Adaptability Using Reflection

Susann Sonntag, Hermann Härtig, Oliver Kowalski, Winfried Kühnhauser, Wolfgang Lux
GMD, Gesellschaft für Mathematik und Datenverarbeitung mbH
Schloss Birlinghoven
53757 Sankt Augustin, Germany
sonntag,haertig,kow,kuehnhsr,lux@gmd.de

Abstract

Adaptability, i.e. the ability of a system to adapt dynamically to changes in its execution environment, is considered as an important property of computer systems. Scaling directory replication in name servers and load balancing are well known examples. However, adaptability in today's systems, if present at all, generally is burnt in and dedicated to singular properties of the execution environment.

This paper discusses a more universal approach to adaptability. The approach is based on the — in some cases artificial — separation of nonfunctional properties from the functionality of application systems. The scheme provides full control over non functional properties while preserving transparency for the application programs.

To study that approach, reflection has been incorporated into the BirliX object model and its implementation. It has been used to experiment with some non functional properties such as migration and protection.

The paper motivates the need for a universal adaptability scheme, revisits reflection as used in programming language systems and discusses design decisions for reflective systems, describes the resulting reflective BirliX model and its implementation and finally describes applications.

1 The need for adaptability

The starting point for our work is the observation that many of today's more sophisticated applications have a common problem exploiting the power of distributed systems. They need to dynamically adapt the underlying system to their special needs to meet requirements in performance, security, or fault tolerance. A few examples are:

- Mobile computers using phone lines, radio, or fiber optics for communication need to adapt to various bandwidths using appropriate protocols [3].
- Frequently used services (e.g. file services, name services) have to scale to the number of their clients. These services can be adapted to growing networks by a growing scale of service replication [4].
- Recent work [5] indicates that migration can be a very powerful mechanism if it is not restricted to just processes or files. It has to be easily adaptable to various types of objects and to a wide variety of purposes.
- Auditing is employed in some systems for intrusion detection. The detection of suspect events causes systems to switch to a security-alert state. Here adaptation can be used to tailor the system entities to provide more detailed auditing information, e.g. by activating several additional auditing sensors. [6] gives more examples for security policies using adaptable object behaviour.
- Multimedia systems must adapt to changing environments. E.g. once the error rate of a communication channel increases during a video conference, it is desirable to employ lower resolution protocols and use the extra bandwidth for error correction.

For these and other examples adaptability can be provided using two different techniques:

1. application level adaptation

Application specific solutions are integrated into an application. The basic system supports generic primitives. This leads to a duplication of effort and makes it hard to incorporate new technology/algorithms/idea etc.
2. system level adaptation

Adaptation solutions are integrated into the basic system. These solutions have to be sufficiently universal to support a wide range of requirements. That leads to partial loss of control over the algorithms etc. employed in the adaptation, i.e. the algorithms cannot be tailored to the application.

To overcome the drawbacks of these two ways our approach provides adaptability by controlling the properties of objects. In our computing model applications are sets of interacting objects. The functionality of an object is application specific and defined by methods within a type description. Additionally, each object gets a set of system defined properties like persistence, migratability, protection and others. All the system defined properties determine the object model of the system. The so called infrastructure is the component of the basic system which puts the object model into effect. This infrastructure which determines the properties of objects has to be modifiable to make systems dynamically adaptable. It has to put into effect individual object models for each object in contrast to determining one system wide object model.

This leads to a new programming model. Application programming is independent of infrastructural properties of application programs. Infrastructural properties for objects used by application systems are determined during configuration or reconfiguration of these systems. That leads to a higher degree of reusability of applications as well as infrastructural programs.

A mechanism to materialize our scheme is reflection as found in programming languages. There reflection is used to determine the semantics of special language constructs and to define the way objects are built. To study the feasibility of the approach and its usability we incorporated reflective features into the BirliX operating system [7]. So we are able to exchange and manipulate infrastructural properties of BirliX objects at runtime.

The ability to virtually replace the execution environment of an application obviously causes virulent security threats. To overcome these threats formed a major part in the work to build an adaptable system.

In this paper we first describe reflection in general terms. Thereafter, we outline the infrastructural model. Then, we describe the BirliX system and show, how reflection has been added rather straightforward to the existing operating system BirliX. We used the implemented features for adaptation of migration policies and for security purposes. These two examples are shown. According to the experiences with our application examples we discuss results and consequences of our prototypical implementation. The last section discusses related work.

