Implementation of Genericity for customizable 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

Résumé: *Les méthodes de Génie Logiciel actuelles, basées sur des représentations graphiques complexes, ne sont réellement opérationnelles que si elles sont supportées par des Ateliers de Génie Logiciel. De tels outils, hélas forts délicats à implémenter, prennent en charge le suivit de la méthode, libérant ainsi les ingénieurs qui se concentrent sur les problèmes à résoudre.*
Nous proposons dans cet article quelques principes de conception d'un Atelier de Génie Logiciel générique. Nos principes sont dérivés du modèle ECMA-NIST et se focalisent sur certains aspects (représentation graphique et intégration de nouvelles fonctions). Nous décrivons brièvement l'implémentation de ces concepts dans la plate-forme FrameKit et présentons notre expérience d'utilisation de cette plate-forme pour la construction de CPN-AMI, un Atelier de Génie Logiciel basé sur les réseaux de Petri.

Mots Clefs : *CASE, Intégration d'application, Méthodes de Génie Logiciel.*

Abstract: *Software engineering methodologies rely on various and complex graphical representations and are more useful when associated to CASE tools designed to take care of constraints that have to be respected. However, such tools are complex to implement.*
This paper proposes some principles derived from the ECMA-NIST model for the conception of a generic CASE environment and outline how some major aspects can be implemented (graphical representation and integration of new CASE functions). Finally, we describe the implementation of these concepts in the FrameKit platform and present results based on our experience with the construction of CPN-AMI, a Petri net based CASE.

Keywords : *CASE, Application Integration, Software Engineering Methods.*

1 Introduction

Software engineering methodologies rely on various and complex graphical representations as SA-RT **[14]**, OMT **[27]** Class-Relation **[7]** 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 methods.

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 **[26]** : 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 **[3]** is mostly data oriented and dedicated to Ada development. ESF **[11]** and HP-Softbench **[13]** suggest a communication oriented architecture. ISTAR **[8]** proposes a strong "process orientation" based on a contract concept defining inputs, outputs and constraints. Then, some standards like

ECMA **[9]** and then CORBA **[23]** 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 for such environment, especially when tools come from various origin. In a project like Ptolemy **[25]**, 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 **[15]**) takes into account the definition of evolutionary interfaces between major components.

This paper proposes an interpretation of the ECMA-NIST architecture in order to define a parameterization of a software platform dedicated to the implementation of various CASE. We try to formalize how such a platform can be parameterized in order to define a generic CASE environment model that emphasizes low cost reuse of software components.

For example, this parameterization is suitable for a quick and easy prototyping of CASE. A customized CASE is then deduced from a generic CASE plus information («values» associated to formal generic parameters). Figure 1 illustrates this customization procedure.

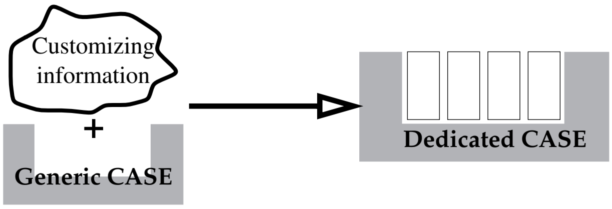


Figure 1 : *From a Generic CASE to a dedicated one.*

We briefly detail how we have implemented it in FrameKit **[16]**, a software platform dedicated to the prototyping of CASE tools we use for experimenting implementation of formal methods.

Section 2 deals with common techniques used in tool integration. Then, we detail in section 3 our interpretation to define a model of generic CASE, how it works and the outlines of its implementation in FrameKit. Finally, Section 4 presents results computed after the use of FrameKit for one year and an half.

2 Integration of software components

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 :

- the *a priori* integration : involved tools are designed especially for integration in a specific environment.
- the *a posteriori* integration : involved tools are already designed; source files may not be available.

The a priori integration 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 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...).

