A Language-Based Approach to Construct Structured and Efficient Object-Based Distributed Systems

Markus Pizka and Claudia Eckert
Munich University of Technology
Department of Computer Science
80290 Munich, Germany
{pizka,eckertc}@informatik.tu-muenchen.de

— This project is sponsored by the DFG (German Research Council) as part of the SFB #342 —

Abstract

Classical object properties such as encapsulation ease the construction of distributed systems. The object paradigm supports modeling of real world problems in a natural way and delivers units of distribution to the resource management level. To enhance the performance of distributed systems, more detailed application-specific information like potential communication dependencies should be exploited.

To fulfill this requirement, we propose a top-down driven, language-based approach to construct structured distributed systems. We introduce the object-based distributed programming language INSEL, that supports advanced structuring concepts. Structural dependencies between passive and active objects are determined at the application level and exploited by the resource management system to transform INSEL programs into efficient executables.

1. Introduction

Object technology has gained wide acceptance in the community of software developers constructing large-scale application systems as well as operating systems (OSs). Object-orientation supports component reuse, object interaction via well-defined interfaces, inheritance and encapsulation to name only a few important features. Current research focuses on the development and efficient realization of distributed object-based systems to keep pace with the rapid changes in hardware technology. Networks of workstations (NOWs) together with high-speed interconnections like ATM offer great amounts of storage and computing resources. Great efforts are still needed to cope with the challenging demands coming along with the transit from centralized to distributed computing. The most important issues and requirements concern two main areas: providing appropriate programming environments and performing efficient resource management.

From the program developers point of view, a programming environment is required, that offers concepts, features and tools to simplify the development of large-scale distributed applications. This enables application developers to use the available computing power in a comfortable and easy way. The developer should not be bothered dealing explicitly with any realization details such as using socket addresses or port numbers of services to realize distributed object communication, binding object names or keeping track of current load distribution. Details of the underlying hardware configuration and the OS being used should be hidden for programmers. Moreover, application developers should not be forced to handle new concepts (e.g. different RPC semantics or object naming schemes) and tools (e.g. interface definition languages) coming along with execution environments. Hence, homogeneity of concepts offered by the programming environment is required. Interoperability between application objects should be supported in a transparent way. To cope with the complexity of large-scale software products, structuring features are essential. Facilities like class hierarchies, that are part of object-orientated programming environments typically use flat object hierarchies that are insufficient and need to be enhanced.

For the user of a distributed application, efficient execution is the predominant requirement. Complex distributed applications like banking systems, massive parallel computing or multi-media applications occupy large amounts of system resources and claim individual OS support like security, fast context switching or real-time processing. Therefore, execution environments are needed, offering distributed object management services that can be customized to application-specific requirements and that are able to dynamically adapt their policies, depending on changes in

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

overall resource utilization and application-specific needs. Advances in the area of global resource management are strongly needed to ensure efficient execution of large-scale distributed applications, demanding huge amounts of resources. At the OS level, objects serve as units for resource management. We state that further structuring of the set of objects enables better global resource management decisions.

The remainder of the paper is organized as follows. Section 2 investigates the benefits and deficiencies of existing distributed object-based systems ranging from well-known computing environments like OSF/DCE [17] and OMG/CORBA [16] to object-oriented distributed OSs. Section 3 explains the fundamentals of our top-down driven approach that aims to overcome some of the revealed deficiencies by providing new language and structuring concepts which are introduced in Section 4 and 5. Our resource management system, is presented in Section 6. Considerations regarding interoperability are presented in Section 7. Section 8 plots the current status of our project, before we give some concluding remarks in Section 9.

2. Related Work

The aim of this section is to investigate existing distributed environments and platforms and to elaborate their benefits and deficiencies with respect to the requirements stated above. From the lessons learned, we will derive our proposed solution, which is heavily based on using structuring features in a systematic way to overcome some of the shortcomings detected.

2.1. Comfortable Programming Environments

Computing environments for distributed applications like OSF/DCE [17] or ANSA [2] offer tools and services to support a procedural programming paradigm providing a RPC mechanism. Products based on the Common Object Request Broker Architecture (CORBA) [16] specification of the Object Management Group (OMG) aim to support the development and integration of object-oriented software in heterogeneous environments, emphasizing interoperability of application-level objects as well as reuse of components.