2 Reflection

Reflection originates from programming languages: "Computational reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation. We define computational reflection to be the behaviour exhibited by a reflective system, where a reflective system is a computational system which is about itself in a causally connected way." [8].

"Causally connected means that any changes made to a process self-representation are immediately reflected in its actual state and behaviour" as defined in [9]. To achieve this the basic level is represented at a so called meta level by reliable data. We specify objects according to their position of a particular level. Objects on the meta level are called meta objects, objects on the basic level are called basic objects.

In programming languages reflection allows the programmer to determine the way entities described in the language are created, initialized, accessed and destroyed. Reflection offers the ability to adapt these entities to application requirements in a language specific sense.

Reflection has first (for the authors' knowledge) been used in Smalltalk-80 [10]. The implementor of the language system can define how an object will be constructed. The Smalltalk-80 programmer uses this definition implicitly by specifying the key word class. The CLOS [11] programmer is given constructs to define and use meta classes to change the semantics of language constructs. 3-KRS [12] in addition, provides access to the reflective facilities at runtime. It is an interpreter system.

The designer of a reflective system faces a number of questions [13]:

- What should be reflected upon?
  Which part of an object should/could be changed by an operating system?

- How is the causal connection maintained?
  How is the relationship between objects on the basic level and on the meta level represented?

- When does the system shift up to a meta level?
  When is a method of an object acting as a meta object called?
Although the reflection techniques of programming languages cannot be directly transferred to object models provided by operating systems, the concept of reflection promises a good approach to provide universal adaptability. The idea to use reflection in operating systems is not new ([14]). Recently reflective facilities have been incorporated in operating system design with several goals (e.g. for accessing and choosing several components of an operating system in an easier way, for the support of multilingual environments or for offering different abstractions). These goals do not meet our requirements for protectable systems and adaptability of infrastructures to application specific environments. Therefore we propose a different reflective architecture for the BiiX operating system.

3 The Reflective Object Model

Our starting point was the observation that sophisticated applications very often have difficulties to make efficient use of the power of distributed systems. The examples made clear that one significant problem is the incapability of the underlying systems to adapt smoothly to specific needs of the applications. Our belief is that integrating the concept of reflection into the object model of a system is an adequate way to provide a homogeneous paradigm for adaptation.

Objects in any object model have **functional and non-functional** properties. Functional properties are defined by the *type* of an object, non-functional properties are defined by the object’s execution environment. Execution environments are defined e.g. by the semantics of object oriented programming languages, or by operating systems with object oriented application interfaces. As examples, a non-functional property of a programming language object is the inheritance scheme, non-functional properties of operating system objects are persistence, audit, distribution etc.. We will refer to those components that implement the non-functional properties of the system’s object model as the system’s *object infrastructure*. In a universal computing model where any application system is a set of interacting objects a significant part of the application’s execution environment is provided by the system's object infrastructure.

Our concept of a reflective object model is based on a rigorous separation of object and infrastructure. In terms of chapter 2, object and infrastructure form a reflective system and are causally connected: any changes made to the object's representation within the infrastructure (meta level) are immediately reflected in the object’s actual state or behavior (basic level). Operations at the meta level are called either by calling the methods of the infrastructure object explicitly or by calling methods of the object that are implemented by methods of a meta object of the called object.

By modifying the infrastructure, we modify the non-functional properties of the object model without touching an object’s functional properties. By interpreting the term *infrastructure* as an individual entity glued to each individual object, an individual infrastructure is associated to each object.

The adaptability of the infrastructure of each individual object is achieved by implementing the infrastructure via a set of *meta objects*. Thus adaptability of the non-functional properties of an object is achieved in our model via substituting its meta objects.

Separating of object and infrastructure leads to a simplification of the object model. How does the simpler object model, the basic system has to support, look like? We call it the *basic object model*. There are two essential aspects of the basic object model:

- **minimality**
  The basic object model describes objects in the usual sense of the object oriented programming paradigm as encapsulations of data and code. It includes methods for creating, deleting and calling methods of objects.

- **reflectivity**
  The basic object model includes the possibility of connecting meta objects to an object and dynamically modifying such connections to build up appropriate properties for this object.