Three common techniques have been defined :

- *Encapsulation* : if executable file only are available, the tool must be encapsulated. The driver is an intermediary process that handles both communication with the platform and translation of exchanged data **[2]**. It may have to emulate the original execution environment of the tool

and/or implement the graph that models dependencies between tool's functions.
- *Rehosting* : this technique supposes that object files are available. Then, it is possible to link them with the execution environment libraries in order to get an executable file. Libraries that emulates standard system calls are then provided.
- *Strong integration* : if source files are available, they can be modified and adapted to the CASE environment interfaces. This solution does not raise any major problem. The a priori integration defined in section 2 is also a type of strong integration.

According to **[29]**, the integration procedure must take into consideration five integration axis :
- *Platform* : tools must be executed 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.

3 Towards a generic CASE model

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 **[29]** 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* : it describes representation rules of a knowledge domain,
- *Model* : it is the description of a given knowledge using a formalism. It is a «document» composed with objects defined in the formalism,
- *Service* : it is a tool function that correspond to operations in a design methodology.

Formalism is more related to the User Interface axis. Model is related to User Interface and data management axis. Finally, the service notion is strongly connected to the environment axis.

3.1 User Interface

In FrameKit, presentation and display services are strongly constrained. Both types of services are supported by Macao **[21]**, 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.

3.1.1 Elementary formalism

The definition of an elementary formalism corresponds to a list of basic classes C=<A,I> where :
- A is a set of elementary attributes.
- I is a set of clips that constitutes the class interface to which a name and a maximum number of connections is associated. Clips allow typed plugging between basic classes.

Example 1 : *Let us consider the object "Macintosh computer" composed of the following components :*

- *A = {name};*
- *I = { (appletalk clip,2), (ethernet clip, *)}.*

The name of the object identifies its category. Clips allow to plug an other object. The maximum of connection via an "appletalk" clip is 2. There is no connection limit via the ethernet clip.

Using this definition, a textual formalism contains one basic class having one textual attribute only. However, a graphical formalism contains «nodes» related via «connectors». So, to support any type of representation, we introduce two types of basic classes :

- Nodes being the components of the representation,
- Connectors describing relations between components of the representation.

Two nodes are linked together by means of a connector. Nodes and connectors are plugged through clips. It is possible to define connectors linking more than two objects (to model a bus for example).

Example 2 : *Let us consider two classes of nodes (N1, N2) and two classes of connectors (C1, C2). It is possible to assign a direction for each connector by means of clips. For instance, C1 will be dedicated to relations between N1-nodes and N2-nodes only. C2 is dedicated to express relations between N1-nodes only.*

We call formalism the tuple <A,I,N,C> where :

- A is a set of global attributes. Each attribute is typed. A set of elementary types is predefined in the environment.
- I is a set of interfaces. Interfaces are useful for hierarchy management. If I is empty, the formalism is elementary (non hierarchical representation).
- N is a set of basic node classes : they contain attributes and clips. An attribute describes a piece of information related to the node. A clip defies plug compatibility with connectors (to be connected, a connector must contain a similar clip).

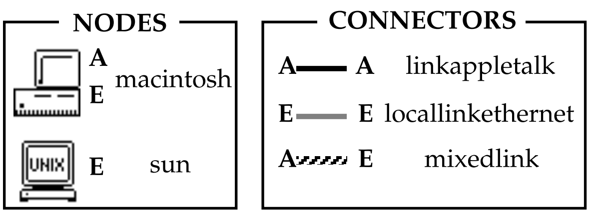


Figure 2 : *Items in the LOCNET formalism.*

- C is a set of basic connector classes : they allow to connect nodes. They contain of attributes and clips as well.