Benefits: The construction of client-server-style applications benefits from these computing environments because client-side code can deal basically with application-specific issues rather than with low-level mechanisms, like socket addresses and TCP/IP details. Facilities like late binding as well as separation of interface specification and object implementation in CORBA reduces the efforts needed to extend and adapt existing applications to changing functional requirements. Undoubted, CORBA and OODCE mark important milestones on the way to comfortable and simplified programming of distributed systems.

Deficiencies: Programming within these environments is not as easy as it should be. New concepts are introduced by each of these environments. CORBA introduces, for instance, the notion of object references or an exception handling feature, whereas DCE introduces, for example, a thread concept to enhance passive entities with an activity. Synchronization problems stemming from these enhancements must be solved by the programmer himself. In addition, the software developer is burdened with name servers (e.g. traders in ANSA or directory servers in OSF/DCE) to search available services or to register own services. He has to cope with interface definition languages (e.g. DCE-IDL, OMG-IDL) to specify interfaces. Remote object accesses must be handled in a different way than local onces, by first binding client-stubs.

Hence, these distributed computing environments still load a heavy burden on the programmer. Due to the heterogeneity of the concepts, he has to spend a considerable amount of time to learn how to handle these concepts and tools correctly and how to combine them as far as possible with his well-known programming language concepts and paradigms. Heterogeneity seems to be an unwanted source for programming errors that could be avoided by supporting a conceptual homogeneous environment. Such an environment should offer adequate object models and programming paradigms as well as structuring concepts to cope with the complexity of large-scale applications.

2.2. Efficient Resource Management

Performance issues with respect to efficient resource management have not been a major issue in the design and implementation of the environments mentioned above. Consider the CORBA environment. Transparent resource management is the task of the Object Request Brokers (ORBs) but actually existing ORBs limit their services to locating objects and marshaling parameters. If, for instance, an ORB is able to exploit application-specific informations, like access patterns, the realization of these accesses might be optimized using the available resources more efficiently. As an efficient resource management is the intrinsic task of an operating system it seems worth to have a look at current research activities in this area.

Distributed Object-Oriented OSs: Object-oriented technology enables customizing OSs to application needs at a pre-execution time (e.g. Choices [4]) but an application-specific adapted resource management requires greater efforts. Work performed with so called reflective architectures (e.g. Apertos [23]) show a promising way: the OS offers different policies to perform resource management, for instance different scheduling algorithms. Each application

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

is enabled to select a policy which properly matches its specific needs. Based on the reflection facility, OS and applications may interact to adapt resource management policies dynamically. To come to proper decisions, appropriate information about the application is needed. Such information can be gained by analyzing structural dependencies between application-level objects. But unfortunately, due to the lack of structuring facilities of the underlying programming languages the potential to perform analysis and efficient application-specific resource management can not be exhausted within these approaches.

To summarize, a programming environment is required that offers appropriate concepts to develop well-structured distributed systems on a high-level of abstraction. In addition, the structural dependencies must be exploited by the underlying execution environment, for instance by ORBs in the CORBA context. To meet these requirements we propose a top-down, language-based approach.

2.3. Other Language Based-Approaches

Several well known research projects, such as COMANDOS [6], GUIDE [18] and ORCA [3] — to name only few — tried to provide homogeneous distributed programming environments by following a language-based approach. Projects like COMANDOS and GUIDE lack support for parallelism at the language level. Due to additional compatibility issues the programmer has to use general purpose OS services that hardly match language requirements.

With ORCA major steps towards application-specific resource management were taken. Tools, such as the compiler are tailored to ORCA and enable static and dynamic analyzes. These analyzes are used by the resource management system to optimize accesses to shared objects. Unfortunately, to ease these tasks, the language is burdened with restrictions like missing support for pointers.

3. Fundamentals of the MoDiS Approach

The acronym MoDiS stands for Model oriented Distributed Systems and emphasizes that abstract concepts and models [11] are the foundation of our project. It is best characterized as a top-down driven, language and object-based approach to develop distributed systems.

3.1. Bottom-Up Vs. Top-Down Orientation

Bottom-up constructed application programming interfaces inherently suffer from inadequate services offered to the application level. This is due to the fact that bottom-up construction of OS services and programming interfaces aims to enhance the simple functionality provided by the hardware to more powerful services offered to the application level, with little consideration of concrete requirements of applications or properties of programming languages. This usually leads to general purpose resource management features and some general basic services, which are not matched to application needs. For example, UNIX systems offer heavy-weight processes to the application level, which are inadequate to realize fine-grain parallelism (in this paper we do not differentiate between concurrency and parallelism). To meet the demand of more flexibility, systems like Mach [10] introduced threads. But again, this is another fixed and general purpose abstraction lacking flexibility. For example, in Mach 3.0 it is not possible to create a really light-weight activity that only executes a short computation in parallel without having the overhead of a relatively large fixed size stack portion and a predefined port name space for communication.