The basic object model is put into effect by the so called *basic infrastructure* provided by the basic system. Each object created using a type definition is built using this implementation structure. Therefore it is characterized as follows:

- Each object is an object in the usual sense of object oriented programming.
- Each object maintains causal connections to its meta objects (possibly none) supporting other properties.
- Each object can be an infrastructure object, provided that it supports methods for evaluating method calls forwarded from other objects.

This means that each object is provided with the ability to maintain its causal connection to its meta objects. Therefore it is easy for a given object to find a meta object. Methods for the maintenance of the
causal connections are accessible via the interface of the object. An object itself is able to manipulate its infrastructure. For doing this the object has to call its own method for establishing a connection to a meta object via its interface.

Each object is able to act both as an object and as a meta object if it supports methods for handling forwarded method calls. From the system’s point of view it is not possible to determine whether an object acts as an object or as a meta object. The only requirement for a meta object is that it implements methods for evaluating messages forwarded from other objects. But an object will be a meta object only if it is connected with an object to provide it with new properties. Such a connected object itself may have another infrastructure object or it may implement reflective structures internally.

Connecting objects to other objects establishes an “object - meta object” relation between them. Such relations are valid exactly for two particular objects. For users such relations are invisible. The location of an object in the meta hierarchy is relative. From the view point of different objects it is possible to find one meta object on different meta levels. So it is possible to build optional meta structures, not only hierarchies. Thereby the infinite tower of reflection can be terminated by objects which do not have a meta object.

4 Reflection in BirliX

Now, we will have a look at the initial BirliX object model and will incorporate the new approach into it. Thereafter, we will show some applications of the implemented concept.

4.1 The Initial BirliX Object Model

The BirliX system ([7]) is basically a BirliX Type management system with mechanisms for the definition and the instantiation of types, and the communication of instances. BirliX Types share a common set of functional and non-functional attributes inherited from the system-defined PrimaryType.

The BirliX Type paradigm provides the smallest unit of identification and communication, the Type Instance. Application programs on BirliX are sets of interacting instances distributed among several nodes in a computer network. Instances are identified and located by name servers on top of the kernel.

BirliX Types have type descriptions that contain the type interface (signature) and the methods (code of exported operations).

BirliX Type Instances communicate by invoking methods. There is no other way for accessing an instance. The underlying communication mechanism is a network-transparent remote procedure call (RPC) that — for reasons of efficiency, security and fault tolerance — maintains a client/agent relationship between caller and callee.

Each instance has a unique identifier; the mapping of high-level user names to unique identifiers is done by name servers which themselves are Type Instances. Unique identifiers are independent of instance attributes like type or location. Location information is provided by name servers as hint. The establishment of a client/agent communication connection (binding) is performed by the methods of the PrimaryType within the kernel.

Instances can be active on their own; they may contain threads that run in parallel to each other within the same instance.

Instances are persistent. They continue to live as long as they are referenced, e. g. by a name server or a binding from another instance. When neither a binding nor internal activities exist, an instance releases processor and main memory resources; a passive representation is created and written to permanent storage. Any new binding to a passive instance reactivates the instance. Activation and passivation is done by methods defined within the PrimaryType.

Instances are not protected against system crashes, but they provide checkpoint and recover methods that create checkpoints (additional passive representations) and reactivate instances from checkpoints. Passive representations can be sent to other locations. Thus, checkpoint and recover primitives are also used as the basic primitives for type-independent instance migration.

Instance security is based on access control lists, subject restriction lists, and authentic messages ([15]). Access control lists protect individual instance operations, subject restriction lists limit the accessible world for individual callers, and authentic messages guarantee unforgeable caller identification.

Based on these mechanisms, server-level policies like replication, transactions and migration are implemented.

During every day use and the usage as a research vehicle some observations [16] concerning the PrimaryType were made. To carry along all PrimaryType methods is too expensive. Experiments on security ([6]), migration ([5]) and replication ([17]) in the BirliX operating system show that the mechanisms implemented within the PrimaryType are in-
flexible. The scheme that each instance inherits every method defined within the PrimaryType has two major drawbacks. First, the scheme burdens all instances with a significant overhead that might not be justified (only few instances actually make use of the checkpoint/recovery or migration methods). Second, a method defined within the PrimaryType is type-independent; it provides only the generic part of a method that is adapted to the type-dependent needs at the point when a method is inherited. This static form of method adaptation has the disadvantage that any application-dependent adaptation of an instance to an environment (e.g., a new recovery policy) has to be done during the definition of the used types.