Example 3 : *Let us consider the LOCNET formalism that describes a local network composed with Macintosh computers and Sun workstations (Figure 2) :*

- *A = {name, local-address} where name is a string, local-address is a number.*
- *I = ∅.*
- *N = {macintosh, sun} where :*
  - *macintosh attribute is name (string); clips are : (appletalk, 2) and (ethernet, *).*
  - *sun attributes are name (string) and local address (integer); clip is (ethernet, *).*
- *C = {appletalk_link , ethernet_local_link , mixed_link} where :*
  - *appletalk_link has two clips appletalk*
  - *ethernet_local_link two clips ethernet*
  - *mixed_link has one clip ethernet and one clip appletalk*

Example 4 : *The Figure 3 gives an example of a LOCNET model. It is impossible to connect an appletalk connector to a sun node because there is no appletalk clip*

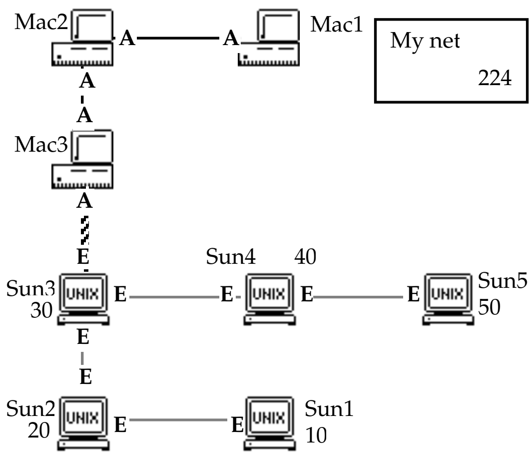


Figure 3 : *Example of a LOCNET model.*

3.1.2 Operations on formalisms

Some operations are useful to establish relations between formalisms :

- Enrichment : it is an addition of nodes, connectors, global attributes or interfaces;
- Restriction : it is a subtraction of nodes, connectors, global attributes or interfaces;
- Composition : it is a combination of formalisms.

It is possible to define subsets or super sets of a formalism by means of the first two operations.

Formalisms (elementary or not) can be composed to produce a complex formalism. This is useful to describe hierarchical graphical representations. We call recursive a formalism composed with himself. Such a configuration is necessary to describe an hierarchical object oriented description like OMT.

Hierarchical models derived from composed formalisms are then composed of pages. Each page is an instanciation of one of the formalism that composes the complex formalism. Links between pages are performed by means of boxes. A box is a node in a page, related to another page. So, hierarchical models can be viewed like an oriented graph where nodes represent pages and arcs boxes. Any graph has a node without predecessor corresponding to a root page that is the “top” of the model.

Example 5 :*Let us define the NET formalism that describes a network and its subnets (Figure 4). We consider that I = {(ethernet, 1)} in the LOCNET formalism; where the ethernet clip represents the entry point of the local network. The NET description is :*

- *A = {name,IP-address} where name is a string, IP-address is a number.*
- *I = ∅.*
- *N = {sun, subnets} where*
  - *sun attributes are name (string) and local address (number); clip is (ethernet, *).*
  - *subnet is a box including a LOCNET model. The clip is (ethernet, 1)): LOCNET interface.*
- *C = {ethernet_link} where :*
  - *ethernet_linkhas one clip ethernet*

A NET model is then composed of submodels called pages. In Figure 5, node “Small net” is a box that links the current page to the page on the left.

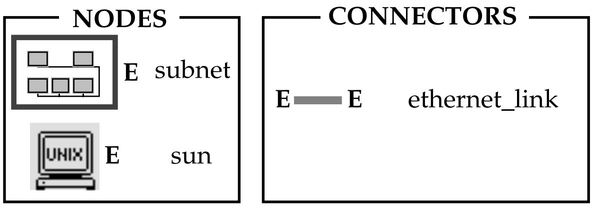


Figure 4 : *Items of the NET formalism.*

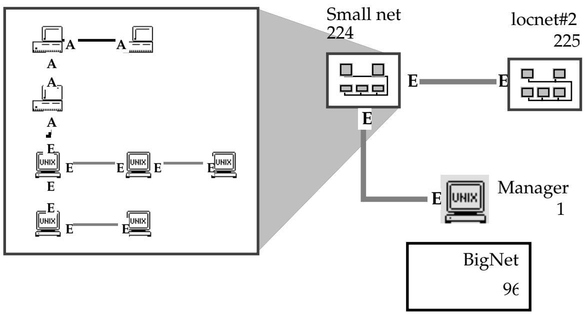


Figure 5 : *Example of a NET term.*

3.1.3 Services and tool invocation

Macao is not only a graph editor, it is also the front end of FrameKit. It allows a user to connect to FrameKit and open sessions. During a session, the user may apply services on the models. A session may be interrupted and restarted afterwards.

