction of frames

Exemplars are the key elements for scaffolding a *virtual construction* mechanism for frames (Coplien's style*). This mechanism plays an important role for supporting *persistency* (saving/restoring frames).

* See "*Advanced C++: Programming Style and Idioms*", by James O. Coplien, Addison-Wesley, 1992.

These mechanisms are not yet fully available in V1.0.

4 Future Work

- Destruction of frames is not supported yet.
- Save/restore of frames is to be done. Since frames may be shared between several slots, we must carefully save a given frame only once.
  - The extension of class `Frame` should be represented in an optimized fashion to facilitate searching, saving and restoring. At this time, the extension is just an `SPset<Frame>`.
- The support library should be improved.
  - Class `String` should be optimized and, in particular, should use “copy on write” to make copying and returning from function efficient.
  - Class `SSet` and `SPSet` should be real classes, with efficient searching and not a simple specializations of the `SList` and `SPList` classes.
- At this time, it is possible to redefine a slot daemon in a subframe, but not the whole slot itself. This will be considered for further releases.
- However the structure of the first version of FrameLib is to be revisited completely since we intend to perform code generation using a Meta Object Protocol (MOP). This will have the advantage of loosening the dependency between the YAKL parser and FrameLib itself. Also, we need to handle metaobjects representing the type of objects at run time in a much more powerful way than RTTI proposes. Here again a MOP appears to be useful. The MOP we are considering relies on Open C++ v2.0.

A Support library

FRAMELIB uses its own library of fundamental data structures. At this time, only a preliminary version only is available, containing character strings, lists and sets, and dictionaries (symbol tables).

A.1 Common definitions (`common_defs.H`)

The file `common_defs.H` is included by all files in FRAMELIB (and its support library). It includes all needed standard header files (`typeinfo`, `exception`, `iostream`, `assert`...).

A.1.1. Exception handling (`Exception.H`)

File `common_defs.H` includes the file `Exception.H` which defines an exception class similar to ANSI C++ one (including a method `what`). The following macro helps in defining exceptions in new classes.

`DCL_EXCEPTION(cl, en, base)`
: Creates the class `en`, deriving from class `base`, and nested within class `cl`. Class `base` itself should be `Exception` or derived from it. The message returned by `what` will be something like `"cl::en"`. Thus, in

```
class String
{
     DCL_EXCEPTION(String, all, Exception);
     DCL_EXCEPTION(String, out_of_range, all);
     ...
};
```

the two exception classes `String::all`, and `String::out_of_range` are defined. Class `all` is a base class of `out_of_range`; `all` derives from `Exception`. For the first class, the `what` method returns `"String::out_of_range"`.

The previous pattern is used consistently throughout the library: each class with exception defines a nested class `all` which is the base class for all the specific exceptions thrown by the embedding class. This makes it possible to catch either a specific exception, or any exception thrown by a given class, as in this continuation of the previous example:

```
try {
     // do something with String's
}
catch (String::out_of_range) {
     // specific action for bad indexing
}
catch (String::all) {
```

```
        // trap any (other) exception thrown by String
    }
```

Remember that the order of `catch`'s *is* significant.

A.1.2. Automatic objects (`Auto.H`)

When manipulating exceptions, one must be very careful about automatic cleaning up: it is guaranteed only for *automatic* (i.e. stack allocated) objects. The file `Auto.H`, which is included from `common_defs.H`, helps turn some kinds of objects into automatic ones. It contains the definition of two classes:

- `Auto_Ptr<T>`, a template class for dynamically allocated objects, that is objects allocated by `new`;
- `Auto_File` for automatic closing of files.

An `Auto_Ptr` object is created from a pointer to `T`; this pointer *must* be the result of a call to `new T`. The two operators -> and * are defined to allow access to the object pointed to. The accessor ptr returns the value of the pointer itself.

```
Auto_Ptr<String> ps = new String("hello");
*ps = “bonjour";
cout << Ps->length();
String *p = ps.ptr();
```

The class `Auto_File` turns a `FILE *` obtained by `fopen` into an automatically closed object. An `Auto_File` object is just usable as a `FILE` pointer:

```
Auto_File fp("foo", "r");// call fopen("foo", "r")
int buf[10];
fread(buf, sizeof(int), 10, fp);// read 10 int’s into buf
```

A.2 Class `String` (`String.H`)

This is a very simple class encapsulating character strings. It provides full value semantics (copy constructor and assignment), concatenation (operators `+` and `+=`), relational operators, indexing (operator `[]`), substring operator (operator `()`), and usual iostream operators (`<<` and `>>`).

