Layla: A Pattern-based Framework for Network Management Interfaces

Rudolf K. Keller Jean Tessier*

Département d'informatique et de recherche opérationnelle
Université de Montréal
C.P. 6128, succursale Centre-ville, Montréal (Québec) H3C 3J7, Canada
voice: (514) 343-6782, fax: (514) 343-5834
e-mail: keller@iro.umontreal.ca, Jean.Tessier@att.com

web: http://www.iro.umontreal.ca/~keller/***Layla***

Abstract

Developing network management interfaces (NMIs) is a challenging task involving multiple software layers, application programming interfaces (APIs), specification languages and tools. In order to ease the job of NMI developers, we have developed Layla, a prototype application framework supporting Open Systems Interconnection (OSI) NMIs. Layla is based on a heterogeneous yet coherent system of design patterns that comprises previously published patterns, several new and domain-specific patterns taken from NMI standards, as well as a couple of basic patterns relevant in Layla's API. Our research indicates that frameworks can indeed be built for a domain as complex as NMIs, and that they have a positive impact on both the development process and the resulting NMI products. Also, our experience confirms that the expected benefits of using design patterns, such as flexibility, reusability, and documentation value, do hold, yet that building pattern-based frameworks is a highly iterative and demanding activity. In this paper, we discuss APIs for NMIs and the need for application frameworks, detail the Layla framework and its system of patterns, describe NMI development based on Layla, and put our framework into perspective.

Keywords: Application Framework, Network Management Interface, Design Pattern, API (Application Programming Interface), OSI (Open Systems Interconnection).

Point of Contact: Ruedi Keller

This work was in part funded by the Ministry of Industry, Commerce, Science and Technology, Quebec, under the IGLOO project organized by the Centre de Recherche Informatique de Montreal, by Teleglobe Canada Inc., and by the National Sciences and Research Council of Canada.

* Author's contribution is part of his Master thesis research at Université de Montréal. Author's current affiliation: AT&T Laboratories, Advanced Technologies Division, Holmdel, NJ.

INTRODUCTION

Network management systems are used to control and monitor the components of distributed systems such as communication networks, where many different subsystems need to collaborate together to offer a service. Communication networks are large dynamic systems that evolve over time, with parts getting removed and others being grafted, often from various sources. Network management is a challenging task in that it usually requires remote access to widely distributed information coming from various sources. Operations have to be performed on large numbers of system components. Moreover, the access interface to the components can greatly vary, depending on their nature, type, and manufacturer. We define a *network management interface* (NMI) as the middle layer of a network management system, situated between the high-level control processes and the low-level components of the system [16]. The lower layers, which usually depend heavily on the execution platform at hand, are thus not part of the NMI.

Figure 1 illustrates a sample distributed system (network) under management. The system consists of two workstations communicating through a switch. Each workstation has a communication port that is attached to each end of the connection path, and the path itself is contained in the switch. The management system includes a management console that has access and control over all the components of the network through a symbolic representation provided by the NMI. Note that the NMI must include a number of communication stacks to access all the various components.

International standardization bodies have produced various tools for defining network management systems and their NMIs. Among the most advanced tools are the *Common Management Information Service (CMIS)* of *Open Systems Interconnection (OSI)* [16], and the *Simple Network Management Protocol (SNMP)* of the Internet [12]. Whereas CMIS, along with *CMIP*, its protocol for information exchange between systems, is based on the object-oriented

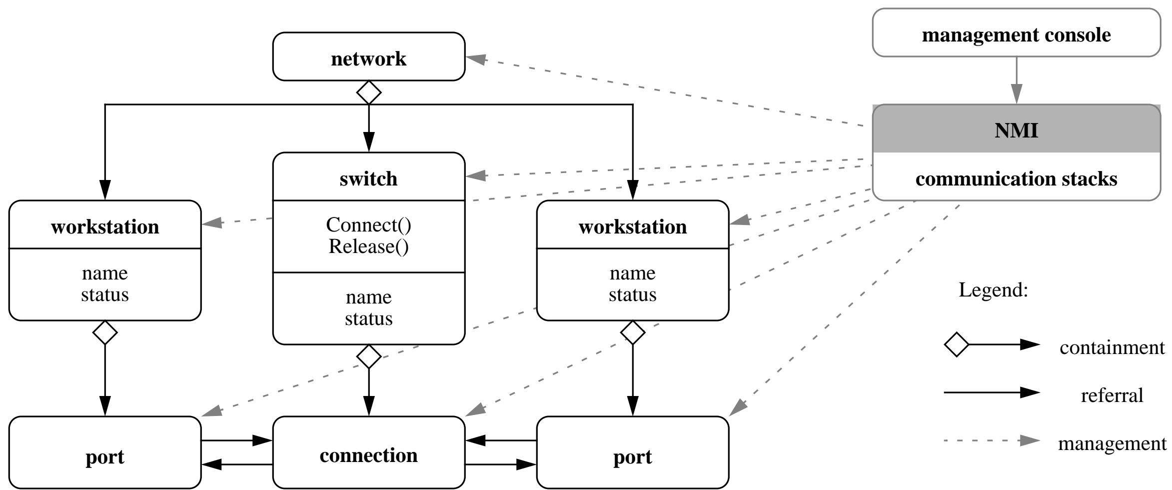

Figure 1: Sample network managed by network management system.

paradigm, SNMP uses tables not unlike the tables used in the relational model of databases. However, SNMP is moving towards the object-oriented paradigm, with its new version *SNMPv2* embodying some notion of inheritance.