To overcome theses deficiencies we follow a **top-down** oriented approach, deriving low-level facilities from requirements of the application level. Within the MoDiS project, we developed abstract concepts to construct structured distributed systems. Top-down orientation in our sense means, that the construction of a distributed system starts with the specification of the system at a high level of abstraction, using our programming language INSEL, which provides language concepts that are well adapted to our abstract concepts. A system specified in this manner consists of a structured set of objects with conceptionally well defined properties. The structures within the system describe the dependencies between the different objects of the system.

Realization of such a system on a given hardware platform is done by stepwise refinement of the abstract properties towards more concrete ones. With each transition from one level of abstraction n to a lower level of abstraction $n-1$, mappings from more abstract properties to concreter ones have to be found [12]. The potentialities of the target abstraction levels determine different realization alternatives for these transitions. For example, to realize the creation of a large passive object with a huge amount of data, at certain levels of abstraction, the resource management system has to choose between different alternatives, as e.g. storing in local or remote main or secondary memory or maybe even a mixed solution. Each of these different realization techniques would have advantages and disadvantages, depending on the context in which they are used. Since the process of realization started with an abstract model of the system consisting of abstract objects and structural dependencies, these structures can now be used by the resource management to come to appropriate decisions.

Some of these transformation steps are performed at compile-time, others have to be done dynamically at runtime. Hence, the process of refinement encloses the complete life-cycle of a distributed system. This is a signifi-

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

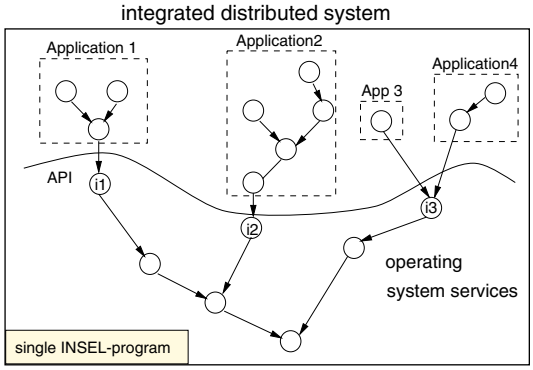

Figure 1. Holistic view of a distributed system

cant difference to common systems which for instance often separate compilation from lower levels of resource management (e.g. scheduling decisions). By tightly coupling these steps of transformations we are able to preserve application-specific informations for all levels of abstraction to enable efficient resource management.

The process of transformation ends at a very low level of abstraction where only rudimentary functionality is needed to fulfill the requirements. In our project, these low-level services are provided by a new micro-kernel, called *DYCOS* [7]. It fits into our top-down approach, by not imposing any major restrictions on the utilization of physical resources, but only providing a more comfortable interface to computing, communication and storage resources than bare hardware does.

Besides efficient resource management, our top-down approach eases the task of programming distributed systems, since the programmer specifies the distributed system at a high level of abstraction with a homogeneous repertoire of language concepts. The programmer does not have to cope with the details of the physical realization such as deciding about the placement of a specific object or explicitly using OS services like threads and semaphores.

3.2. Language-Based Approach

We combine top-down orientation with a language-based approach that leads to a single system spanning OS functionalities and user-level applications. With the notion of *'system'* we refer to a structured set of objects that realizes OS functionalities and user-level applications. Figure 1 illustrates this integrated view of a distributed system. Applications hook into the running system by connecting to interface objects ($i1$ to $i3$). Different interface objects provide different potentialities to utilize OS services.

Language-based means that we use our programming language INSEL (Section 4) to develop OS services as well as user-level applications. This has important consequences: 1) No additional heterogeneous concepts are introduced by OS services. 2) User-level objects and OS services are accessed in a uniform manner. 3) OS services are themselves structured according to our formal concepts. 4) Well-defined structural dependencies among all objects of the system are recorded and can be considered for global resource management decisions. This encloses dependencies between objects of a single application as well as inter-application dependencies and even dependencies between applications and OS level objects.