A.2.1. Public members of `String`

```
String(const char * = 0)
String(char)
~String()
```

Regular constructors and destructor. The first one, used as the default constructor, builds the empty string.

```
operator const char *() const
String(const Sub_String&)
```

Conversions from `String` to regular C string and from

`Sub_String` to `String`. See below for an explanation about `Sub_String`.

```
String(const String&)
String& operator=(const String&)
String& operator+=(const String&)
```
Copy operations with full value semantics.

```
int length() const
bool is_empty() const
```
Accessors to the length of the string (not including the terminating null character).

```
friend String operator+(const String&, const String&)
```
Concatenation operator.

```
friend bool operator==(const String&, const String&)
friend bool operator!=(const String&, const String&)
friend bool operator<(const String&, const String&)
friend bool operator<=(const String&, const String&)
friend bool operator>(const String&, const String&)
friend bool operator>=(const String&, const String&
```
Relational operators.

```
char& operator[](int)
char operator[](int) const
```
Indexing. The first operator is for variable strings. It may be used as the lefthand side of an assignment. The second is for constant strings and is forbidden as the lefthand side of an assignment.

```
Sub_String operator()(int, int = -1)
const Sub_String operator()(int, int = -1) const
```
Substring operators. If `s` is a `String`, `s(i, j)` returns the substring of `s` from index `i` to `j`, inclusively. If the second parameter is omitted, take up to the end of string `s`. The first operator is for variable strings. It may be used in the left hand side of an assignment; the string is then modified in place:

```
String s = "0123456789";
String s1 = s(3, 5);    // s1=="345”
s(2, 7) = “hello, kids"; // s=="01hello, kids89"
```
The second operator is for constant strings and is forbidden as an lvalue. A `Sub_String` may be used wherever a `String` may.

```
friend ostream& operator<<(ostream&, const String&)
friend istream& operator>>(istream&, String&)
```
Usual iostream operators.

```
void fput(FILE *fp) const
void fget(FILE *fp)
```
These functions are temporary. They help to circumvent a **g++ 2.7** bug

which prevents from using a file stream when RTTI is active.

```
void save(FILE *fp) const
static String *restore(FILE *fp)
```

Persistency support.

A.2.2. Exceptions of class `String`

`String::out_of_range`

Bad indexing or substringing.

`String::bad_io`

Bad IO operation in `fput`, `fget`, `save`, or `restore`.

A.3 Container classes

The FRAMELIB support library provides generic (template) classes for lists (sequence of elements or of pointers to elements) and sets (unordered collection of elements or pointers to elements). It also provides a generic symbol table (that is (string, value) pairs).

A.3.1. Iterators

Each of these container class, say *C*<*Elem*> where `Elem` denotes the type of the elements in the collection, has an associated (passive) iterator class, named *C*`_Iter`<*Elem*>. A *C*`_Iter`<*Elem*> object is constructed from a `C`<*Elem*> object; operator `()` returns a reference (or a pointer if `C`<*Elem*> is a collection of pointers) to the current element in the `C`<*Elem*> object *and* advance to the next element; method `is_at_end` returns `true` if there is no more element in the *C*<*Elem*> object. Thus with an `SList` collection (a simple sequence of elements, see below):

```
SList<int> l;               // a list of int
...
SList_Iter<int> it(l);      // an iterator to traverse l
while (!it.is_at_end())     // full traversal of l
{
     int current = it();    // take current and advance
     cout << current << ' ';
}
```

Finally, method `reset` allows to set the iterator back onto the “first” element of the collection.

All iterators may possibly throw the `access_beyond_end` exception (more precisely `C_Iter<Elem>::access_beyond_end`) if one attempts to use operator `()` beyond the last element of the collection.

A.3.2. Generic lists (`SList.H` and `SPList.H`)

`Slist<Elem>` is a simple list of objects of type `Elem`. `Elem` is not supposed to be a pointer type, and must have full value semantics (copy operations must be defined). Elements are copied by value into the list itself, thus they cannot be shared between several container classes.

```
SList(void)
~SList(void)
```
The (default) constructor builds the empty list; the destructor deletes the whole list.

```
SList(const SList&)
const SList& operator=(SList&)
```
Copy operations; the right hand side list is duplicated, thus any element is duplicated as well.

```
bool is_empty(void) const
int length(void) const
```
Accessors to the number of elements in the list.

```
Elem *search(const Elem&) const
```
Search a (copy of the) given element in the list; return 0 if not found.

```
void append(const Elem&)
void prepend(const Elem&)
```
Insert (a copy of) an element at the end (`append`) or the beginning (`prepend`) of the list.

```
Elem get(void)
```
Return a copy of the first element of the list *and* remove this first element from the list.

```
bool del(const Elem&)
```
Destroy a (copy of the) given element from the list; if the element is not found, the list is unchanged and the function returns `false`; otherwise it returns `true`.

```
friend ostream& operator<<(ostream&, const SList<Elem>&)
void save(FILE *fp) const
static SList<Elem> *restore(FILE *fp)
```
Usual output operator and persistency support functions.