In the *IGLOO* project, a large, ongoing project on object-oriented software engineering, we are involved in the project part that addresses network management problems. Over the past two years, we have developed *Layla*[1] [17, 18], a prototype application framework for NMIs. Using Layla, we have built several NMIs to date, in cooperation with *Teleglobe Canada Inc.*, our main industrial partner.

In designing *Layla*, we wanted to leverage off commercial implementations of standardized network management protocols, and therefore came up with a number of wrapper classes that encapsulate the specific details of any particular protocol engine. Layla was built for OSI NMIs and therefore includes provisions for the object-oriented nature of CMIS which are not necessarily found in other protocols for network management. At an early stage in the development of Layla, we decided to take an approach based on design patterns [2, 4, 11]. The resulting framework

1 In the *Zohar*, Layla is an angel in charge of the newly created spirits (Gustav Davidson, *A Dictionary of Angels*, Free Press, 1967).

architecture can be described as a heterogeneous system of design patterns. The system consists of previously published, general-purpose patterns, several new and domain-specific patterns taken from NMI standards, as well as a couple of basic patterns relevant in Layla's application programming interface (API). The patterns are implemented as a system of C++ classes that form the framework. The framework encapsulates to a large extent the underlying communication API (henceforth simply referred to as "API"), as has been shown with the adoption of two different, commercially available APIs.

Below, we first discuss currently available APIs for NMIs and the need for application frameworks. The architecture of Layla is described next as a system of design patterns. We also outline application development with Layla as a five step process. Afterwards, an example is presented, and design and implementation aspects are covered. Finally, we put Layla into perspective and draw some conclusions.

APPLICATION PROGRAMMING INTERFACES FOR NETWORK MANAGEMENT INTERFACES

Many software manufacturers offer packaged solutions for implementing NMIs. Such solutions are often called *Application Programming Interfaces* (APIs). An API typically includes data structures and function prototypes, as well as a set of precompiled libraries that implement those functions. A developer can describe a network management function in terms of API calls and implement the NMI by reusing the code in the libraries.

Figure 2 shows a typical network management system based on an API. The components of the network are represented as *Managed Objects*. The Managed Objects are grouped under an agent (*Agent Application*) and controlled remotely by a manager (*Manager Application*). The agent and the manager use both the *API* to communicate with each other and to exchange management information.

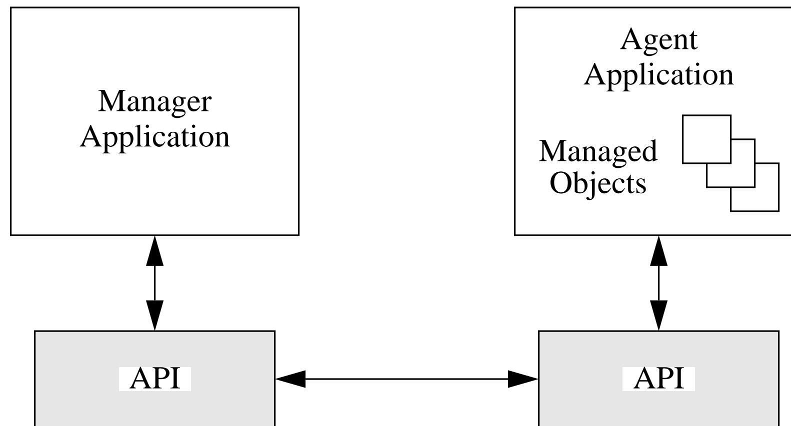

Figure 2: APIs for Network Management Interfaces.

Developing an NMI based on an API usually involves a number of steps (see Figure 3). First, the developer must specify the NMI using the specification language(s) supported by the API. For example, CMIS uses one language for describing object classes and relationships *(GDMO)* and another for the data structures used by the objects (ASN.1). In contrast, the SNMP standard uses only one specification language to describe data structures (ASN.1), and the semantics are described using plain English. Once the specification has been written, an automated tool is used to map it to a specific programming language, often C. The tool, typically some sort of *compiler*, needs to generate the appropriate *data structures* and *utility functions* in the target implementation language. Once generated, the developer can use these structures and functions to implement the NMI. The data structures are used by the *application* to pass information to the *API*. The utility functions are used by the application to manage those structures, and by the API to transfer data across process boundaries (via the *presentation layer (communication module)*). The API also comprises management protocols for data exchange across the various computing platforms typically found in today's large and heterogenous networks.

Network management APIs are a powerful development tool, since they take care of many low-level communication issues such as connection establishment/release, buffering,

synchronization, and type conversion between different computing platforms. Still, they exhibit several shortcomings.

