FrameKit, an Ada Framework for a Fast Implementation of CASE Environments

Fabrice Kordon & Jean-Luc Mounier,
LIP6-SRC
Université P.&M. Curie
4 place Jussieu, 75252 Paris Cedex 05, France
E-mail: Fabrice.Kordon@lip6.fr, Jean-Luc.Mounier@lip6.fr

Abstract : *Software engineering methodologies rely on various and complex graphical representations and are more useful when associated to CASE (Computer Aided Software Engineering) tools designed to take care of constraints that have to be respected. Now, CASE tools gave way to CASE environments (a set of tools that have a strong coherence in their us). This concept provides enhanced solutions for software reusability while the environment may be adapted to a specific understanding of a design methodology.*
This paper describes FrameKit, an Ada based framework dedicated to the quick implementation of CASE environments. We summarize first the concepts implemented in FrameKit and illustrate them using a detailed example of a simple tool implementation and integration.
Key word: *Generic CASE, Software platform, Tool integration, Software Engineering, quick implementation*

1. Introduction

Software engineering methodologies rely on various and complex graphical representations such as SA-RT, OMT, UML etc. They are more useful when associated to CASE (Computer Aided Software Engineering) tools designed to take care of constraints that have to be respected. Such tools help engineers and facilitate the promotion of such methodologies.

Now, CASE tools gave way to CASE environments which may be adapted to a specific understanding of a design methodology. A CASE environment can be defined as follows **[17]** : it is a set of tools that have a strong coherence in their use. This concept provides enhanced solutions for software reusability. CASE environment are built on a platform that allows tool plugging. Communication and cooperation between tools must subsequently be investigated.

The implementation of CASE environments is a complex task because they need various functions like a graphical user interface, database facilities and, of course, the operations that are related to the methodology they implement (compilation of specifications, animation/simulation of specifications, code generation from specification, etc.).

Even early platforms offer solutions for tool reuse and cooperation. One of the first one, APSE **[2]** is mostly data oriented and dedicated to Ada development. ESF **[6]** and HP-Softbench **[8]** suggest a communication oriented architecture. ISTAR **[4]** proposes a strong "process orientation" based on a contract concept defining inputs, outputs and constraints. Then, some standards like ECMA **[5]** and then CORBA **[15]** provide a complete architecture model that identifies required services and considers discrete dimensions of cooperation between tools and a hosting platform (usually data, control and presentation).

Experimentation over large projects have outlined the difficulty to maintain such environment, especially when tools come from various origin. In a project like Ptolemy **[16]**, the software basis for the project have largely changed in order to ease maintenance as well as new development. Such work (in particular, the Tycho interface system **[9]**) take into account the definition of evolutionary interfaces between major components.

This paper describes FrameKit **[10, 11]**, a framework dedicated to the quick implementation of CASE environments. FrameKit is parameterized in order to provide a framework for the customization of CASE environments dedicated to a given method (Figure 1). FrameKit is mostly integrated in Ada (a small amount of C is used for the interface with Unix) and provides enhanced Ada Application Program Interfaces (API) to operate a light but efficient customization procedure.

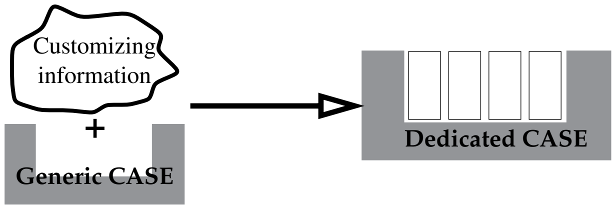


Figure 1 : *From a Generic CASE to a dedicated one.*

We present first the main lines of the FrameKit architecture (Section 2.). Then, we present principles of tool design and implementation and illustrate with a detailed example how the Ada API implement these principles.

2. Overview of the FrameKit Architecture

2.1. Structure of a CASE environment

A CASE environment is composed of several cooperative components :

- a *platform* having communication and data storage capabilities;
- a set of *tools* driven by the platform. Each one is an independent software which can run out of the environment and offers functions that may enrich it.

To achieve this enrichment, a procedure called *integration* has been defined. We distinguish two types of tool integration : a *priori* and *a posteriori*.

- the *a posteriori* integration : involved tools are already designed; source files may not be available.

The a priori integration concern tools that are especially designed to run in a CASE environment. It does not raise any major problem while the selected implementation techniques and standards are considered at the implementation stage. Platform functionalities are usually used the best way, especially when APIs (Application Program Interface) are available.

The a posteriori integration concerns already designed tools (some times, source files may not be available) to be integrated in a CASE environment. It requires an adaptation of the imported software. The complexity of such an operation depends on several criteria regarding modularity and portability of the tool : these aspects concern both its functionalities and its relation with the execution environment (file system, operating system...).

According to **[18]**, the integration procedure must take into consideration five integration axis :

- *Platform* : tools must run on a platform giving a transparent access to heterogeneous machines and to the operating system.
- *Presentation* : the user interface must be homogeneous for any tool. Window managers and look and feel style guides are useful.
- *Data* : tools have to exchange and share data.
- *Control* : tools have to cooperate, notifying events to others tools. They may also need services provided by others ones.
- *Process* : the main goal of an environment is to support development processes. Thus, it is of interest to define a technique to describe such processes.

However, the definition of these five axis are quite theoretical. It is difficult to manage them all properly. In FrameKit, we have chosen to reduce them to three :

- Presentation axis and basic aspects of process functions are grouped in a *User Interface axis*,
- Some of the Data axis defined in **[18]** are covered by *the Data management* axis ,
- Platform axis and basic control functions are grouped together in an *Environment axis*.

As a guide to both types of integration, we introduce the following notions : *Formalism*, *model* and *Service*. A Formalism describes representation rules of a knowledge domain. A model is the description of a given knowledge using a formalism; it is a «document» composed with objects defined in the formalism. A service is a tool function that correspond to operations in a design methodology. Services are related to a set of formalisms (i.e. the operation has a signification for these formalisms) and thus, can be applied on models issued form these formalisms.

The formalism notion is more related to the User Interface axis. Model notion is associated to both User Interface and data management axis. The service notion is strongly connected to the environment axis.

2.2. User Interface

In FrameKit, presentation and display of services are strongly constrained. Both types of functions are supported by Macao **[13]**, a polymorphic editor able to manipulate models after the corresponding formalism description. It provides a unified look and feel for both the manipulation of models and access to the services integrated in FrameKit.

The construction of a new formalism does not imply any recompilation of Macao. All the required information is defined in an external file that expresses possibilities of the formalism. Of course, Macao deals with syntactical aspect only, semantical ones are a convention between the user and the tool.