Class `SPList<Elem>` is very similar to `SList<Elem>`, except that pointers to `Elem` are stored in the list, not `Elem` objects themselves. This makes sharing possible, but *no* direct support for controlling it is provided. The methods are quite similar to `SList`, with slight variations.

```
SPList(void)
~SPList(void)
```
The (default) constructor builds the empty list; the destructor delete the

whole list.

`SPList(const SPList&)`
`const SPList& operator=(SPList&)`
Copy operations; only the pointers are duplicated, thus the elements are shared between the two lists.

`bool is_empty(void) const`
`int length(void) const`
Accessors to the number of elements in the list.

`void append(Elem *)`
`void prepend(Elem*)`
Insert an element at the end (`append`) or the beginning (`prepend`) of the list.

`bool search(Elem *v) const`
Search the given pointer in the list; return 0 if not found.

`friend ostream& operator<<(ostream&, const SPList<Elem>&)`
Usual output operator; print the objects pointed to, not the pointers values.

A.3.3. Generic sets (`SSet.H` and `SPSet.H`)

`SSet<Elem>` is a simple (unordered, without redundancy) set of object of type `Elem`. `Elem` is not supposed to be a pointer type, and must have full value semantics (copy operations must be defined). Elements are copied by value into the set itself, thus they cannot be shared between several container classes. The default constructor build the empty set.

`bool contains(const Elem& e) const`
Return `true` if (a copy of) the given element is in the set.

`void add(const Elem& e)`
Add (a copy of) the element into the set, if it is not present yet.

`int card(void) const`
Return the cardinality (number of elements) of the set.

`int is_empty() const`
Return `true` if the set is empty.

`friend ostream& operator<<(ostream&s, const SSet<Elem>&)`
Usual output operator.

Class `SPSett<Elem>` is very similar to `SSet<Elem>`, except that pointers to `Elem` are stored in the set, not `Elem` objects themselves. This makes sharing possible, but *no* direct support for controlling it is provided. The methods are quite similar to `SSet`, with slight variations.

`bool contains(Elem* e) const`
Return `true` if the pointer value can be found in the set.

`void add(Elem *e)`
: Add the pointer into the set, if it is not present yet.

`int card(void) const`
: Return the cardinality (number of elements) of the set.

`int is_empty()const`
: Return `true` if the set is empty.

`friend ostream& operator<<(ostream&, const SPSet<Elem>&p)`
: Usual output operator; print the objects pointed to, not the pointers values.

A.3.4. Generic symbol tables or dictionaries (`Symbol_Table.H`)

`Symbol_Table<Symbol>` is a dictionary of `Symbol`'s. `Symbol` may be any type provided that it has *full value semantics* (copy operations must be defined), that a method

```
String Symbol::name(void) const
```

has been defined, and that operator `<<` exists for symbols. The `String` returned by method `name` is used for searching, inserting, and retrieving in the table. There are no redundancy (it is impossible for the same name to appear more than once). The table itself has no value semantics (no copy). The `Symbol`'s are copied (by value) into the table.

```
Symbol_Table(int = HASH_SIZE)
Symbol_Table(int, const Symbol[], int = HASH_SIZE)
~Symbol_Table(void);
```
: The first constructor builds an empty table. The second turns a regular C array of `Symbol` into a `Symbol_Table`. Elements are copied (by value). The default value of `HASH_SIZE` is 1009, enough for a table upto 1000-2000 symbols.

`int nb_symbols(void) const`
: Return the number of `Symbol`'s currently in the table.

`bool operator()(const String&)`
: Search for the symbol with the given `String` as `name`; return `true` if found.