First, most APIs available today provide only an interface to the C language, which entails that management applications must be mainly implemented in C. On the other hand, many NMI specification languages are object-oriented. This discrepancy of paradigms forces the developer to deal with object-oriented specifications, yet procedural interfaces. Clearly, the specification and implementation paradigms should be the same, regardless of the intermediary paradigm supported by the API.

A second problem comes from the use of generated code in application development. Applications are highly dependent on the output of the specification compiler, and any changes either to the initial specification or to the compilation mechanism may require major modifications to the application code, ultimately leading to serious maintenance problems. The generated code must therefore be encapsulated so that it is decoupled from the application-specific code. In this way, changes to one kind of code may be without impact onto the other kind.

Third, many APIs offer functionality which is not necessarily relevant to the NMI under development. For example, several APIs support multiple management methodologies, e.g., CMIS and SNMP. This leads to large and complex APIs which are overly difficult to master and support. Also, such APIs lead to larger applications requiring more computing resources. Therefore, a means for tailoring an API to specific requirements is badly needed.

Finally, many APIs are limited to the lowest common denominator of the functionalities provided by the various flavors of NMIs. They do not exhibit the higher-level functionalities that might be expected from specific NMIs. Yet, these high-level functionalities are often the very reason why a particular flavor of NMI was adopted in the first place. For example, CMIS includes a common access interface for all managed objects and an operation filtering mechanism. The

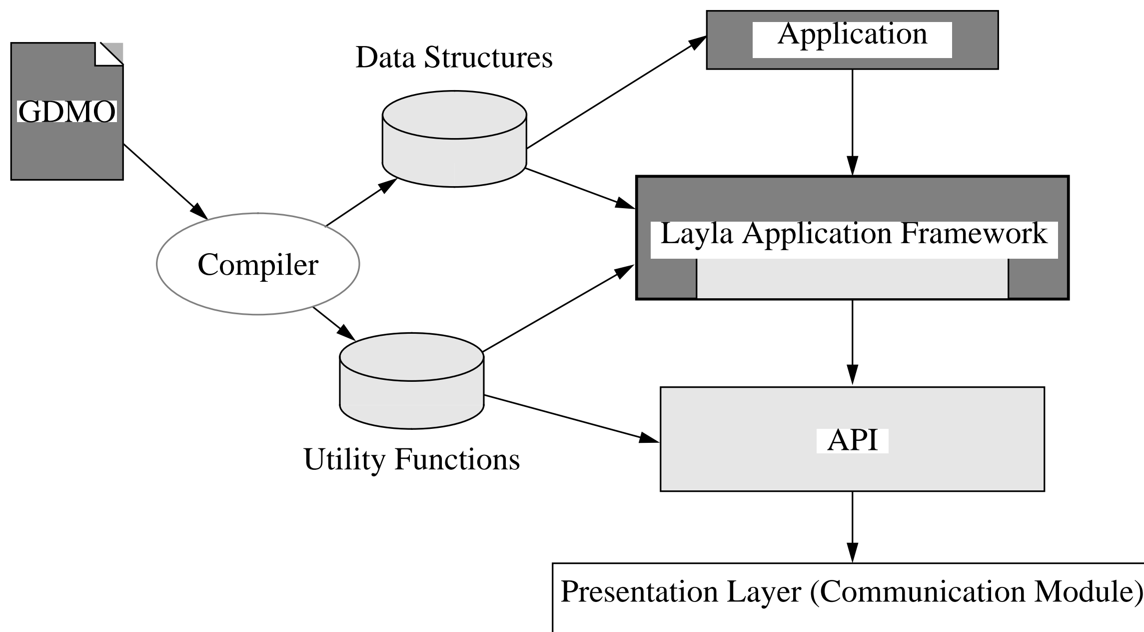

Figure 3: Layered structure of NMI, with Layla mediating between object-oriented (dark grey) and procedural (light grey) components.

developer has to implement those mechanisms for each NMI, whereas it would be more efficient to make them part of the API and reuse them across implementations.

We felt that these shortcomings would be best addressed by devising an application framework to mediate between the application and the API. The developer then simply has to deal with the object-oriented specification of the NMI and the framework, which itself is object-oriented, and thus the above mentioned paradigm gap is closed. Also, a framework encapsulating the underlying API makes NMIs independent of the API that is actually used. Furthermore, generated code should be hidden within the framework, and the framework should be powerful and flexible enough to provide minimum but sufficient functionality for building NMIs, including high-level services.

With these requirements in mind, we set out to build Layla, our prototype application framework. Figure 3 depicts its role in NMI development, where it stands between the object-

oriented NMI application and the procedural API, providing an object-oriented interface to the NMI developer. It mediates between object-oriented (dark grey) and procedural components (light grey), and comprises itself a procedural kernel for interfacing with the API. Note that introducing such an additional framework layer may lead to performance degradation. Our experience in the project, however, suggests that with careful framework design, performance loss can be kept to a minimum.