Services (called tasks in ECMA), are not originally typed. However, they take in input a set of models and produce another set of models. So, we propose to type a service S as follow :

> S : $F_i \rightarrow F_o$ where both F_i and F_o are included in F^*, the set of all defined formalisms.

A service is then an operation typed using input and output formalisms. This is a signature in the sense of programming languages.

Let us consider $F_i = \{f_{i1},..f_{iN}\}$ and $F_o = \{f_{o1}..f_{oM}\}$ ($f_{i1},..f_{iN},f_{o1}..f_{oM} \in F$). There are two types of services, according to conditions on F_i and F_o :

- Transformations : there is no relationship between elements of F_i and elements of F_o.

• Derivations : F_o is obtained from F_i. It means that, $\forall\ f_{ii} \in F_i, \exists\ f_{oj} \in F_o \mid f_{oj}$ is either an enrichment or a restriction of f_{ii}.

To ease the understanding of service by users, it is of interest to associate one service to similar tools' functions. Such an association corresponds to the definition of a polymorphic service. Polymorphism may be of interest either from :

• the "tool side" when it corresponds to discrete tools' operations having the same function. This is useful when the choice of one tool respects specific criteria (access rights, complexity according characteristics of the input model...);

• or the "formalism side" when it corresponds to semantically similar tools' operations applicable on discrete formalisms.

So, polymorphic services are useful to introduce performance criteria ("tool side") or to enhance transparency for related formalisms ("formalism side"). Their implementation involves mechanisms that are similar to dynamic binding in object languages **[22, 27]**.

Example 6 :*Let us consider two polymorphic services where :*

Connectivity evaluates if a graph is connected or not, statistic provides statistics about the formalism (number or nodes, number of connectors etc.). Potential polymorphism is outlined below :

- *The connectivity service is polymorphic on the "formalism side" because it has a signification for any graphical (and non hierarchical) formalism. It is reasonable to suppose that one tool is able to process any type of formalism if the model manager implements the characteristics mentioned in section 3.3.*
- *The statistic service is polymorphic on the "tool side" if, for a given formalism there are more than one tool that perform this function. Then the choice of a given tool is performed according to criteria associated to the service.*
- *The statistic service is polymorphic on both tool and formalism sides if several tools perform the function for discrete formalisms. When the service is invoked, the choice of the tool relies on the selected formalism and then, if more than one tool perform the function for this formalism, according to criteria associated to the service.*

Typing of services is not sufficient to implement a methodology because a sequence of operations may have to be performed according to a given order. This is why permissions are associated to each services. There are two types of permission access :

• Static permission deduced from the user identification,

• dynamic permission computed after the session state.

We propose in FrameKit the two mechanisms. First, access permission allow to hide a set of services. This is useful when implementing roles in a project (i.e. the project manager should have access to project management services while a quality engineer should be able to use quality services). Permission are evaluated once when users get connected.

Second, FrameKit manages *session variables*. A session variable contain a string and can be evaluated in a service precondition. Tools may affect values to these variables. Service conditions are evaluated after any service invocation. It is a way to temporarily disable or enable services in order to force a sequence that respects a given methodology.

It is also possible to consider the status of a service execution to build a service condition. Configuration like the one proposed in the following example can then be performed.

Example 7 :*Let us consider three services : compile, link, display_errors. compile must be run first. If it terminates OK, then, it is possible to link. Otherwise, the user must apply display_errors. In both cases, compile cannot be run a second time.*

According to the requirements, services condition should be defined as follow :

- *compile : compile was never launched,*
- *link : compile was launched OK*
- *display_errors : compile was launched with problems.*

3.2 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 **[2]** or capsule **[12, 26]**. For communication, the use of an internal Data Definition Language (DDL) makes this translation process easier and supports heterogeneity in tools.