4.2 Reflection in the BirliX Object Model

The main question during the interpretation of our reflective model concerns the separation of object and infrastructure. Using the initial BirliX object model we observed that there are non-functional properties which are used via method call (e.g., migration, checkpointing). They appear as functional properties. So we have to draw up a borderline and have to define which property we treat as non-functional and which property we treat as functional. We put up this borderline along the initial PrimaryType and treat all application-defined methods as functional properties.

But at first let us have a look at different presentations of objects (or instances; we use these terms synonymous) for a better understanding of the following material.

While objects in programming language systems are described as encapsulation of data and code providing an interface (see figure 1) system supported objects, like instances in BirliX, may be expressed as objects surrounded by infrastructural properties (see figure 2).

![Figure 1: The interface model of a programming language object](image)

Embedding objects into infrastructures is performed by the basic system. This static binding of infrastructural properties to the functionality of an object prevents the manipulation of the object infrastructure. Removing this static binding and introducing explicit methods for constructing and manipulat-

![Figure 2: A system supported object surrounded by its infrastructure](image)

ing the infrastructure enables the adaptability of the object infrastructure.

In case of the BirliX system that means: remove all non-essential properties from the PrimaryType and introduce reflective features into the PrimaryType. The new PrimaryType includes:

- **essential components**
  such as creating, passivating, deleting of instances as well as receiving and handling messages,

- **reflective components**
  such as manipulating of causal connections to meta objects as well as forwarding of messages according to present causal connections.

According to the presentation of objects in figures 1 and 2 the objects built up using the new PrimaryType may be shown as in figure 3.

![Figure 3: An object with reflective facilities](image)

The exchange of meta objects as parts of the infrastructure means at least the exchange of algorithms. During exchanging algorithms generally two security relevant aspects are observed:

- Exchanging the algorithms has to be controlled according to the security policy of the system.

- Exchanged algorithms have to be unable to destroy data structures or algorithms of other enti-
The implementation of reflective features in BirliX considers these aspects in a straightforward manner. Access to reflective features is only possible by calling the appropriate method. Like all methods in the BirliX system this method call is protectable. The method implementing the exchange of infrastructural properties is realized in the system kernel and therefore it is unchangeable.

By handling messages we are able to manipulate the execution of methods. So we can define special properties of objects. According to handling messages BirliX method execution consists essentially of three parts (see figure 4) implemented by the PrimaryType:

- receiving messages,
- handling messages (executing the requested operation), and
- sending messages.

Figure 4: The three parts of method execution in BirliX

The PrimaryType guarantees the forwarding of messages to bound meta objects at these points. Meta objects for receiving or sending messages are incorporated into the message flow. They act as filters (called Infilt or Outfilter). Messages may be manipulated in the filters. Upon returning a forwarded message to the PrimaryType they are able to indicate whether this message is allowed to pass or not. This gives us the possibility to use filters for several security purposes too.

Meta objects for handling messages execute a requested meta operation. They are called by the PrimaryType instead of executing an objects own operation concerning the infrastructure. Therefore we call them Infra meta objects. Figure 5 points out the meta object hooks.

Figure 5: A BirliX object with reflective features

Infilt and Outfilt offer changes in the semantics of the message passing (additional encryption algorithms, a concrete protection policy, broadcast sending of messages and so on) and Infra offers additional methods concerning infrastructural properties as migration, checkpointing, recovering.

Implementing meta objects we observed some problems. Infilt and Outfilt may need type specific information for the evaluation of special parameters of the message. Such information may be used during the implementation of the appropriate meta object or the meta object is able to get these information from an abstract type description of the destination object.

The implementation of Infra requires additionally access to the data representation of the supported object. BirliX objects are built out of passive (memory) and active (processor) resources. The state of these resources is represented in segments, threads, stacks, tables, etc. Infras need methods to access these representations to get object specific information and to manipulate these representations. The PrimaryType has to deliver an appropriate interface.