OVERVIEW OF FRAMEWORK ARCHITECTURE

For our framework development, Teleglobe supplied us with two different APIs, both geared towards OSI NMIs as defined by the CMIS standards [16], and providing the low-level functionality required. One of them, used at the beginning of our project, is *BaseWorX* from *AT&T* [1], the other one, on top of which the current version of Layla is built, is *DM* from *Hewlett Packard* [5]. We have strived for complete encapsulation of the API into the framework, so that the classes defined for a given NMI can be compiled with one API or the other, thus making the NMI independent of the API that is actually used. However, since the NMI is dependent on the structures generated by the specification compiler that comes with the API (see Figure 3), this encapsulation can never be complete.

From the outset, we were taking a development approach based on design patterns. Such an approach has the potential of making architectures easier to modify, maintain, and reuse, and to improve system documentation, as has been widely publicized [2, 4, 11, 14]. We started by building a simple toolkit containing some utility classes. But as new classes were added that took more and more of the application's main processing, the toolkit was turned into an application framework. In designing the framework, we were guided by published, general-purpose design patterns which we applied to the network management-related tasks at hand. Framework tasks

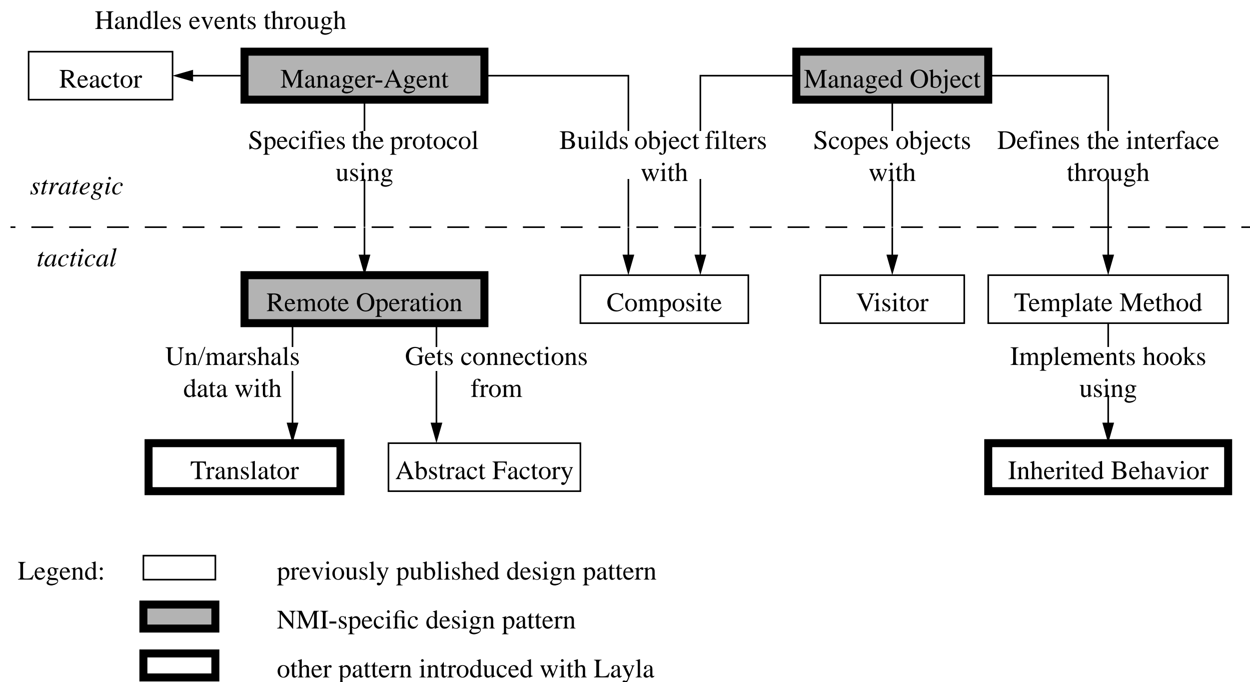

Figure 4: Pattern-based framework architecture of Layla.

which were not captured by existing design patterns led us to study and devise domain-specific design solutions, some of which we consider generic enough to be design patterns. In this paper, we describe Layla 1.1, the most recent version of the framework. Since the current version is still an evolving prototype system, it is likely that the framework architecture will eventually comprise some further design patterns. We believe, however, that the general structure of the pattern system is now stable. Using Layla, NMIs are built by specializing some of the framework classes and composing them with others (in its current form, Layla is a hybrid of a white box and a black box framework [6]), and abstract superclasses take care of the interrelations between the concrete classes.

Each major task in Layla is described by a design pattern, many of which come from the literature. This indicates that the published design patterns are indeed expressive and generic enough to be easily applied to a new application domain. Several tasks and design solutions that were not already design patterns originated from the constraints imposed by the underlying

network management standards. Since these solutions are applicable to virtually any network management system, they can be thought of as NMI-specific, hence domain-specific design patterns [18] (*Manager-Agent*, *Managed Object*, *Remote Operation* in Figure 4). Two further tasks were considered flexible and generic enough to be applicable to other fields, and were thus described as design patterns, too (*Translator* and *Inherited Behavior* in Figure 4). They were then circulated on mailing lists and generated some interesting feedback from the design pattern community.