We state that this integrated view together with structuring concepts offers new opportunities for system-wide resource management. Without the knowledge of global dependencies, a distributed OS has to choose between two possible resource management strategies. First, it could try to optimize the realization of certain applications regardless of influences on other applications that are running in the system at the same time, which could violate fairness requirements. Second, it could concentrate on balancing workloads. This does not take any application-specific requirements into account, which might lead to considerably weak performance of specific applications. Since neither of these strategies is satisfactory, a combination of both is required that can be achieved by combining a top-down and a language-based approach.

4. INSEL Concepts

According to our formal concepts [11], we derivated language concepts for distributed programming. INSEL (Integration and Separation Language) is an imperative, object-based and type-safe high-level programming language with an Ada-like syntax [1]. We chose an object-based style of programming since objects support reuseability, structuring of complex systems and modeling of real world problems in a natural way. At the OS level, objects can serve as units of management and distribution and therefore facilitate global resource management.

We did not introduce a completely new programming paradigm or choose the functional style of programming as in SISAL [5], since we intend to keep distributed programming as simple as possible.

INSEL provides language concepts, to explictly determine sequential and parallel computations, on a high level of abstraction, without any references to the OS or the physical execution environment such as number of processors. This transparency enables adaptability of INSEL applications to varying hardware configurations.

INSEL distinguishes between *named* and *anonymous* objects. Named objects are known at compile-time. Anony-

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

mous objects are dynamically created in the path of a computation using a `NEW` operator. Pointers to anonymous objects can be duplicated or passed between objects in the system, which makes resource management difficult and less efficient. To reduce these awkward properties, INSEL does not support the creation of a reference to a named object as it is possible in C++ or Ada. INSEL does as well not support the explicit deletion of objects. An INSEL object is automatically deleted if its computation has ended and it is no longer accessible by any other object.

4.1. Generator Concept

All INSEL objects are created as instances of class describing components, called *generators*. The interface and implementation of an object is fully determined by its generator. INSEL does not yet differentiate between interface and implementation of objects as it is done for example in CORBA.

Our generators can be compared to *classes* in C++ [21] with the difference that generators are integrated in the system just like any other object. This is a contrast to other object-oriented languages, where *classes* are static components, that are organized separately from the flat hierarchy of the objects. Languages like C++ or Eiffel [15, 13] provide hierarchical structuring of classes, through inheritance. They often support a hierarchical organization of the class name space by providing nesting of class definitions. But instances of these classes are organized in a flat hierarchy, uncoupled from structural dependencies, implied by class hierarchies and naming schemes. This is different in INSEL, where the structuring of generators predetermines dependencies between object instances.

4.2. Object Model

INSEL is object-based in the sense that it supports encapsulation but no inheritance. INSEL objects can be either passive or active. Active INSEL objects are called *actors*. Objects of both kinds can be created dynamically at runtime. Each compound object has a *declaration* and a possibly empty *statement part*, both determined by its generator. The declaration part might contain declarations of local objects, methods, or nested generators. The statement part can be compared with a *constructor* in common object-oriented languages.

Actors serve for the explicit specification of parallelism on a high level of abstraction. The actor concept defines abstract properties of active objects. The programmer does not specify any properties referring to the physical realization of an actor such as a specific machine or a specific thread or tasking concept. By creating an actor, a new flow of control is established, that executes the statement part of the new actor in parallel to the flow of control of its creator. An actor terminates if it has reached the end of its statement part and all its dependent objects have terminated.

Semantics of passive INSEL-objects are similar to those of other OO-languages. By creating a passive object, the flow of control of the creator switches to the newly created passive object to execute its statement part. When the end of this computation is reached, the passive object terminates, the flow of control switches back to the creator, which can later on interact with the terminated object via its access methods.

Object interaction

Both active and passive objects encapsulate data and services to access the data. The interface of an object is determined by *exported* access methods. Two different possibilities for the interaction of INSEL objects exist. First, they can directly interact in a client-server style. Second they can cooperate indirectly by using shared objects. Hence, INSEL supports message passing as well as the shared memory paradigm, which is a contrast to platforms like Orca [3] or CORBA which only allow for message passing style of programming. We found that both paradigms are necessary to enable a natural style of distributed and parallel programming.