A common DDL, implemented at the platform level, provides an indirect but standardised 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 **[18]**. However, this solution is heavy to implement and we propose a simplified model that is suitable for building a simple platform like FrameKit.

3.2.1 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.

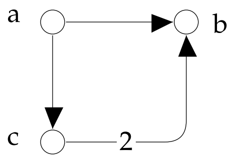
3.2.2 Fine grained data

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 attribute 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.

Example 8 :*Let us consider a small model defined using the elementary formalism OrientedGraph (Figure 6).*

Its definition is transported using simple messages that carry out syntactic aspects only. Instruction CN create a new instance of the referenced class (a «node» in line 3). CA instanciates a new connector of *the referenced class (an «arc» in line 9). CT associates a textual label to connectors or nodes (affectation of value «bø to label «name» of object number 4 which is a «node» in line 4). FO is used to identify the formalism identification and VM the version of this formalism. This information is used for check by tools only.*




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

Figure 6 : *OrientedGraph model and its corresponding internal description.*

One last advantage of this mechanisms is that it rely on ASCII information only. This 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.

3.3 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 mentioned in Section 3.1.3 It is the set of low-level services 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...).

The third point is also important because it proposes a framework for the evolution of the environment. The distribution 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 9 : *Figure 7 proposes an instanciation of the distribution model we propose. Let us imagine that a software engineering environment is being developed in discrete places. Such a distribution 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).*

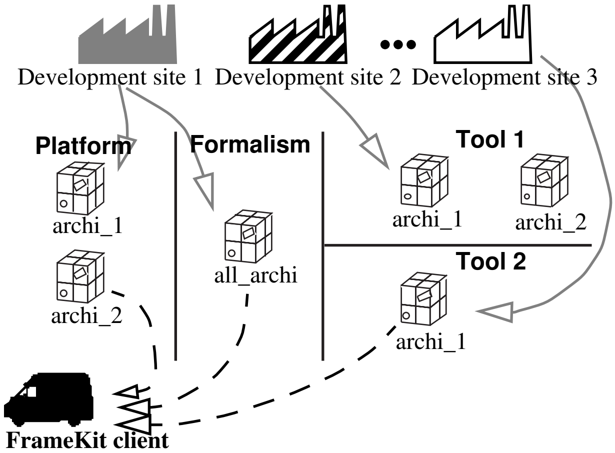


Figure 7 : *Example of the distribution model.*

3.4 Evolution of the generic CASE

A CASE environment *E* is made of a generic platform that offers communication services and a generic user interface (it can be seen as a default tool). *E* has two generic formal parameters : a set of formalisms and a set of services (polymorphic or not).

Definition of new formalisms or services should be supported by dedicated administration tools. In that case, these tools must be provided by the generic platform.

We note the generic CASE environment E (F^*,S^*) where F^* is the set of formalisms and S^* the set of services

During its execution, the environment changes; it “moves” into several *E (F,S)* adapted to the current user needs. *F* and *S* are the effective values given to F^* and S^* in order to obtain the appropriate *E*. The process which produces *E* from E is called adaptation and respects the following constraints :

- *F* is included in F^* which represents all formalisms defined for *E*;
- *S* is included in S^* which represents all possible services available for *E*;

F^* and S^* are potentially unbounded. If we need a formalism which is not already included in F^*, it is possible to add it. S^* has the same property, however, its elements always respects the following constraint : $\forall\ s \in S^*\ s : F_i \rightarrow F_o$ where F_i and $F_o \in F$

The evolution can be dynamic (Figure 8). It is similar to inheritance in Object Oriented languages : the set of formalisms and services can be modified during its execution. Adaptation is dynamic when it does not requires any compilation. The CASE environment is modified through configuration files or any other similar mechanism.