In FrameKit, description of formalisms is object-oriented. This allows an easy management and updating of formalisms. Each class in the formalism is either a node or an edge (interconnecting nodes) and contains a set of labels (string values, digits...) that characterizes instances of the object. Additional information (how it looks, is it a link to a «sub-level» when the formalism is hierarchic, etc.) must also be provided to fully describe the formalism.

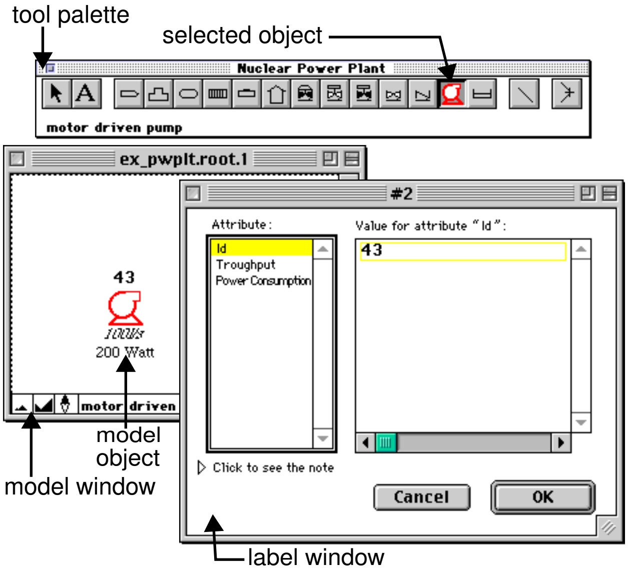


Figure 2 : *Link between formalisms and model in Macao.*

Figure 2 illustrates the relation between a formalism description and related models. Classes declared in the formalism are described in the tool palette (top of Figure 2). Users can select one of them and create instances in the model (a model object) in a window that contains a model description (bottom left in Figure 2). It is then possible to edit labels related to this objects and declared in the corresponding class description.

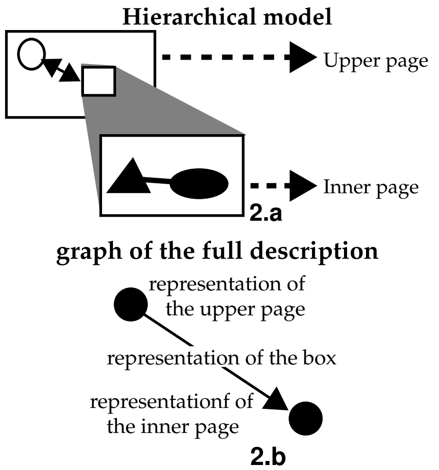


Figure 3 : *Structure of a hierarchical formalism.*

Formalisms may be composed when they are hierarchical. In that case, some nodes are associated to another formalism. These nodes (called «boxes») can be «opened» to display its content in a new page. Models are thus composed of pages; each page is a part of the model.

Figure 3 shows how it works. In Figure 3.a, an node in a page is associated to another page. The description of the model structure (Figure 3.b) is an oriented graph in which, nodes represent pages and, edges links between a box and a page. So, hierarchical graphical descriptions are described using a set of (flat) formalisms.

2.3. Data management

The data management axis deals with both data storage in a repository and data representation. To cooperate, tools use intermediate files to exchange data. However, they are usually not designed for data exchange with foreign software. Data translations must be performed : some integration techniques rely on the addition of a software layer called driver **[1]** or capsule **[7, 17]**. For communication, the use of an internal Data Definition Language (DDL) makes this translation process easier and supports heterogeneity between tools (for example, the use of discrete programming language).

A common DDL, implemented at the platform level, provides an indirect but standardized communication between tools allowing an easy maintenance of the tool set. Adding or modifying a tool needs only to update one interface between the tool and the platform. Tool maintenance is performed apart from the host platform. The tool evolution is hidden by the communication driver.

Tools need to store persistent data which may be shared. The environment has to provide a set of functions to manage such data. When the number of shared files grows, the use of a shared object database is the most interesting solution **[12]**. However, this solution is heavy to implement and we propose a simplified model that is suitable for building a simple platform like FrameKit.

FrameKit provides a model for both large grained and fine grained data :

- Large grained data are information components like models, results or any other information managed by tools (libraries, preferences etc.). FrameKit proposes discrete types of large grained data and store them using sample repository functions;
- Fine grained data are fine information components like element in the model (nodes, edges, their relations and their labels). Fined grained elements dare stored using elementary messages.

2.3.1. Large grained data

Large grained data are information components like models, results or any other information managed by tools (libraries, preferences etc.)

FrameKit types large grained data using tool-defined keys and behaviors. Tool-defined keys are keywords used to find out an information in the FrameKit repository. The platform uses this information but does not have any knowledge of the corresponding semantics. Three types of data behavior correspond to three persistency approaches :

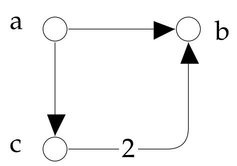
- ***model-associated*** data concern all the information associated to a model. It is useful to properly handle version management : when a model changes, associated results become obsolete and should be deleted and recomputed if needed. Such data are stored with the model description in a cell stamped by its last modification date. The cell is destroyed when the model is updated;
- ***user-associated*** data concern all the information related to a user (preferences, information potentially shared by models...). This information remains reachable until the user is deleted;
- ***global data*** concern all the information related to a CASE environment. It is stored in cells that may be associated to a tool, a formalism or to the platform itself (administration data only). Data last as long as the entity (tool, formalism or platform).

To implement these discrete behavior, a proper use of directories is sufficient. Global data is stored in a directory potentially shared by all users and tools. user associated data is stored in a user associated directory. Finally, model-associated data is located in a directory that last as long as the model does not change.

2.3.2. Fine grained data

Fine grained data are fine information components. To ease both their storage and handling, FrameKit implements a message based approach. Each element in the model (nodes, edges, their relations and their labels) are stored using elementary messages

Messages describe elementary actions like «create a new node numbered n_1 having class N», «associate nodes n_1 and n_2 by means of a connector c_1 from class C», «associate a textual label named A and having value X to node n_1» etc. This description technique is generic because it works regardless any knowledge of the corresponding formalism. the name of classes are defined using strings and instances of classes are named using integers.




```
FO(13:OrientedGraph)
VM(2,6,12)
CN(4:node,4)
CT(4:name,4,1:b)
CN(4:node,3)
CT(4:name,3,1:c)
CN(4:node,2)
CT(4:name,2,1:a)
CA(3:arc,5,2,4)
CA(3:arc,6,2,3)
CA(3:arc,7,3,4)
CT(9:valuation,7,1:2)
```

Figure 4 : *OrientedGraph model and its corresponding internal description.*

Example 1: *Let us consider a small model defined using the elementary formalism OrientedGraph (Figure 4).*
Its definition is transported using simple messages that carry out syntactic aspects only. Instruction CN create a new instance of the referenced class. CA instanciates a new connector of the referenced class. CT affects a value to labels on arcs or nodes. Please note that object instances are named using a unique object identifier provided by Macao (here, node labeled a has id 2). Tool have to use this identifier to access objects.
FO is used to identify the formalism and VM the version of this formalism. This information is used for check by tools only.