The execution semantics for all requests to an object is solely *at-most once* and all requests are served synchronously. We do not offer any asynchronous style of communication, because this would introduce the necessity to cope with difficult error situations. A service request to an actor is served in the same way as one to a passive object, with the exception, that when requesting a service from an actor, the caller and the callee synchronize using operation-oriented rendezvous semantics. The caller has to wait for the callee to accept the request. After accepting the request, the callee performs the calculation, returns the result and both the caller and the callee continue their computations in parallel. This is the only synchronization technique available in INSEL to coordinate parallel activities. We do not provide any other low-level and error prone mechanisms like semaphores.

Figure 2 illustrates some of the concepts, described in this section. (1) starts the definition of a generator for actors of class `system`. (2) declares the interface of passive objects of class `D_t`, which solely consists of the access method `get`. (3) defines the implementation of class `D_t` which comprises a declaration (4) and a statement part (5). (6) declares a generator for pointers to anonymous objects of class `D_t`. (7) defines a generator for actors of class `T_t` (interface and implementation), which offer the service `coop`. (8) defines the statement part of objects of class `system`.

```
TASK system IS                          -- (1)
  DEPOT SPEC D_t ...                    -- (2)
    FUNCTION get ...

  DEPOT D_t (x: IN INTEGER) IS          -- (3)
    v : ARRAY[1..x] OF INTEGER;         -- (4)
    FUNCTION get ...
      RETURN x;
    ...
  BEGIN ... END D_t;                    -- (5)

  TYPE DP_t IS ACCESS D_t;              -- (6)

  TASK T_t IS                           -- (7)
  c     : INTEGER;
  e     : D_t;
  dp    : DP_t;
    ENTRY coop IS
    BEGIN
      ...  count := count + 1; ...
    END coop;
  BEGIN
    dp := NEW D_t(42);
    ... ACCEPT coop; ...
  END

  d: D_t(8);
  t: T_t;
BEGIN                                   -- (8)
  OUTPUT(d.get);
  t.coop; ...
END;
```

Figure 2. Sample INSEL program

5. System Structures

The programmer implicitly determines structural dependencies within an INSEL program by nesting of generators, ordering of declarations, definition of pointer generators and so on. In section 6 we will elaborate on how this information can be exploited to improve the performance of distributed applications.

5.1. Definition Structure

Nesting of generators and objects establishes a hierarchical name space. For each INSEL object O, the set of visible and accessible objects and generators is determined by the *execution environment* – $U(O)$. The definition structure serves as a base to compute $U(O)$.

Definition 5.1 (Definition Structure)
An object O is definition dependent on object P — $\delta(O, P)$ $\Leftrightarrow$ the generator for O is contained in the declaration part of object P.

$U(O)$ is calculated by tracing levels of nesting, which is done by transitively following the definition structure and collecting informations about visible objects and generators on each level of nesting. Given a certain object O, we first investigate δ to locate the generator G of O, which is a local component of an object P. All components of P that are declared before G, according to the sequence of declarations at the same level of nesting, are visible to G and added to $U(O)$. We continue to compute $U(O)$ by recursively descending the δ-structure, e.g. next step would be, to analyze object H, which is given by $\delta(P, H)$. The computation ends when an object is reached, that is not definition dependent on any other object (the root object).

5.2. Execution Structure

The execution structure is composed of three relations, that describe dependencies among the parallel and sequential flow of controls within the system. This delivers important information to the load managing system and the scheduler.

Along with the creation of a new actor A, a new flow of control is established that executes the statement part of A in parallel to the computation of its creator. This relationship is recorded by the π-structure.

Definition 5.2 (π – structure)
Object A operates in parallel to object O — $\pi(A, O)$ $\Leftrightarrow$ A is an actor and was created by O.

The execution of the statement part of a newly created passive object is sequentially embedded in the flow of control of the creator. Requests to passive objects are as well executed by sequentially embedding the operations of the requested service into the flow of control of the caller. These sequential relationships are recorded by the σ-structure.

Definition 5.3 (σ – structure)
An object O is sequentially dependent on an object P — $\sigma(O, P)$ $\Leftrightarrow$ the flow of control has moved from object O to object P.

The third component of the execution structure is the κ-dependency. It describes communication dependencies between actors that synchronize to realize requests of services with rendezvous semantics.