The produced CASE is easier to maintain : addition of new kits can be performed easily by adding/removing kits. In Figure 8, the second dynamic adaptation can be considered either as maintenance on tools and/or formalisms or the installation of a new function in the CASE environment. Such a mechanism is more difficult to implement but allows more flexibility.

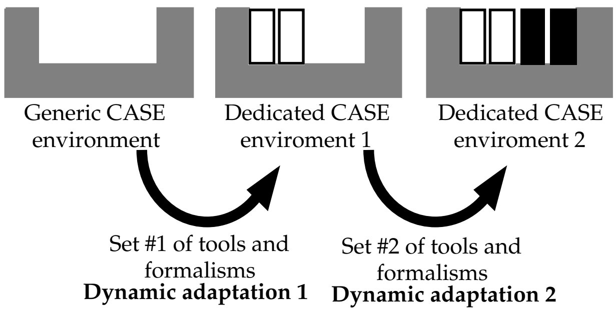


Figure 8 : *Dynamic adaptation mechanism.*

Such an evolutionary mechanism however has an influence on the architecture of tools in the environment. Let us now consider both implemented tools (a priori integration) and imported tools (a posteriori integration).

3.5 Structure of an implemented tool

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 3 The algorithmic part of the program should be disconnected from the environment and relate with it only by means of the APIs (Figure 9).

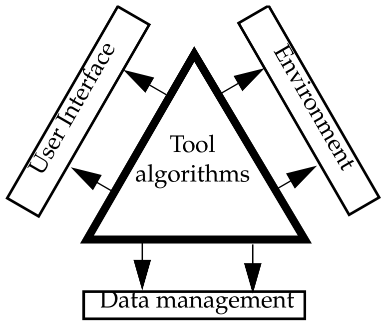


***Figure 9 :** Architecture of a tool designed to run in the software environment (a priori integration).*

Our implementation in FrameKit follow this strategy. Moreover, the «main» program of an application 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 link with APIs libraries is required.

3.6 Structure of an integrated tool

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.

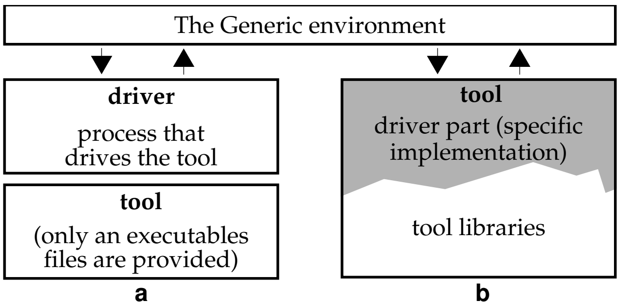


***Figure 10 :** Possible architectures of a posteriori integrated tools.*

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 10.a). The environment only knows about this process which architecture is the one defined in Figure 9. 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 10.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.

4 Results

We have implemented those principle in the FrameKit integration platform which is freely available on the Internet since March 1997 **[16, 20]**. We have also used FrameKit to build CPN-AMI 2, a Petri net based software environment for the modeling and evaluation ad prototyping of distributed applications **[19]**.

CPN-AMI 2 relies on three formalisms : Well formed Petri Nets **[4]**, OF-Class and H-COSTAM that are Petri net encapsulations suitable for an Object Oriented modeling approach **[6]**. OF-Class focuses on the preliminary design and structuration of the system (coherence between software component interfaces) while H-COSTAM emphasizes implementation aspects of the system (mapping of a conceptual solution into an operational software architecture).

The production process of a customized CASE (presented in Figure 1) is instanciated in

Figure 11. CPN-AMI implementation is reduced to the description of three formalisms and the implementation or integration of sixteen tools. Once formalisms and tools (either implemented or integrated) are declared in the FrameKit structure, they are available to the users and kit are produced for distribution over the Internet.

Within the sixteen tools currently integrated in FrameKit to compose CPN-AMI 2, four have been developed in other universities (GreatSPN **[5]**, PROD **[28]**, EVR-unfold **[10]** and dot **[17]**).