This mechanisms relies on ASCII information only, which is a way to solve most portability problems as well as exploitation of data by programs running on discrete target architectures without having to use XDR mechanisms. In fact, in FrameKit, all data are stored in ASCII format.

2.4. Environment

The environment axis supports the following points :

- association of an Operating System «command line» to a service (i.e. a given compiler is associated to the service compile and is invoked a given way);
- encapsulation of the Operating System functions like program invocation, program communication, navigation through the repository system etc.;
- definition of a diffusion model to facilitate installation and evolution of the environment.

The first point is strongly related to the management of services. It is the set of low-level mechanisms required to support services as they appear to the user.

The second point is important to support tool integration as well as tool implementation. It should be properly implemented in the APIs used to program in such an environment. Of course, a level of abstraction is necessary in order to enforce portability. This is important for multi-platform implementation and diffusion.

For example, in FrameKit, we have implemented the following functions :

- A high level communication model has been defined : several implementation are proposed (some may have restrictions). Then, any software component able to support one of these implementations should be easily integrated in FrameKit;
- A high level transmission of information by means of messages is built on top of the communication model, like the Macao widget-like mechanisms to manage interaction with users;
- A repository offers storage services. This repository hides File system related mechanisms (file naming system...).

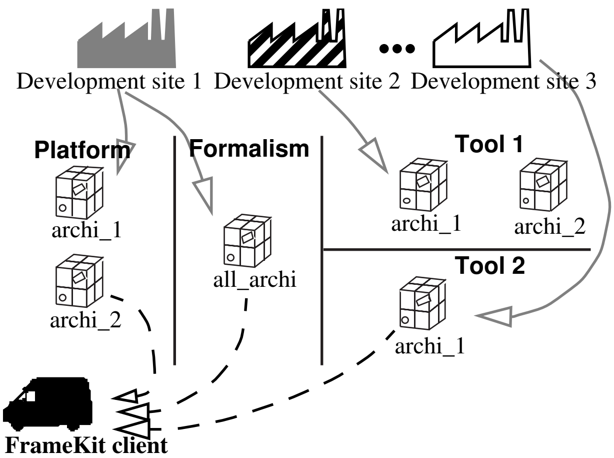


Figure 5 : *Example of the diffusion model.*

The third point is also important because it proposes a framework for the evolution of the environment. The diffusion approach we propose rely on *kits*. A kit is an elementary installation component that contains elements to be installed by a specific administration tool. There should be four types of kits :

- Platform kits contain executable and data of the environment (administration tools, communication libraries etc.),
- Formalism kits contain all the definition of a new formalism in an installed environment;
- Tool kits contain information to install new tool and its associated set of services (executable files, initial data etc.);
- Custom kits for local upgrade of any element (platform executable, tool executable etc.); it enable the construction of patches that fixes bugs of a previous distribution.

Example 2: *Figure 5 proposes an instanciation of the diffusion model we propose. Let us imagine that a software engineering environment is being developed in discrete places. Such a diffusion strategy enables :*

- *a distributed upgrade of kits (developers only upgrade kits they are responsible of),*
- *a custom installation by clients (each client picks up what he needs).*

3. Implementing Tools to Customize FrameKit

3.1. Structure of a tool designed for FrameKit

To hide target architecture related mechanisms (and meet platform integration), all presentation, data, control axis should be implemented and available for applications by means of Application Program Interfaces (API).

Thus, tool designed to run in the target environment take benefits from these APIs. To meet this requirement, three API corresponding to the three axis presented in Section 2.1. The algorithmic part of the program should be disconnected from the environment and relate with it only by means of the APIs (Figure 6).

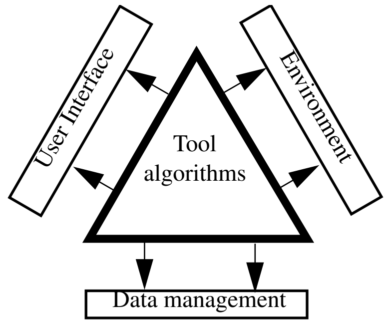


Figure 6 : *Architecture of a tool designed to run in the software environment (a priori integration).*

All implementation in FrameKit follow this strategy and even then «main» program of applications is a part of the FrameKit libraries. This enable to always correctly initialize all required resources to operate the three API's and call the «tool main program» without having to change initialization directives over the FrameKit versions. Only a new binding with APIs libraries is required. This strategy is also used for administration tools (that uses standard API but are considered as a part of the platform) as well as platform programs.

3.2. A posteriori integration in FrameKit

Tools to be a posteriori integrated in the type of environment should be disconnectable from their user interface. Discrete techniques could be considered according to the set of available information developers provide on their software.

If source code is available, it is possible to adapt it to fit the API described in the previous section. Then, the result is similar to an a priori integration. However, it should be avoided for tools for which implementation is not controlled by the integration team : the integration work has to be done when a new version is released.

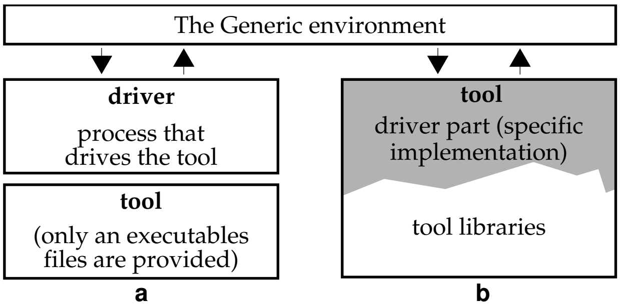


Figure 7 : *Possible architectures of a posteriori integrated tools.*

If only executable file is available (plus information about exchange formats), it is possible to drive the tool by means of a specifically implemented process (Figure 7.a). The environment only knows about this process which architecture is the one defined in Figure 6. The driver and the tool communicates by means of any mechanism encapsulated in the environment (see environment axis).

If tool libraries are provided (plus description of data structure), they can be directly linked to a driver to make a unique executable file (Figure 7.b).

In both cases, the driver translate information in the required format and then, translate back results for display by means of the user interface.

3.3. Application on a toy example

So, for both tool construction or tool integration, the implementation work is reduced to write an application using the standard FrameKit API. We propose to detail the Ada based way to implement a new tool in FrameKit.

We first describe the tool to be implemented and then describe its implementation. Source code fully provided, separated by explanation and comments. Some execution screen shots are provided to illustrate how the FrameKit environment behaves according to the corresponding stimulations.