Figure 4 depicts the system of patterns that constitutes the architecture of the framework. Strategic patterns [15] are placed at the top, and the more generic but less critical tactical patterns are shown in the region below the separator line.[2] Arrows stand for use-relationships, indicating that the originating pattern uses the destination pattern for the functionality mentioned on the label of the arrow. We believe that this system of patterns can serve as a basis for other frameworks and applications in the NMI domain [18]. As is, it captures well the Layla architecture and conveys a high-level understanding of the workings of the framework.

NMIs interact with one another using the *Manager-Agent* pattern. Each NMI can either play a manager or an agent role. The management protocol between managers and agents is described using the *Remote Operation* pattern, which itself uses the *Translator* pattern for the marshaling and unmarshaling of arguments, and *Abstract Factory* to obtain connections between NMIs. NMIs handle management events using the *Reactor* pattern.

Agent NMIs use the *Managed Object* pattern to model their resources, using *Template Method* and *Inherited Behavior* to implement the required call-back methods. The *Composite*

2 Note that the distinction between strategic and tactical pattern refers to the role of a pattern in a particular context [15]. In Figure 4, patterns are grouped into strategic and tactical according to their role in Layla.

pattern can be used to represent complex filters used to screen the targets of operations, with the screening action itself being handled by the *Visitor* pattern.

The *Manager-Agent* and *Remote Operation* patterns, governing a large part of our prototype framework, can be considered key patterns of an NMI pattern system. They are detailed and compared to related patterns (such as the patterns presented in [2]) in [18]. The *Managed Object*, *Translator*, and *Inherited Behavior* patterns are further discussed in [17]. Template descriptions as suggested in [4] of all these five patterns are available on-line at the Layla web site.[3] All other patterns of the system are described in [4], except for *Reactor*, which is discussed in [13].

APPLICATION DEVELOPMENT WITH LAYLA

Developing a Layla application, whether it is an agent, a manager, or both, is closely tied to the specification of the information to be exchanged. The development procedure consists of five steps. The first two steps consist in preparing the specification for use by the NMI and change little with the introduction of Layla. The remaining three steps deal with implementing the NMI itself. The amount of work required for these latter steps is an order of magnitude lower, if Layla is used instead of a "pure" API. Remember that Layla supports OSI NMIs and uses CMIS.

1. Writing the managed objects' specification in GDMO and ASN.1. A specification of the resource to be managed, in the form of managed objects, has to be written using the GDMO and ASN.1 languages. GDMO describes the class structures and relationships, while ASN.1 describes the data types used for attributes and various parameters. The GDMO code and the ASN.1 code are usually placed in separate files. Other information regarding those files might be required, depending on the GDMO compiler that is used.

---

3 Layla website: `http://www.iro.umontreal.ca/~keller/Layla`

2. Compiling the specification and generating the code. The specification files are put through the GDMO compiler in order to produce the utility functions for the API (see Figure 3). Moreover, additional tools that come with Layla need them as input files, too. These tools are used to produce custom tables for the automatic processing of the ASN.1 data types (printing, copying, etc.). The tools may slightly vary from API to API, since they take into account the information already provided by a given API.

3. Implementing the managed objects as subclasses of `CMO`. For each managed object class in the specification, a corresponding C++ class must be implemented. All managed object classes in GDMO inherit from a special managed object class named `top`, which is itself defined in GDMO [16]. In Layla, `top` itself inherits its managed object interface from a special class named `CMO`, which defines how to access a given managed object (a class instance). This interface uses the *Template Method* pattern [4] to provide access methods which in turn call hook methods that are redefined by the subclasses. These hook methods use the *Inherited Behavior* pattern mentioned above in order to reuse and extend the body of the corresponding inherited method. As these tasks are quite repetitive, the use of a number of C macros is suggested to ensure consistent behavior throughout. In later versions of Layla, we want to supply tools that will eventually automate most of these tasks.

4. Implementing an agent application. The developer using Layla can build an agent application by simply defining a subclass of `CAgent` (itself a subclass of `CCMISApplication`), which initializes a given MIB with instances of the subclasses of `CMO` that were implemented in the previous step. The behavior inherited from the class `CAgent` enables the processing of requests coming from manager applications. However, it is up to the developer to provide the back-end necessary to realize the *Managed Object* pattern.

5. Implementing a manager application. The developer using Layla can build a manager application by defining a subclass of `CManager` (itself a subclass of `CCMISApplication`), which performs particular management functions such as collecting data, compiling reports, monitoring, etc. The behavior inherited from the class `CManager` allows for the automatic sending and receiving of CMIP messages, yet the developer has to provide the manager-specific behavior as well as the manager's user-interface, typically a graphical user interface.