Definition 5.4 (κ – structure)
Actor A communicates with another actor B — $\kappa(A, B)$ $\Leftrightarrow$ A requested a service from B, B has accepted the service and both, A and B are synchronized to perform the requested service.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

5.3. Locality Structure

Actors or passive objects, which are declared in the declaration part of an object O can be expected to be mostly used by O and its nested objects. Therefore, this kind of location dependency gives hints to the runtime and operating system, to co-locate objects on either the same node or at least close to each other.

Definition 5.5 (λ – structure)
An object O is local to an object P — $\lambda(O,P) \Leftrightarrow O$ is a named object and is declared in the declaration part of P.

5.4. Structure of Pointer Generators

Efficient memory management of objects that are dynamically created in the path of a computation using the `NEW`-Operator is difficult. This is due to the fact that pointers can be passed around and even duplicated, which disables an easy stack-like memory management. We try to facilitate the management of such *anonymous* objects by tracking the location of generators for pointers.

Definition 5.6 (γ – structure)
Object O is γ-dependent on object P — $\gamma(O,P) \Leftrightarrow O$ is an anonymous object and P is the location where the generator that is needed to create pointers to O is declared.

5.5. Termination Structure

Combining the λ and the γ structures described above, we defined a termination dependency for passive and active INSEL-objects, which simplifies memory management considerably [8] [22].

Definition 5.7 (ϵ – structure)
Object O is termination dependent on object P —

$$\epsilon(O,P) \Rightarrow \begin{cases} \lambda(O,P) & \wedge \quad O \text{ is a named object} \\ \gamma(O,P) & \wedge \quad O \text{ is an anonymous object} \end{cases}$$

We basically use the termination dependency to ensure that no object is deleted as long as it could potentially be accessed by another object. For example: $\epsilon(O,P)$ determines that object P must have terminated its computation, before object O can be deleted. It also determines that the prerequisite condition to delete object P is the termination and deletion of object O.

Figure 3 illustrates some of our structural dependencies. It shows a snapshot of a simple INSEL system at runtime, that evolved from executing the program listed in figure 2. By starting the execution of the program, a root actor `r` was created, which is of class `system`. `r` elaborated its declaration part and created object `d` and actor `t`. Both are location and termination dependent on `r`. In turn, `t` has created object `e` and an anonymous object `a'` of class `D_t`, which is termination dependent on `r` because of its γ-dependency on `r`. Currently `r` and `t` are synchronized, that means `r` requested a service from `t` and `t` has accepted to serve this request.

6. Resource Management

The INSEL resource management system aims to transform an abstract distributed system given as an INSEL program into a low-level representation that can efficiently be executed on a distributed hardware configuration.

In the following subsections we will first present our basic approach to develop a distributed resource management system that adapts to changing requirements and provides scalability to varying sizes of the distributed system and the hardware configuration. After that we will demonstrate some practical benefits of our concepts to structure distributed systems.

6.1. Management Architecture

Based on the termination dependency, we cluster objects into *actor contexts* (see figure 3), which are essential units of management. Each actor context consists of one actor and all of its termination dependent passive objects. By associating an abstract manager with each actor context, we construct a reflective manager architecture. The task of each manager is to enforce actor context-specific resource management. It is obvious that managers need application-level information to adapt to the requirements of actor contexts. This information is provided by analyzing our structural relationships at compile-time as well as at runtime. The interaction between application and management layer is accomplished completely transparent to the application.

It is important to notice that these managers are abstract in that they are not necessarily objects that are linked to actor contexts. Such a rigid implementation of managers would introduce an enormous management overhead, which would disable a flexible realization of fine-grained parallelism. Instead, a manager might just be given by a simple data structure or it might itself be a rather complex object, comprising own activities and objects. For example, if an actor A does not contain any local (analyzing the λ-structure) generators for pointers, then the associated manager does not have to be prepared for heap management. Another technique is to predetermine the κ-structure at compile-time based on analyzing the execution environment $U(A)$. If this shows, that A has no potentialities to communicate with other actors, then communication facilities are omitted, that in turn leads to a more light-weight manager implementation. A *minimal* manager is completely realized as inline code generated by the INSEL compiler and only supports stack handling for its