`Symbol& operator[](const String&)`
: If a symbol with the given `String` as `name` is in the table, return a reference to it. Otherwise insert a new symbol with this name and return a reference to it. Thus symbol tables are a sort of associative array in the manner of AWK.

```
void del(const String&)
void remove(const String& s)
```
: Both function remove the symbol with the given `String` as `name`

from the table; if the symbol is not in the table, do nothing.

```
friend ostream& operator<<(ostream&,
       const Symbol_Table<Symbol>&)
```

Print out the whole table.

```
void save(FILE *fp) const
static Symbol_Table *restore(FILE *,
       Symbol_Table<Symbol> * = 0)
```

Persistency support. If the second argument of `restore` is present, it must point to an existing `Symbol_Table` (not necessarily empty) , which is filled with the symbols read from the input. Otherwise, a new `Symbol_Table` is allocated within `restore`.

B Template instantiation

This is really a serious problem with present C++ compilers, since no portable scheme has been defined so far and the fourth coming ISO C++ standard does not addresses this issue. Each compiler has its own way to support–or not to support–automatic instantiation of templates. GNU **g++** decided (at least as of version 2.7.x) not to support automatic instantiation. Thus the programmer *must* perform *explicit* template instantiation.

We decided to use the possibility offered by **g++** of having external templates (compilation flag `-fexternal-templates`). This requires that we use the `#pragma interface` and `#pragma implementation` facilities*.

Let us describe how we perform the template instantiation** using class `SPSet` as an example. Suppose that in the whole application we have to instantiate `SPSet<String>`, `SPSet<int>`, and `SPSet<Person>`. Since the files `SSet.H` and `SSet.C` which contain the definition of class `SSet` both include the line

```
#pragma interface
```

no code will be generated unless we do the instantiation explicitly in a file containing a pragma implementation. To do this, we introduce file `SPSet_impl.C`:

```
// File SPSet_impl.C: implementation of class SPSet
//--------------------------------------------------
#pragma implementation “SPSet.H”
#include “SPSet.H”

// Explicit intantiation for class String
```

* “Facility” is ironic here!

** This is one among many possible schemes. We suggest it because it seems simple and efficient.

```
#include “String.H”
template class SPSet<String>;
template class SPSet_Iter<String>;
template ostream& operator<<(ostream&, const
SPSet<String>&);

// Explicit intantiation for int
template class SPSet<int>;
template class SPSet_Iter<int>;
template ostream& operator<<(ostream&, const
SPSet<int>&);

// Explicit intantiation for class Person
#include “Person.H”
template class SPSet<Person>;
template class SPSet_Iter<Person>;
template ostream& operator<<(ostream&, const
SPSet<Person>&);

// Other explicit instantiations of SPSet, SPSet_Iter...
```

Note that friend functions (here `operator<<`) must be instantiated separately from the class itself. Also, iterator classes usually have to be instantiated as well.

Of course the file `SPSet_impl.C` must be compiled and linked with the other files constituting the application. But, here this is not enough. The linker will complain about several functions of `SPList` not being defined. The reason is that `SPSet` uses `SPList` for its implementation. Thus we have to introduce also (if not done already) the file `SPList_impl.H` with a contents very similar to `SPSet_impl.C`:

```
// File SPList_impl.C: implementation of class SPList
//-------------------------------------------------
#pragma implementation “SPList.H”
#include “SPList.H”

// Explicit intantiation for class String
#include “String.H”
template class SPList<String>;
template class SPList_Iter<String>;
template ostream& operator<<(ostream&, const
SPList<String>&);

// Explicit intantiation for int
template class SPList<int>;
template class SPList_Iter<int>;
template ostream& operator<<(ostream&, const
SPList<int>&);
```

```
// Explicit intantiation for class Person
#include "Person.H"
template class SPList<Person>;
template class SPList_Iter<Person>;
template ostream& operator<<(ostream&, const
SPList<Person>&);

// Other explicit instantiations of SPList,
SPList_Iter...
```

Needless to say that the same construction must be applied to all template classes used either by the library or the application. For instance, files like `Slot_impl.C` and `Range_impl.C` will certainly be mandatory.

C File and Directory Structure of the Frame Library

`common.mk`

Common definitions for Makefiles in subdirectories. Usable with SUN **make**.

`include_support`

A subdirectory containing common definitions, support library classes specifications, and support template classes implementations. This directory should be in the search path of C++ include's.

`src_support`

A subdirectory containing source files for non-template support classes (at this time, only `String`).

`test_support`

A subdirectory containing source files for testing individually the support classes. The Makefile here builds all the tests.

`frames`

A subdirectory containing the source of the frame system, together with a test example. The Makefile here builds the test (`tst_Frame`).

`frames/doc`

A sub-subdirectory containing the documentation for the frame library (this document) (FrameMaker format).

Index

Symbols

A

B

C

D

L

M

N

O

P

R

S

T

U

V

W

Y

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399