3.3.1. Presentation of the example

Let us consider the Graph formalism. This formalism describe graphs and is composed with :

- a "node" class,
- a "edge" connector,
- a "arc" connector.

both connectors can relate nodes between them. Two labels are associated to these formalism objects (nodes, edges and arcs) : "name" and "value". Five global labels ("author(s)", "version", "information", "project" and "title") provide information about the model. Figure 8 shows how Graphs are managed by the Macao User Interface : nodes are represented by a circle, edges appear as a line and arcs look like an arrow. The formalism description takes about fifteen minutes, which is definitely shorter than designing a new graphical interface.

The tool we want to build has to check if there is a connector between two nodes designated by the user. The tool has a verbose option that ask for the user name (in order to provide him with a more convivial answer). There are two ways to invoke it :

platform mode
tool_example <*framekit_config_param*> [-verbose]
standalone mode
tool_example -s [-verbose] obj_id1 obj_id2

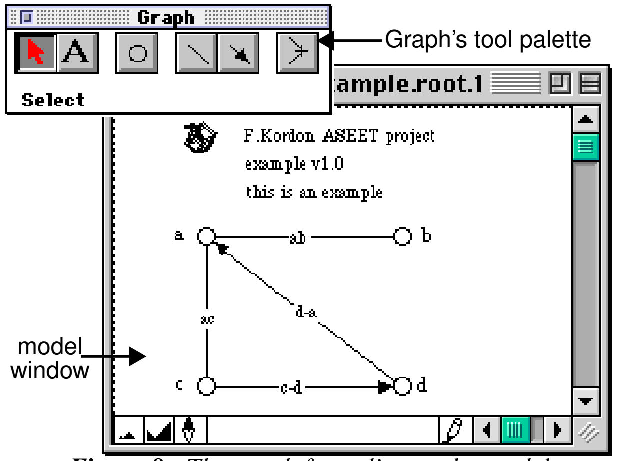


Figure 8 : *The graph formalism and a model.*

Two additional parameters are required in standalone mode because the tool runs without user interface. These parameters supply the application with Macao object identifiers.

3.3.2. The generic main provided by FrameKit

As mentioned in Section 3.1. applications' main program is a part of the FrameKit libraries. It is a generic Ada procedure for which the following generic parameters have to be provided :

- `TRACE_FILE_NAME` is a string used to generate the trace file name (if traces are displayed in the application),
- `IS_STANDALONE_SUPPORTED` is a boolean that indicates if the tool can be executed in standalone mode,
- `MAIN_ALGORITHM` is a reference to the main procedure of the application,
- `TOOL_NAME` is the tool name in the FrameKit environment,
- `TOOL_VERSION` is the tool version in the FrameKit environment,
- `TOOL_COPYRIGHT` is a tool copyright automatically displayed at launch,
- `KIT_NAME` is the name of the kit in which the tool will be integrated (used in the distribution procedure),
- `ON_LINE_HELP` is a one line help displayed when the tool crashes (i.e. a non FrameKit exception is raised),
- `FK_INTERRUPTION_FROM_IU_INIT_HANDLER_SYSTEM` is a reference to the procedure that initialize the interruption handler manager,
- `FK_INTERRUPTION_FROM_IU_HANDLER` is a reference to the procedure to invoke when an interruption is provoked from Macao.

A default value is associated to the three last parameters. They are respectively : the "no help available" string and two procedures that do nothing (it is then assumed that the service will be declared as non-interruptible).

3.3.3. Implementation of interruption handlers

Let us now define a package for managing execution interruptions. The package specification is provided herafter :

```
with FK_STRINGS, FK_API_DATA_MANAGEMENT;
use FK_STRINGS, FK_API_DATA_MANAGEMENT;
package TOOL_EXAMPLE_HANDLER is
   -- The type that defines possible actions supported by the
   -- interrupt handler.
   type HANDLER_OPERATIONS is
      (DEFAULT, -- nothing
       DISPLAY_MESSAGE); -- display of a message
   -- New primitive that allows a tool designer to change the
   -- action to perform at interrupt.
   procedure SET_HANDLER_TREATMENT
      (WHAT_TO_DO : in HANDLER_OPERATIONS);
   -- Primitives required if interruptions are supported
   procedure SET_HANDLER;
   procedure INTERRUPT_HANDLER
      (TOOL_NAME : in STRING);
end TOOL_EXAMPLE_HANDLER;
```

The type HANDLER_OPERATIONS is useful when discrete interruption treatments should be handled by the tool according to the current execution phase (for example, when the same program provides several services). The SET_HANDLER procedure set treatment to default and SET_HANDLER_TREATMENT (not required) allows to change the current treatment. We provide below the body associated to INTERRUPT_HANDLER.

```
procedure INTERRUPT_HANDLER
            (TOOL_NAME : in STRING) is
begin
   case REMIND_TREATMENT_TO_DO is
      when DEFAULT =>
         null; -- we do nothing;
      when DISPLAY_MESSAGE =>
         FK_PUT_MSG (MESSAGE  => "Canceled",
                     HISTORIC => TRUE);
   end case;
end INTERRUPT_HANDLER;
```

In that procedure, REMIND_TREATMENT_TO_DO is a global variable of package TOOL_EXAMPLE_HANDLER that select the current treatment to process by INTERRUPT_HANDLER. The primitive FK_PUT_MSG is a part of the FrameKit API and displays a message either on the current terminal (standalone mode) or in the Macao historic window.

3.3.4. Programming the tool

We now illustrate, using the implementation of the tool, the use of the API primitives to manipulate the FrameKit environment. Basically, any tool has to use the three major standard API : FK_API_DATA_MANAGEMENT handles the Data Management axis, FK_API_USER_INTERFACE that handles supports the User Interface axis and FK_API_ENVIRONMENT_COMMUNICATION supports the Environment axis (as they are both presented in Section 2.1.). There are also numerous available standard tools available (multi-language message management, lists, tree, etc.).

```
with FK_API_DATA_MANAGEMENT,
     FK_API_ENVIRONMENT_COMMUNICATION,
     FK_API_USER_INTERFACE,
     TOOL_EXAMPLE_HANDLER,
     CHAINES_VARIABLES;
use  FK_API_DATA_MANAGEMENT,
     FK_API_ENVIRONMENT_COMMUNICATION,
     FK_API_USER_INTERFACE,
     TOOL_EXAMPLE_HANDLER,
     CHAINES_VARIABLES;
procedure TOOL_EXAMPLE_BODY is
   -- Local variables to be used in the example
   VERBOSE_MODE  : BOOLEAN := FALSE;
   CRT_ARG : POSITIVE := 1; -- counting args
   SELECTED_NODE_1: FK_OBJECT_IDENTIFIER;
   SELECTED_NODE_2: FK_OBJECT_IDENTIFIER;
   NAME : VSTRING := TO_VSTRING ("user");
   MODEL : FK_MODEL_DESCRIPTION;
   PAGE : FK_PAGE_DESCRIPTION;
```

An automatic trace system can be enabled and disabled automatically using respectively FK_ENABLE_TRACE and FK_DISABLE_TRACE procedures. There are numerous trace classes defined for the platform and a set of trace classes available for tool design (in this example, KFK_TRACE_MAIN is used). The FK_PUT_IN_TRACE procedure is the one that display traces for a given trace class. Traces are strings written into a file automatically created at first need, according to the information provided in the TRACE_FILE_NAME generic main parameter (see Section 3.3.2.).