A SAMPLE APPLICATION USING LAYLA

One of the practical interests of Teleglobe in the Layla project was the implementation of the *Xcoop* protocol [3]. This protocol supports the reservation and establishment of ATM VPCs. *Asynchronous Transfer Mode (ATM)* [9] is a transport protocol for high-speed, optic fiber-based integrated networks. It divides a data stream into cells that are then sent on a permanent switched circuit, or on a *virtual path connection (VPC)*. These VPCs are established between ATM systems prior to the actual connection establishment phase. Xcoop is a protocol that uses network management functions to reserve and activate such VPCs. The Xcoop MIB uses six managed objects classes; but with all their superclasses, this number goes up to a total of fifteen managed object classes.

While developing the Layla framework, we implemented an Xcoop agent for a transit system. An Xcoop agent and a *mapper manager* interact according to the *Manager-Agent* design pattern. The Xcoop agent contains the MIB, a collection of managed objects implemented using the *Managed Object*, *Template Method*, and *Inherited Behavior* patterns. The managed objects are grouped using the *Compositor* pattern. An implementation of the *Abstract Factory* pattern is used to connect the two processes and to enable CMIP operations (implemented using the *Remote Operation* pattern). The mapper manager sends an `M-GET` request to the top object in the agent's

containment hierarchy with a scope designating the whole tree. When the agent receives the request, it uses a `CScope` object (an implementation of the *Visitor* pattern) to enact the `M-GET` request in each and every managed object in the MIB. Each managed object gives rise to an `M-GET` response message back to the mapper manager, which sorts them out and uses that information to display a map of the MIB.

The managed object classes that comprise Xcoop contain a fair number of attributes. Implementing them became rapidly tedious. Even though a large part of the work could be handled in a *cut & paste* manner, there were slight variations from one iteration to the other that had to be taken into consideration. This prompted us to use with Layla extensively the macro facilities of the C preprocessor. This addition to the Layla framework greatly eases the job of implementing managed object classes as subclasses of `CMO`, without compromising system performance.

The implementations of the Xcoop example and of other NMIs we have built use the *TMN* library, a set of generic managed object classes for telecommunication networks [7]. Also, we reused parts of *DMI*, a CMIS library of core managed object classes for the management of distributed systems, which in turn reuses data types from a number of sources [16].

In Table 1, some quantitative data about the Xcoop and a tutorial example is presented. The tutorial example helps novices to get familiar with Layla. It implements the NMI for a simple network similar to the one presented in Figure 1 [17]. Note that a lot of library code could be used, particularly in the tutorial example, where the application-specific code represents less than 5% of the overall specification code. Much of the code in the managed object implementation files (`.C`) consists of calls to macros that generate the proper processing syntax upon expansion. Also note that major parts of the implementation code consist of *Template Method* hooks, which are implemented through *Inherited Behavior* and which contain a lot of *cut & paste* material (some 50% of the total code).

The fact that Layla is a framework means that most of an application's processing is taken care of by the framework, not by the application developer. Once the managed object classes are implemented and incorporated in the framework, it becomes fairly easy to write management applications. One needs only describe the particular processing that is to be performed by the application. Layla also allows for easy reuse of application code with different MIB specifications. On the down side, the resulting executable programs are larger and possibly slower than if they had been developed with the API only. But an application built directly on top of an API would be much more complex, error-prone, difficult to maintain, and ultimately less robust. Layla provides a flexibility and ease of use that might have been impossible to achieve otherwise. The tutorial example mentioned above was implemented by a beginning programmer, yet unfamiliar with Layla, in about four days. Our own experience indicates that it would have taken him much longer, just to learn the workings of the API, had Layla not been there. We made similar observations with other NMIs, including the Xcoop example.

DESIGN AND IMPLEMENTATION OF LAYLA

In developing Layla, we were faced with a number of design decisions and trade-offs. Four of them are briefly discussed below. Then, we report on the implementation of the framework.

1. Framework functionality vs. simplicity. Quite a few commercial APIs contain capabilities not only for OSI network management, but also for Internet network management through the SNMP protocol. Some even let the users of the API supply their own network management protocol. We decided to provide as simple an interface as possible for CMIS. By gearing Layla solely toward OSI network management, we reduced the scope of the interface that is presented to the application. The application programmer only needs to learn a fairly focused interface, instead of the overly complex interface of certain APIs.

NMI	Specification Files		Implementation Files	
	Lines of Code in ASN.1	Lines of Code in GDMO	Number of C++ Classes	Lines of Code[a] in C++
Tutorial example	37	126	4	2,285
Xcoop example	253	635	15	8,128
Libraries (TMN, DMI)	963	3,822	*n/a*	*n/a*

Table 1: Size of specification files and implementation files for tutorial example, Xcoop example, and libraries**.**

a. comprises approximately 20% empty lines and 5% comment lines.