The problem is to find an interface with basic methods which are appropriate for all possible meta objects. Experimenting with adaptable migration policies we got such an interface and introduced it into the BirliX PrimaryType:

- SuspendCommunication — suspends communication with other instances,
- SuspendInstance — suspends internal activities of the instance,
- CollectKernelState — collects the kernel state of the instance, which is distributed in several system tables, in one data structure,
**SendMessage** — sends a message to the destination host, used to transfer the instance state successively.

**ReleaseResources** — releases all resources not needed more at the source host.

**UpdateLocation** — informs the source host that there is a hint to the new location of the instance.

**AnnounceInstances** — announces the the instance to the destination host.

**ReceiveMessage** — receives a message at the destination host which is sent from the source host.

**DistributeKernelState** — distributes the kernel state of the instance to the system table on the destination node.

**ResumeInstance** — resumes all activities of the instance.

**ResumeCommunication** — resumes communication with other instances.

These methods can be divided into three groups:
1. methods to access passive resources as sending, receiving, duplicating, deleting memory resources,
2. methods to access active resources as suspending, resuming activities, and
3. special methods depending on the concrete basic object model and on the adaptability problem as collecting/updating kernel states, garbage collection after sending a whole representation of an object.

Treating the PrimaryType as a common part of each object these services are methods of it. The execution of these operations is protectable in the same way as the access to all other methods. Other methods delivering specific information about the state of attributes of an object may be included into the type definition by a so called type generator. In the next chapter we show, how a migration policy can be implemented using reflective features and these additional system services.

### 4.3 Some applications

The following discussions of application sceneries show the usability of the proposed concept. Two examples were chosen. Outlining these examples advantages and disadvantages of the implemented reflective features are described.

We discuss a solution to the migration problem outlined in [5] by using the reflective features.

With the incorporation of reflective features into the BirliX operating system we are able to extent the security architecture of the system. We discuss how these features may be used to change the semantics of the underlying communication protocol.

#### 4.3.1 Migration

For migration two problems have to be addressed:
1. How is the data representation of the object moved?
2. When is a migration initiated?

With the ability to connect several meta objects to a basic object these two problems may be solved independently.

Migration as an infrastructural property is an operation upon the representation of an object. The mechanism supporting migration has to be implemented in an *Infra*. The request to migrate the object is forwarded to the bound *Infra*. The migration *Infra* collects the necessary object specific information, suspends all activities within the appropriate object and sends all memory resources to the destination node. There another migration *Infra* receives the segments and resumes the activities. For the usage of the correct migration *Infra* on the destination node the basic system of the destination node has to take care.

To support a load balancing policy it is possible to incorporate an *Infilter* into the message flow of several objects. This *Infilter* may collect information about source and destination of communication, about the amount of communication in a specified time interval etc. Depending on the collected information this *Infilter* may initiate the migration of a chosen object.

By exchanging one or both of these described meta objects we are able to manipulate the used load balancing policy. By exchanging *Infilter* the manner how the object will be migrated is manipulated. Several migrating mechanisms for different purposes are known. By exchanging *Infilter* we can manipulate the decision when the object will be migrated depending on incoming messages. The exchange of one of these meta objects is independent of the exchange of the other meta object. Therefore several combinations are possible with only a few meta objects. The type definitions for these meta objects are reusable in different contexts.

#### 4.3.2 Custodians within the BirliX security architecture

The ability to incorporate meta objects into the message flow of objects is very important for the security architecture of BirliX systems. The BirliX security architecture ([6]) is based upon the control of messages via filters hooked into the message flow between
two communicating instances. Filters give the possibility to control messages in accordance with different requirements. By changing filters it is possible to exchange or manipulate the used protection policy according to a global security policy.

Reducing the functionality of filters to watching over messages we get the described concept of custodians ([6]). Using appropriate implementations of these custodians the system will be mandatory protectable.

Using Infilter or Outfilter, the support of many security policies is possible by security mechanisms like encryption/decryption or access control lists. More intelligent filters even may contain full control policies like Bell-LaPadula ([18]) or shared control policies. By binding filter meta objects with filter meta objects a combination of different security policies is possible.

4.4 Results

We included reflective features into the BirliX PrimaryType to give the objects the ability to construct/reconstruct their infrastructure dynamically. The implemented mechanisms are homogeneous to the whole system concept and are governed by the BirliX security architecture.