Two procedures allow to access command line argument transparently (e.g. without having to consider that platform parameters may be inserted before tool parameters) : FK_ARG_COUNT and FK_ARG_VALUE that behaves like well known Unix argc and argv.

```
begin -- for TOOL_EXAMPLE_BODY
   FK_ENABLE_TRACE (KFK_TRACE_MAIN);
   for I in 0 .. FK_ARG_COUNT - 1 loop
      FK_PUT_IN_TRACE (KFK_TRACE_MAIN,
          S => "argument number" &
                INTEGER'IMAGE (I) &  ASCII.HT &
               "= """ & FK_ARG_VALUE (I) & """");
   end loop;
   -- The environment is now correctly set and the exception
   -- handler is operational with a default action. Let us check
   -- parameters
   if FK_ARG_COUNT > CRT_ARG then
      if FK_ARG_VALUE (CRT_ARG) = "-verbose" then
         VERBOSE_MODE := TRUE;
         CRT_ARG      := CRT_ARG + 1;
      elsif FK_IS_IN_FRAME_KIT then
         -- in FrameKit, "-verbose" is the only parameter
         FK_PUT_ERROR (MESSAGE => """" &
             FK_ARG_VALUE (CRT_ARG) &
             """: bad parameter (FrameKit mode)",
             EMPHASIS => FALSE);
         -- to signal a problem
         raise FK_PROCESSED_WITH_PROBLEM;
      end if;
   end if;
```

The tool we design requires to work on designated objects. These objects are provided by the Macao user Interface, we have thus to consider two discrete ways to extract them : using the standard API when the tool runs under FrameKit and via the command line when the tool runs in standalone mode (this function cannot be emulated).

The FK_IS_MODE_STANDALONE function allows us to know in which mode we are running. Then, objects identifiers are converted from the values extracted in the command-line. If less than two parameters are provided, then an error is displayed using the FK_PUT_ERROR_MESSAGE primitive and the exception FK_PROCESSED_WITH_PROBLEM is raised. This exception is a communication standard in FrameKit to provokes a premature exit. This exception is caught at the main level and information is provided to the environment to signal that execution was aborted.

```
   -- Get designated objects (on the command line in standalone,
   -- using FrameKit otherwise)
   if FK_IS_MODE_STANDALONE then
      -- we run in standalone mode
      if FK_ARG_COUNT >= CRT_ARG + 1 then
         begin
            SELECTED_NODE_1 :=
               FK_STRING_TO_OBJECT_ID
```

```
            (FK_ARG_VALUE (CRT_ARG));
         SELECTED_NODE_2 :=
            FK_STRING_TO_OBJECT_ID
               (FK_ARG_VALUE (CRT_ARG + 1));
      exception
         when CONSTRAINT_ERROR =>
            FK_PUT_ERROR (MESSAGE =>
               "Big argument problem",
               EMPHASIS => FALSE);
            raise FK_PROCESSED_WITH_PROBLEM;
      end;
   else
      -- two nodes should be identified
      FK_PUT_ERROR (MESSAGE =>
         "Two objects required",
         EMPHASIS => FALSE);
      raise FK_PROCESSED_WITH_PROBLEM;
   end if;
else
```

In FrameKit mode, it is simpler because designated object identifiers are directly provided using the FK_GET_A_DESIGNATED_OBJECT primitive. Tests are also simpler because Macao forbid the service execution as long as no object is designated. We thus only have to check that at least two objects have been transmitted (information provided by FK_GET_NUMBER_OF_DESIGNATED_OBJECTS).

```
   -- running in standalone mode, Macao does not invoke
   -- the service if no object is selected
   if FK_GET_NUMBER_OF_DESIGNATED_OBJECTS < 2
   then
      FK_PUT_ERROR (MESSAGE =>
         "Two objects requires",
         EMPHASIS => FALSE);
      raise FK_PROCESSED_WITH_PROBLEM;
   elsif FK_GET_NUMBER_OF_DESIGNATED_OBJECTS
         > 2 then
      FK_PUT_WARNING (MESSAGE =>
         "extra objets are discarded",
         EMPHASIS => FALSE);
   end if;
   SELECTED_NODE_1 :=
      FK_GET_A_DESIGNATED_OBJECT (1);
   SELECTED_NODE_2 :=
      FK_GET_A_DESIGNATED_OBJECT (2);
end if;
```

At that stage, the we can work on the model. Let us notice that interruption is checked periodically to reduce the hypothesises on the compiler implementation (it causes an I/O in the FrameKit software bus). It is also a way to set when interruptions can be supported by the tool. Here, we consider that the tool enter in a phase for which the display_message treatment is associated to interruptions.

```
-- check for interruption signal
FK_CHECK_FOR_USER_INTERRUPTION;
-- let us consider that we provide a message when execution
-- is canceled
SET_HANDLER_TREATMENT
   (WHAT_TO_DO => DISPLAY_MESSAGE);
```

The verbose mode introduced in our example provides a good illustration of user interaction. The primitive FK_GET_A_LINE ask for a one line text via the User Interface (a specific dialog window is created to input the line). Other primitives allow to handle multi-line texts, item selection within a list and standard dialog boxes. All I/O in FrameKit are handled by the User interface. Tools activate these functions using vidget like messages. In standalone mode, an ASCII emulation is provided.

```
-- preparation of verbose mode
if VERBOSE_MODE then
   declare
      CONTINUE : BOOLEAN;
   begin
   -- Get the user name
      FK_GET_A_LINE (MSG =>
         "Please give me your name",
         RESULT => NAME,
         NO_CANCEL => CONTINUE,
         TITLE => "Demand");
      if not CONTINUE then
         -- user has clicked on CANCEL. We retaure the default
         -- value
         NAME := NAME;
      end if;
   end;
end if;
```

Models are stored in FrameKit by means of elementary messages, as described in Section 2.3.2. These messages are handled via an API in order to avoid a direct manipulation of these messages. Acquisition of the description is performed via the FK_ACQUIRE_MODEL_FROM_DISK primitive. Then , the user may access separately to pages (in this example, there is only one root page because the formalism is flat). When a page description is loaded, access to objects is performed either one by one (via their position) or using their internal identifier.