2. Performance vs. code readability and flexibility. We have resolved this trade-off almost always in favor of the latter. This means that we would rather distribute a behavior among a group of cooperating classes than put it into one big, monolithic class. An example is the Layla implementation of CMIS filters as collections of classes. The various types of filters are organized in subclasses following the structure imposed by CMIS. Each subclass is responsible for some specific filtering action, including composing `AND` and `OR` filters. The up side is that this organization allows the programmer to build arbitrarily complex filter constructs. The down side is that it adds some overhead to the processing of such constructs. In one instance, however, we opted for performance. Instead of splitting the various CMIP messages across multiple subclasses of some abstract class, they were all merged into one class `CMessage`. This class has a type-identifying field that specifies which kind of message is represented. A hierarchy of `CMessage` subclasses might be a cleaner design, but would probably lead to a greater performance loss.

3. Synchronous vs. asynchronous Remote Operation. In the synchronous form of the *Remote Operation* pattern, the invoker is suspended while his request is being processed, just like a regular procedure call. In the asynchronous form of the pattern, the invoker regains control immediately after his request has been sent. He must listen every once in a while to check if an

answer has been received. When *Remote Operation* is used as part of the *Manager-Agent* pattern, only the asynchronous form is acceptable, since a process must be ready to spontaneously handle events such as CMIP event notifications for managers and managed object notifications for agents. Version 1.1 of Layla only implements asynchronous *Remote Operation*. We are currently experimenting with a synchronous version so that applications eventually can choose between the two modes of operation and simplify their processing, in case they have no asynchronous constraints.

4. Native vs. generic attributes. Attributes in a given managed object class can be stored and retrieved by accessor methods in one of two ways. Either the accessor methods operate directly on prescribed areas of memory, such as internal variables (native), or they act on an internal collection of attribute values (generic). The use of native attributes is efficient in terms of both required development effort and CPU time. The programmer can reference variables directly using language identifiers, and the compiler will replace them by memory locations. Using generic attributes, on the other end, involves the application of insertion, search, and removal algorithms on the collection of attribute values, which requires greater knowledge from the developer and more CPU time. But once the generic attribute manipulation mechanisms have been installed, they do not need to be re-implemented anymore, whereas the native attribute accessors must be overridden by any subclass that defines attributes. Native attributes lead to a proliferation of similar looking accessor methods, which can be difficult to maintain. Some languages will let programmers define macros that are expanded by the compiler into full-fledged methods, keeping the performance to a maximum while minimizing the amount of code customization needed. The *Inherited Behavior* pattern can also help in this regard. This is the approach taken in Layla.

Layla has been implemented on HP workstations running *HP-UX*. Our development environment includes HP's C and C++ compilers, as well as HP's symbolic debugger *DDE*. An

initial version of Layla was implemented using AT&T's BaseWorX as the underlying API. Since Teleglobe was also interested in testing HP's DM, Layla was later modified to use that API rather than AT&T's. As the migration to DM was carried out at a relatively early stage in the Layla development, it is difficult to assess the exact effort required.

The current release of Layla, version 1.1, is the result of several design iterations, which is quite typical for pattern-based development [14]. We believe, however, that by now, the overall system of patterns is stable. Version 1.1 supports exclusively DM and required about two person-years to implement. It contains some 42 classes (plus assorted translation classes) and over 20,000 lines of code. We estimate that it would require about 3 person-months to re-adapt it to BaseWorX. Migration tasks would include the modification of all instances of the *Translator* pattern and some adjustments to the communication infrastructure. Of course, the applications and especially the managed object classes would have to be modified so as to handle the new C structures generated by BaseWorX's GDMO compiler.

Layla uses *Rogue Wave*'s *Tools.h++* class library to implement low-level lists and time-based types. All of Layla's code was put through *Pure Software* tools such as *Purify* and *PureCoverage* in order to test their integrity and plug potential memory leaks.

DISCUSSION AND PERSPECTIVES

So far, Layla has been used in a limited number of applications. A broader implementation base is needed in order to validate all the concepts that have been introduced and to truly prove the genericity of the framework. The elaboration of a number of example NMIs [17], however, gave the authors prime examples of the many issues involved, and experimentation at Teleglobe is providing us with further insight.

Layla is quite unique in offering a high-level, object-oriented interface to NMI development. At the time of writing, we are only aware of two other (object-oriented) frameworks for CMIS NMIs: *OSIMIS* and *JMAPI*. *OSIMIS* [10] is an API providing a C++ interface similar in scope to Layla. OSIMIS is being developed at the *University College of London* and is not commercially available. Our experience from using it in some other part of the IGLOO project indicates that it is not well maintained by industrial standards and that it is fairly complex to use, imposing on the programmer constraints that are much stricter than the ones found in Layla. *JMAPI* [8], developed by *Javasoft*, is a network management framework for Java applications. Its alpha version has been announced for May 1997. Unfortunately, its beta version, available since early 1997, is so unstable that we were not able to evaluate the system. The sample application provided by JMAPI suggests that the framework will provide strong support for the graphical user interface aspect of network management.

A likely reason for the scarcity of object-oriented frameworks for NMIs is the slowness with which object-oriented technologies are being adopted by the producers of software. As most software developers still use procedural languages such as C, a development platform in that paradigm has a better chance of penetrating the market than any other. But as workplace mentalities evolve, the demand for object-oriented platforms is likely to increase.