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

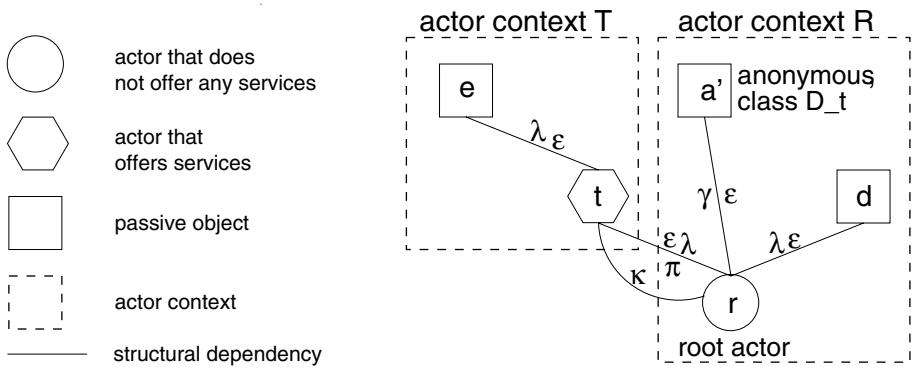

Figure 3. Snaphshot of an INSEL system

associated actor context. More advanced managers provide services to maintain consistency of replicated objects or to perform access controls or load balancing.

6.2. Global Memory Management

Our memory management system is realized in a decentralized manner by our manager architecture, using a 64 bit single address space. This flat address space is roughly divided into three main areas as shown in figure 4 (based on our sample program listed in figure 2): code, stacks and heaps. Each manager is responsible to fulfill the memory requirements of its associated actor context. In our example (see figure 3), the clustering of objects to actor contexts leads to two actor contexts, `R` and `T`. The manager of actor context `R` is responsible for realizing root object `r` and the anonymous object `a'`. To realize named objects, each manager maintains an own stack within the global stack area. Hence, object `d` is realized on the stack that is maintained by the manager of actor context `R`. A realization of anonymous objects on stack is usually not feasible, because their number and sizes are seldom determined at compile-time. Therefore, managers associated with actor contexts that contain generators for pointers (analyzing the γ-structure) also maintain an own heap within the global heap area, where termination dependent anonymous objects (as shown in our example with object `a'`) are placed. In contrast, the manager of actor context `T` does not maintain its own heap, since the actor context does not contain any generators for pointers. Conflicts arising from stacks growing to the beginning of another stack are solved by communication between managers, according to the π-portion of the execution dependency. For example, if the manager for `T` detects a shortage of memory, it communicates with the manager of `R` to claim additional memory for its stack.

This management scheme is decentralized and does not contain a potential bottleneck. The managers operate mostly independently and in parallel, except for claiming or freeing portions of memory. It also adapts to the requirements of the INSEL system, because the number and the potentialities of managers depend on the number and the requirements of actor contexts. This flexible scheme is possible, through utilizing the λ- and γ-dependencies to calculate the ϵ-dependency and exploiting this information for the clustering of termination-dependent objects to actor contexts.

6.3. Transparent utilization of different DSM strategies

INSEL supports cooperation via shared objects. Due to the physical distribution of memory resources, some means to enforce some kind of *Distributed Shared Memory* [14] have to be established. As investigated in [9], knowledge of access characteristics such as producer-consumer relationships helps to chose between different kinds of consistency protocols, like release or strong consistency. This significantly improves the performance of DSM systems. In [22] it has been shown how the required information can be gained by using our structural dependencies.

7. Considerations regarding interoperability

It should be evident that the philosophy behind the MoDiS project differs from those, underlying the work performed in the context of distributed objects platforms as investigated in section 2. Our approach seems to result in a kind of closed INSEL-world, being incompatible to existing applications using conventional programming languages (e.g. C, C++) or being unable to interoperate with existing distributed objects platforms. We do admit that, until now, issues like portability, compatibility and interoperability have been of minor importance for us. Our work focuses on the development of concepts and tools to simplify

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

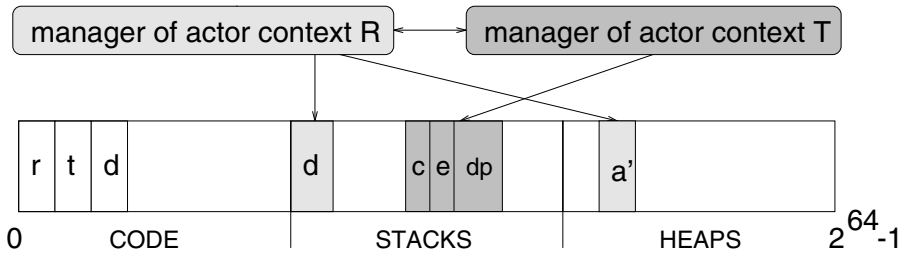

Figure 4. Partitioning of the global address space

the complex task of programming distributed applications as well as on the development of a system architecture, which provides efficient resource management. Nevertheless, we claim that our closed INSEL-world could be opened to cope with the additional issues of interoperability as well. In the following, we will just sketch an idea of how an INSEL-system could be integrated into an open system environment.