When an object description is extracted, three primitives allow to get information :

- FK_GET_ENTITY_INFORMATION provides all information on the object : its class, its type (is it a node or a connector) and its identifier,
- FK_GET_ARC_ENDS is dedicated to connectors description and provides the Macao identifier for the two connected arcs,
- FK_GET_ATTRIBUTE_VALUE (not invoked in our example) provides the value associated to a label as a string list.

In the source code below, we check if designated object are of the appropriate class (the user has to select nodes).

```
-- acquire model (flat = one page only)
FK_ACQUIRE_MODEL_FROM_DISK (MODEL);
FK_GET_ROOT_PAGE (MODEL, PAGE);
-- acquire object description and check objects validity
declare
   DSC_OBJ    : FK_LST_CAMI_MSG;
   NODE_CLASS : VSTRING;
   ENT_TYP    : FK_OBJECT_CLASSIFICATION;
   ENT_ID     : FK_OBJECT_IDENTIFIER;
begin
   DSC_OBJ := FK_GET_OBJ_DESC
       (PAGE_DESC => PAGE,
        ID_OBJ => SELECTED_NODE_1);
   FK_GET_ENTITY_INFORMATION
       (FULL_DSC => DSC_OBJ,
        ENTITY_CLASS => NODE_CLASS,
        ENTITY_TYP => ENT_TYP,
        ENTITY_ID => ENT_ID);
   if ENT_TYP /= KFK_NODE and then
      TO_STRING (NODE_CLASS) /= "node" then
      FK_PUT_ERROR (MESSAGE =>
          "First object is not a node",
          EMPHASIS => FALSE);
      raise FK_PROCESSED_WITH_PROBLEM;
   end if;
   DSC_OBJ := FK_GET_OBJ_DESC
        (PAGE_DESC => PAGE,
         ID_OBJ => SELECTED_NODE_2);
   FK_GET_ENTITY_INFORMATION
        (FULL_DSC => DSC_OBJ,
         ENTITY_CLASS => NODE_CLASS,
         ENTITY_TYP => ENT_TYP,
         ENTITY_ID => ENT_ID);
   if ENT_TYP /= KFK_NODE and then
      TO_STRING (NODE_CLASS) /= "node" then
      FK_PUT_ERROR (MESSAGE =>
          "Second object is not a node",
          EMPHASIS => FALSE);
```

```
            raise FK_PROCESSED_WITH_PROBLEM;
         end if;
      end;
```

In the source code below, we parse all connectors in the description and check their ends. Function FK_GET_NB_CONNECTORS returns the number of connectors located in a page (there is a similar function for nodes). Primitive FK_GET_CONNECTOR_N_DSC extract the Nth connector description.

Note that primitive FK_PUT_MSG used when connector search has been sucessful displays a message associated to the object. In fact, objects will be designated in the User Interface when the message appear.

```
      -- Searching for the arc that relates these two nodes
      declare
         ARCS : NATURAL :=
                FK_GET_NB_CONNECTORS (PAGE);
         DSC_ARC       : FK_LST_CAMI_MSG;
         STARTN, ENDN  : FK_OBJECT_IDENTIFIER;
         ARC_CLASS     : VSTRING;
         ENT_TYP       : FK_OBJECT_CLASSIFICATION;
         ARC_ID        : FK_OBJECT_IDENTIFIER;
         FOUND         : BOOLEAN := FALSE;
      begin
         while ARCS > 0 loop
            DSC_ARC := FK_GET_CONNECTOR_N_DSC
                (PAGE_DESC => PAGE, POS => ARCS);
            FK_GET_ARC_ENDS (DSC_ARC, STARTN, ENDN);
            if (STARTN = SELECTED_NODE_1 and
                ENDN = SELECTED_NODE_2) or
               (ENDN = SELECTED_NODE_1 and
                STARTN = SELECTED_NODE_2) then
               -- We found it, display of results
               FK_GET_ENTITY_INFORMATION
                  (FULL_DSC => DSC_ARC,
                   ENTITY_CLASS => ARC_CLASS,
                   ENTITY_TYP => ENT_TYP,
                   ENTITY_ID => ARC_ID);
              FK_PUT_MSG (MESSAGE => "Message for "&
                  TO_STRING (NAME) &
                  ": this arc (type " &
                  TO_STRING (ARC_CLASS) & ")
                  relates designated objects",
                  NUM_OBJ => ARC_ID);
               FOUND := TRUE;
               exit; -- useless to visit other nodes
            end if;
            ARCS := ARCS - 1;
            FK_CHECK_FOR_USER_INTERRUPTION;
         end loop;
         if not FOUND then
            -- No arc have been found
            FK_PUT_MSG (MESSAGE =>
                "Message for "& TO_STRING (NAME) &
                ": sorry, no arc between objects",
                EMPHASIS => FALSE,
                RESULT => TRUE,
                HISTORIC => FALSE);
         end if;
      end;
      FK_DISPOSE_MODEL (MODEL);
   end TOOL_EXAMPLE_BODY;
```

3.3.5. Configuration of the generic main

We thus instanciate the FrameKit generic main in order to get an executable program :

```
with TOOL_EXAMPLE_BODY,
     FRAME_KIT_GENERIC_MAIN,
     TOOL_EXAMPLE_HANDLER;
use TOOL_EXAMPLE_HANDLER;
procedure TOOL_EXAMPLE is new
   FRAME_KIT_GENERIC_MAIN
     (TRACE_FILE_NAME => "Tool_example",
      IS_STANDALONE_SUPPORTED => TRUE,
      MAIN_ALGORITHM => TOOL_EXAMPLE_BODY,
      TOOL_NAME => "tool_example",
      TOOL_VERSION => "1.0",
      TOOL_COPYRIGHT => "F.Kordon & J-L.Mounier",
      KIT_NAME => "EX",
      ON_LINE_HELP => "tool_example...[-verbose]",
      FK_INTERRUPTION_FROM_IU_INIT_HANDLER_SYSTEM
          => SET_HANDLER,
      FK_INTERRUPTION_FROM_IU_HANDLER
          => INTERRUPT_HANDLER);
```

3.3.6. Declaration of the new service

The portion of service menu related to this example is shown in Figure 9. It is composed with a terminal to invoke the tool («... execution») and a submenu for options. All items in an option list are check marks that can be set (or reset)

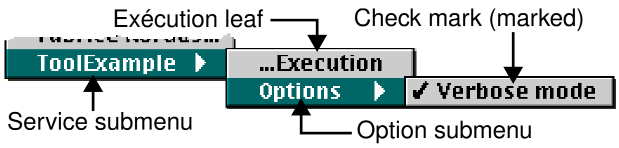


Figure 9 : *Appearance of the service submenu.*

The Service is declared as ToolExample. ToolExample is a hierarchical entry composed of one terminal item and an option list. The terminal entry associates the executable file, its parameter, its external name (language dependent) and its internal name (unique identifier for the menu).