The current version of Layla supports the basic everyday work one would expect from an NMI. It does not yet cover though all of CMIS. Many of the less frequently used features of both CMIP and GDMO are not implemented yet. Such features include specific error reporting, though the most common errors such as *NoSuchInstance* are covered. GDMO allows for the description of attribute- and notification-specific errors that are currently being ignored by Layla. CMIS also allows distinguished names (DNs) to be taken into account for a specific context. Such DNs are

called *local DNs* and implicitly share a common prefix that is part of the context. The current DNs in Layla are all *global DNs* in that the whole list of relative DNs needs to be specified.

The modifications that would be needed for Layla to implement those features are numerous and extensive. It was therefore deemed more practical to focus on core functionality and to implement additional features in the next version of the framework. It is not expected though that these additions will change the framework architecture in any significant way.

In the medium term, we would like Layla to support NMI tools and standards other than CMIS. These plans might well get influenced by developments in the fast moving CORBA/Java area. Also, we will be interested in applying framework technology to certain network management aspects beyond the NMI domain.

Layla is conceived as a pattern-based application framework. Its development demonstrates that pattern-based frameworks can be built for the demanding NMI domain. Experimentation with Layla makes us believe that pattern benefits such as flexibility, reusability, and documentation value of the framework and the resulting applications can indeed be reaped. And we concur with Schmidt [14] that the "integration of design patterns together with frameworks" will be among the areas of increased attention in the years to come.

ACKNOWLEDGEMENTS

The authors wish to express their gratitude to Brahim ould Bah for implementing several examples, and to Teleglobe Canada Inc. for giving us access to their premises and equipments. Our thanks also go to our colleague Gregor von Bochmann, scientific leader of the IGLOO project, for his insightful comments during the course of our project.

REFERENCES

[1] AT&T Bell Laboratories. *BaseWorX Application Platform (AP): Application Management*

Reference Guide, 1994.

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, 1996.

[3] European ATM Pilot project. *Xcoop Interface Specification for the ATM Pilot*, May 1994.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] Hewlett Packard. *HP OpenView Distributed Management Developer's Guide*, September 1994.

[6] Hermann Hüni, Ralph Johnson, and Robert Engel. A framework for network protocol software. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'95)*, pages 358–369, Austin, TX, September 1995.

[7] ITU-T (Intl. Telecom. Union - Telecommunication Standardization Sector). *Recommendation M.3100 Generic Network Information Model*, 1992.

[8] Javasoft (Sun Microsystems Inc.), Mountain View, CA. *Java Management API*, May 1997. Alpha Release Documentation.

[9] D. E. McDysan and D. L. Spohn. *ATM: Theory and Application*. McGraw-Hill, 1995.

[10] George Pavlou, Graham Knight, Kevin McCarthy, and Saleem Bhatti. The OSIMIS platform: Making OSI management simple. In Adarshpal Sethi, Yves Raynaud, and Fabienne Faure-Vincent, editors, *Integrated Network Management IV*, pages 480–493. Chapman and Hall, 1995.

[11] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[12] Marshall T. Rose. *The Simple Book: An Introduction to Management of TCP/IP-based Internets*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[13] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 29, pages 529–545. Addison-Wesley, 1995. (Reviewed Proceedings of the First International Conference on Pattern Languages of Programming (PLoP'95), Monticello, IL, 1994).

[14] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, October 1995.

[15] Douglas C. Schmidt. A family of reusable design patterns for application-level gateways. *Theory and Practice of Object Systems*, 1996. John Wiley and Sons. Special Issue on Patterns and Pattern Languages.

[16] Adrian Tang and S. Scoggins. *Open Networking with OSI*. Prentice-Hall, Inc., 1992.

[17] Jean Tessier. An application framework for OSI network management interfaces. Master's thesis, Université de Montréal, Montreal, Quebec, Canada, April 1996. In French.

[18] Jean Tessier and Rudolf K. Keller. Manager-Agent and Remote Operation: Two key patterns for network management interfaces. In *Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences, Washington University Department of Computer Science, wucs-97-07*, pages 4.8.1–4.8.14, February 1997.

About the Authors

Jean Tessier is currently working for AT&T Labs as part of the Distributed Objects and Data Services group in Holmdel, NJ, USA. He got a Master's degree at the Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada, in 1996, with a thesis on framework design for OSI network management interfaces. He collaborated with Teleglobe Canada Inc. from 1993 to 1995 to perform network management of message handling systems and of ATM switching systems.

Rudolf K. Keller received a Diploma degree in mathematics from the Swiss Federal Institute of Technology (ETH) Zürich in 1983, and a Ph.D. degree in computer science from University of Zürich, Switzerland, in 1989. He was a postdoctoral fellow at University of California at Irvine from 1989 to 1991. From 1991 to 1994, he was a researcher at the CRIM research institute in Montreal. Since 1994, he is an assistant professor at the Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada. His current research interests are object-oriented design, framework design and documentation, component-based software development, real-time systems, and user interface engineering.