With our prototypical implementation we designed some demonstrations. These scenarios of applications (see section 4.3) show both the usability and the limitations of our prototypical implementation. They outline the need of future work:

1. The granularity of adaptability depends on the basic object model supported by the basic system. Our prototypical implementation is based on the initial PrimaryType of the BirliX system. It offers coarse grained properties of objects. So we are able to adapt these coarse grained properties. Changing the RPC communication protocol or the algorithm for locking an instance is still impossible. Offering finer grained properties by the basic object model leads to finer adaptability. A redesign of the BirliX PrimaryType according to this discussion seems necessary.

2. Some meta objects, implementing infrastructure properties, need additional information. E.g. replication strategies need type specific information about the replicas (which attributes are relevant for the consistency check, which methods change attributes and so on). Without type definitions delivering these information dynamically at runtime the meta objects for such strategies will implemented according to a concrete type of replicas. They cannot be reused for replication of objects of other types. A tool generating abstract type definitions in conjunction with the type description used for creating objects is necessary to avoid this disadvantage.

3. Other meta objects, e.g. implementing migration mechanisms, need object specific information about the object being migrated (location and length of stack, data and code areas, activities and so on). The objects have to deliver these information by supporting special methods. By adapting the BirliX type generator we will be able to generate these methods automatically and include them into the type description.

4. Our prototypical implementation is based upon the initial BirliX PrimaryType. The disadvantages, especially concerning performance constraints (see [16]), were not removed. So one method call to an object with one Infilter takes twice an initial method call. The same result is achieved, if the sender object is supported by an Outfilter. The support by an Infra does not concern the method call, except a meta operation is called. With a redesign of the BirliX PrimaryType and using faster mechanisms for communication and changes of address space we hopefully will get better performance.

5 Related Work

Many projects consider the adaptability of a system with respect to particular properties, e.g. replication of files for higher availability ([19, 20]) or migration of processes for load balancing ([21, 22]). We are looking for a universal solution for the problem of adaptability. Therefore we do not discuss these examples here.

An example of an operating system which has been designed using a reflective architecture is the Apertos system ([23, 14]). The main motivation of the design of the Apertos system was to support mobile computing in a multilingual environment and to design an operating system with a reflective architecture. The kernel of Apertos, called MuseCore, provides primitives for the control of the causal connection links. Each object is able to change its environment by using MuseCore primitives. With an extension of the MuseCore it would be possible to add a protection mechanism on which the enforcement of protection requirements could be based.

The main motivation of Apertos is to support a possible execution environment for an object. This is done according to the current system state, the underlying hardware and the interactions with other objects. It does not fully depend on the requirements of the application.
In Apertos reflection is implemented as a new system structuring concept. It is not governed by the object oriented paradigm supported for all Apertos entities. Therefore another reflective architecture for objects based on the BirliX operating system was proposed in this paper.

In the L3 system [24] Chiefs and Clans are implemented as a protection mechanism. We are able to emulate Chiefs and Clans using the implemented reflective features in the BirliX system. The resulting structure of the emulation has something in common with the structure of Chiefs and Clans in L3.

Some projects try to support different programming language objects (COOL-2 [25]. COMANDOS [26]). Thereby different object models on user level are mapped to one object model on operating system level automatically. For this automatic mapping the projects provide several solutions like generic runtime layers or upcall tables. These solutions offer the ability for multilingual environments, but they do not consider the adaptability of the operating system. Combining these solutions with the adaptable object model it is possible to achieve both support for multilingual environments and adaptability.

There are other projects (e.g. Guide [27] or Space [28]) providing basic abstractions of the underlying hardware that offer the ability to construct optional system architectures, object oriented and others. The decision for a concrete abstraction has to be done before using it. Changes at runtime are not possible.

6 Conclusions

Adaptability is considered as an important property of future computer systems, as the ability to exchange the execution environment of a system entity. We incorporated reflective features into the BirliX operating system to support universal adaptability for all objects independent of their type. That resulted in a new programming model by the separation of functional and infrastructural properties. Infrastructural properties need not be considered during the implementation of applications. They are determined late during the configuration or reconfiguration of applications.

Some real life applications show the usability of our approach. However, experiences in implementing some examples point out that more sophisticated tools and better performance are essential to promote our ideas. Thus tools and performance will guide our future work on the BirliX operating system.

References