To enable multi-architecture management as well as an easy installation procedure, service environment variables are introduced and represent absolute paths dynamically computed at invocation time. Here, FK_TOOLS_ROOT is computed using : 1) the FrameKit repository root (set during the installation procedure), 2) the related formalism (tools are sorted by formalisms), 3) the execution architecture. The path indicated after this variable reference is a relative path.

In the description below, access permission are set to the user *me*, the group list *privilege*. Apparently, the tool is only compiled on Sun/Solaris and PC/Linux architectures (no other architecture are set). Access permission can be inclusive (authorized users/groups/architectures are listed) or exclusive (unauthorized users/groups/architectures are listed).

Sometimes, users have to respect a procedure. For example, they have to compile first and then, if no error is reported, link can be performed. Such a sequence is handled in FrameKit by means of service preconditions. Three tags are associated to each internal service identification :

- LAUCHED_OK that returns TRUE if the associated program has run correctly,
- LAUNCHED_PB that returns TRUE if the associated program has outlined a problem,
- NEVER_LAUNCHED that returns TRUE if the tool was never launched.

Service preconditions can also be set according to values of session variables. Session variables are strings set or reset by tools. They allow a more refined mechanism in the management of available and unavailable menu entries (service as well as options). In our example, the ToolExample service menu is unreachable until a service internally named GRAPH_CHECKER is launched correctly (i.e. the tool signal that its execution is correct).

The check mark «verbose mode» is associated to an identifier (VERBOSE) in the item description. This identifier represents a variable that is either valued by "-verbose" (according to the definition) or empty. This variable can be referenced in the command line associated to a service in the EXECUTABLE command. The default value of the parameter (on or off) is provided in the execution (here, it is on).

```
SERVICES_FILE
BEGIN
 SERVICE (NON_TERMINAL, 'ToolExample')
  ACCESS INCLUSIVE
   USER ('me')
   GROUP ('privilege')
   ARCHITECTURE ('SUN_SOLARIS')
   ARCHITECTURE ('PC_LINUX')
  END_ACCESS
  BEGIN_PRECONDITION
   LAUNCHED_OK (GRAPH_CHECKER)
  END_PRECONDITION
  SERVICE (TERMINAL, 'Execution')
   ACCESS INCLUSIVE
    ALL_USERS
   END_ACCESS
   BEGIN_PRECONDITION
    TRUE
   END_PRECONDITION
   EXECUTABLE ($(FK_TOOLS_ROOT)'EX/tool_example',
               '$VERBOSE',
               COMM_NAMED_PIPE, TOOLEX)
   PROTOCOL (SAFE)
   QUESTION_INFO (STOP_ALLOWED, HISTORIC,
                  INFO_OBJECT, NO_FORMALISM)
  END_SERVICE
  SERVICE (LST_CHECK_MARKS, 'Options')
   ACCESS INCLUSIVE
    ALL_USERS
   END_ACCESS
   BEGIN_PRECONDITION
    TRUE
   END_PRECONDITION
   SERVICE (CHECK_MARK, 'Verbose mode')
    ACCESS INCLUSIVE
     ALL_USERS
    END_ACCESS
    BEGIN_PRECONDITION
     TRUE
    END_PRECONDITION
    CHECK_MARK (ON, VERBOSE, '-verbose')
   END_SERVICE
  END_SERVICE
 END_SERVICE
END
```

3.3.7. An idea about service execution

Let us assume that, after all the required stuff (connection, execution of service GRAPH_CHECKER etc.), service TOOLEX has been activated in verbose mode. Figure 10 provides a partial screen shot when the query associated to this option is executed.

The trace window (top right) displays state messages provided by FrameKit and by the tool (is any). The service, as declared in the previous section supports interruption and thus, it contains a STOP button. At least one object is selected (node «a» is apparently one of them) in the model window (middle back) that is a necessary condition to activate the service (its declaration mentions that complementary objects are required). The input windows (bottom right) is the one created when FK_GET_A_LINE is executed. We assume that the user typed «Me» in the dialog box. He can then either click on cancel (a default value is assumed by the tool as programmed) or OK («Me» is then supplied to the tool).

Figure 11 is a partial screen shot when the service is finished. We display the historic window (left) that contains the execution information sent by the tool. Results are displayed in a specific window (bottom front) associated to a set of objects (here, the arc relating the two designated nodes) that contain a message. This window allows to go from result component to result component.

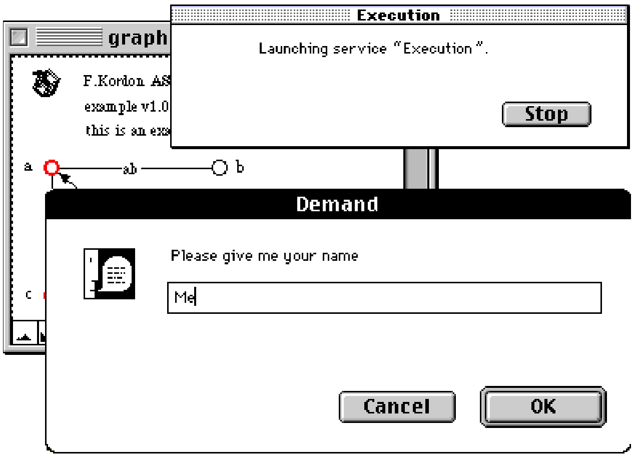


Figure 10 : *Execution, dialog.*

From the programmer's side, a result component is generated by primitives like FK_PUT_MSG or FK_PUT_ERROR. When result components are related to objects (or objects label) the corresponding element is automatically outlined and presented to the user. When no object is associated, the text is displayed in the historic window.

It is also possible for a tool to generate new models or to modify the input model.

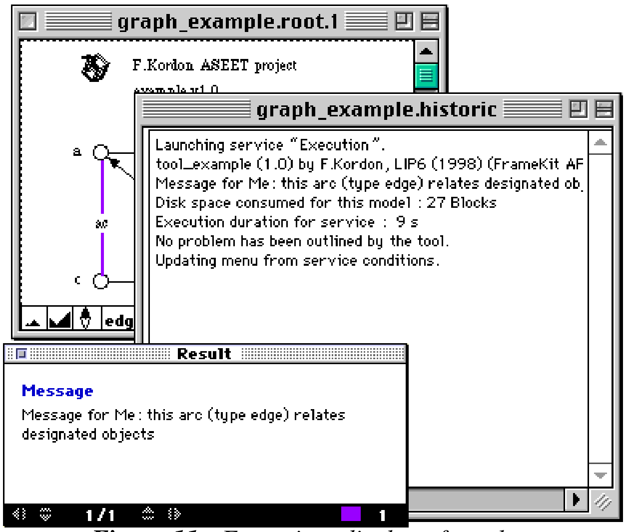


Figure 11 : *Execution, display of results.*

4. Conclusion

We have presented in this paper the principles of FrameKit, a software platform dedicated to the rapid prototyping of CASE environments. FrameKit is implemented in Ada and provides API for an easy implementation of new tools. We have illustrates the principle of tool implementation and declaration in FrameKit by detailing a small example.