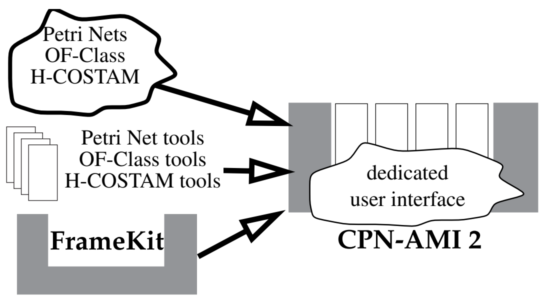


Figure 11 : *Production process of CPN-AMI 2.*

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 **[19]**). This corresponds to the time required to get a first operational version, in order to evaluate the interest of the tool for our methodology. Some times (essentially for PROD and dot), some extra work was required to access to enhanced functions. All imported tools (PROD, ERVunfold, GreatSPN and dot) where integrated using the technique illustrated in Figure 10.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).

CPN-AMI is still being improved and extended by addition of new tools in order to cover and fully implement our design methodology for distributed systems. We use it for industrial contracts and teaching.

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.

5 Conclusion

In this paper, we have presented a parameterization of a CASE environment in order to enhance a quick implementation of CASE tools dedicated to specific development methodologies. The proposed procedure emphasizes two aspects : a standard data definition technique that relies on formalisms as well as association techniques between tool's functions and services. Both formalisms and services are parameters of the generic CASE environment.

The result of this work is implemented in FrameKit , a generic CASE environment we use for the rapid prototyping of CASE environment that implements methodologies for evaluation purpose. It has been used to build CPN-AMI 2, a multi-formalism CASE environment based on Petri Nets. Both FrameKit and CPN-AMI are available on the Internet `<http://www-src.lip6.fr/cpn-ami>` and `<http://www-src.lip6.fr/framekit>`.

6 References

[1] Ada Join Program Office, "Reference Manual for the Ada-95 Programming Language", U.S. Department of Defense, The Pentagon, Washington, January 1995

[2] 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

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

[4] G. Chiola, C. Dutheillet, G. Franceschini & S. Haddad, "On Well-Formed Coloured Nets and their Symbolic Reachability Graph", High Level Petri Nets. Theory and Application. Edited by K. Jensen G.Rozenberg, Springer Verlag 1991

[5] G. Chiola, "GreatSPN 1.5 Software Architecture", In Proceedings of the 5th International Conference Modeling Techniques and Tools for Computer Performance Evaluation, Torino (Italy), February 1991

[6] 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

[7] Desfray P., "Object Engineering, the Fourth Dimension", Addison-Wesley, 1994

[8] M.Dowson, "Integrateed project support with ISTAR", IEEE software, November 1987

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

[10] J. Esparza, S. Römer & W. Vogler, "An Improvement of McMillan's Unfolding Algorithm", in proceedings of Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1055, pp 87-106, Springer Verlag, March 1996

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

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

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

[14] D.Hatley & I.Pirbhai, "Strategies for real-time system specification", Donset house publishing Co, 1988

[15] 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

[16] F. Kordon & J-L. Mounier, "FrameKit and the prototyping of CASE environments", 8th IEEE International Workshop on Rapid System Prototyping, Research Triangle Park Institute, IEEE comp Soc Press N° 97TB100155, pp 91-97, June 1997

[17] E. Koutsofios & S.C. North, "Drawing graphs with dot", Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, September 1991

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

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

[20] MARS-Team, "The FrameKit home page", <http://www-src.lip6.fr/framekit>

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

[22] B.Meyer, “Conception et Programmation par Objets”, InterEditions, 1990

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

[24] J. Rumbaugh, M. Blaha, F. Eddy, W. Premerlani& W. Lorensen, "OMT : Object Oriented Design and Modeling", Masson & Prentice hall, 1994

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

[26] 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

[27] B.Stroustrup, “The C++ programming language”, Addison-Wesley, 1991

[28] K. Varpaaniemi, J. Halme, K.Hiekkanen & T.Pyssysalo, "PROD reference manual", Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995

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