The work presented in this paper is currently implemented and available on the Internet at <http://www-

src.lip6.fr/framekit>. It has been used to quickly prototype numerous CASE environment (about 25 tools distributed over 6 formalisms). The most important one is CPN-AMI, a Petri net based environment dedicated to formal specification and validation of parallel systems also available on the Internet at <http://www-src.lip6.fr/cpn-ami>. It is a collection of tools associated to three formalisms (Petri Nets and two high level description : OF-Class and H-COSTAM **[3]**).

We have experimented FrameKit for more than two years. The fastest tool integration is about ten minutes (it is reduced to a tool declaration) and one the longest took 110 hours. Average tool integration time is about fifteen hours for imported tools and less than half an hour for tools implemented using FrameKit API.

formalism	**Tool name**	**Tool type**[a]	**Integration time (hours)**		**Remarks**
			adaptation	**declaration**	
Petri Nets	GreatSPN (v 1.6)	I	6	0.2	Integration from executable files only, performed using Unix shell language.
	CPNsimulator	D[b]	110	1.5	Highly interactive tool. Major revision due to changes in the management of interaction in FrameKit.
	BooleanCondition	D[b]	0.5	0.2	Integrated using Unix shell language.
	CPNverifier	D[b]	1	0.2	Combination of three tools «glued» in a Unix shell script.
	CPNunfolder	D	0.5	0.2	Integrated using Unix shell language.
	CPNinvariant	D[b]	2	0.2	Integrated after a recompilation using C APIs.
	PROD (v 3.2)	I	24	0.5	Powefull but complex tool. Adapted using a specific driver implemented using Ada APIs.
	EVRunfold	I	4	0.2	Integration from executable files only, performed using Unix shell language.
	PetriBDD	D[c]	4	0.2	Integrated using Unix shell language.
	PrettyGraph (dot)	I	3	0.2	Adapted using a specific driver implemented using Ada APIs.
	LinearCharacterization	D	/	0.2	Integrated as is (it was implemented using C APIs)
OF-Class	OFC-verifier	D	/	0.2	Integrated as is (it was implemented using C APIs)
	PN-loader	D	/	0.2	Integrated as is (it was implemented using Ada APIs)
	PROD-ofc	I[d]	/	0.2	Small adaptation of the integration for Petri nets.
H-COSTAM	HCM-verifier	D	/	0.2	Integrated as is (it was implemented using Ada APIs)
	HCM2PN (prototype)	D	/	0.2	Integrated as is (it was implemented using Ada APIs)

Table 1: *Summary of tool integration to build CPN-AMI 2*

a. **I** for integrated tools, **D** for Developped tools.
b. Adapted from AMI, our previous platform.
c. Reuslt of a cooperation with two other universities and thus not designed to run in FrameKit.
d. This integration inheritates from the one done for Petri nets.

Table 1 summarizes the amount of time spent in the integration process to build CPN-AMI 2 (a full description of these tools may be found in **[14]**). This corresponds to the time required to get a first operational version, in order to evaluate the interest of the tool. Extra work may be required to exploit enhanced functions. All imported tools (PROD, ERVunfold, GreatSPN and dot) where integrated using the technique illustrated in Figure 7.a.

For most tools, we only had to perform a small adaptation (by means of a shell script that centralize the emulation of inline invocations) and the declaration to FrameKit (description of the Macao menu associated to the tool).

We use FrameKit to build demonstrators in industrial contracts. It is also used for two years in a master program to illustrate concepts of middleware components. It is used in a student project to practice their design and implementation.

5. References

[1] J.M. Bernard & J.L. Mounier, "Conception et Mise en Oeuvre d'un environnement système pour la modélisation, l'analyse et la réalisation de systèmes informatiques", Thèse de doctorat de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 1990

[2] J.Buxton, "DoD requirements for Ada programming support environments, STONEMAN", Dod High Order Language Working Group, February 1980

[3] A.Diagne & F.Kordon, "A Multi Formalisms Prototyping Approach from Formal Description to Implementation of Distributed Systems", in proceedings of the 7th "International Workshop on Rapid System Prototyping", N.Kanopoulos Ed, IEEE comp Soc Press, Greece, June 1996

[4] M.Dowson, "Integrated project support with ISTAR", IEEE software, November 1987

[5] ECMA, "A Reference Model for Frameworks of Stoftware Engineerings Environments", ECMA report number TR/55 (version 3), NIST Report, April 1993

[6] C.Fernstrom & L.Ohlsson, "The ESF Vision of a Software Factory", Proceedings of the International Conference on Software Development Environments & Factories, Berlin, May 1989

[7] B.D.Fromme, "HP Encapsulator : bridging the generation gap", HP Journal, June 1990

[8] C.Gerety, "HP softbench : a new generation of software developement tools", HP Journal, June 1990

[9] C. Hylands, E. Lee & H. Reekie, "The Tycho User Interface System", The 5th Annual Tcl/Tk Workshop '97, Boston, Massachusetts, pp 149-157, July 14-17, 1997

[10] F. Kordon & J-L. Mounier, "FrameKit and the prototyping of CASE environments", in proceedings 8th International Workshop on Rapid System Prototyping, N.Kanopoulos Ed, pp 91-97, IEEE comp Soc Press, USA, June 1997

[11] F.Kordon & J-L. Mounier, "Implementation of Genericity for customizable CASE environments", to appear in proceedings of CARI'98, Dakar, Senegal, October 1998

[12] J.Lonchamp, K.Benali, J.C.Derniame & C.Godart, "Towards assisted software engineering environments", Information and Software Technology, vol 33, n° 8, October 1991

[13] MARS-Team, "Macao Home page", <http://www-src.lip6.fr/macao>

[14] MARS-Team, "The CPN-AMI environment (version 2.2.1)", <http://www-src.lip6.fr/cpn-ami>

[15] T. Mowbray & R. Zahavu, "The Essential CORBA: Systems Integration Using Distributed Objects", John Wiley & Sons, 1995

[16] Ptolemy Team, "The Ptolemy Kernel-- Supporting Heterogeneous Design", RASSP Digest Newsletter, vol. 2, no. 1, pp. 14-17, 1st Quarter, April, 1995

[17] D.Schefström, "System Development Environments : Contemporary Concepts", in Tool Integration : environment and framework, Edited by D.Schefström & G. van den Broek, John Wiley & Sons, 1993

[18] A.Wasserman, "Tool Integration in Software Engineering Environments", LNCS 467 : "Software Engineerings Environemnts", pp 138-150, 1990