To interoperate with other systems, a special INSEL object has to be designed and implemented. This INSEL interface object or *bridge* has to provide services comparable with an object request broker known from CORBA. It serves as a link between the INSEL world and the outside world. It is important to note, that such an INSEL bridge will introduce interoperability on a inter-system basis. It will not allow any intra-system interoperability, that is the usage of other programming languages than INSEL to implement objects that are to be executed within the INSEL system. The introduction of such a feature would require more general resource management services, which would be an antagonism that contradicts our objective targets. To export INSEL services to the outside world, the INSEL bridge has to provide a service to register an INSEL object as being usable from the outside world. Registering means, that an IDL specification of the exported object has to be derived automatically from the generator definition of the INSEL object and an interoperable object reference has to be created which can be used by outside clients. A client uses the interface specification provided by the bridge object to compile a client-stub which enables to interact with INSEL server objects as it would interact with local objects, that is by invoking their operations. All server requests are mediated through the INSEL bridge which has to provide a binding between the request and the INSEL implementation of the requested service. As the request is executed within the INSEL world, all features explained in this paper can be exploited to improve the performance of service executions. Hence, the INSEL bridge can be viewed as a solution to the performance problems mentioned by Vinoski in [20], coming along with common CORBA implementations.

To summarize, by tightly integrating facilities such as ORBs into a holistic system architecture, spanning languages, tools and OS services, considerable improvements in performance are enabled. Knowledge about language specific structural dependencies would help distributed objects platforms to realize interoperability efficiently, while providing transparency to the programmer.

8. Current Status and Future Work

Currently three prototype implementations of our programming language INSEL exist. One is an interpreter integrated in an analyzation and visualisation tool, called *DAViT*. It is capable of visualizing all of our structures of a distributed INSEL application at runtime. It serves as a learning tool for collecting practical experiences with our structuring and programming concepts.

DAViT runs on top of a HP-UX workstation cluster, interconnected trough a 10 MBit ethernet network. For the same platform we realized another implementation, called *EVA* [19], which concentrates on load distribution. *EVA* translates INSEL programs into semantic equivalent C++ code, which in turn is being compiled and linked with additional C++ libraries, for runtime support. Our third implementation, *AdaM* [22] performs a translation of INSEL into C. It focuses on distributed memory management techniques and strategies, such as migration and replication of passive objects. It was also our first attempt to exploit micro-kernel technology for our approach. *AdaM* is based on Mach 3.0 running on an ethernet cluster of i486 computers.

AdaM and *EVA* themselves are written in C and C++. They are experimental implementations, in that they translate complete INSEL applications, link them with runtime support libraries and execute them. Those prototype implementations enabled us to collect first experiences with our concepts and their implementation. Naturally they are missing important OS features, as management of users, I/O-system, accounting and so on. We now aim to integrate our experiences with load distribution, memory management and micro-kernel technology into a complete INSEL OS architecture.

Proceedings of The Thirtieth Annual Hawwaii International Conference
on System Sciences ISBN 0-8186-7862-3/97 $17.00 © 1997 IEEE

1060-3425/97 $10.00 (c) 1997 IEEE

From our experimental implementations we also learned that existing tools to construct software systems do not match our requirements, since they are mostly tailored for UNIX environments. We are currently implementing new tools as for instance a native INSEL to machine-code compiler and a new dynamic linker. Using these tools, we will first implement some basic services of a distributed OS in INSEL (distributed scheduling, dynamic loader, etc.). The main task of this base system will be to support the dynamical extension of the running system at runtime.

We are going to implement our system on a cluster of 14 Sun Ultrasparc workstations, interconnected by a 100 MBit/s FastEthernet. In a first step, we will still base parts of our services on Sun Solaris facilitites. In a second step, we will replace the remaining Solaris services with our own micro-kernel *DYCOS*.

9. Conclusion

In this paper we presented an approach to overcome two considerable difficulties of distributed object-based systems. First, our approach provides a homogeneous and fully transparent programming interface to the programmer. It allows the programmer to solely concentrate on specifying algorithms without being bothered with resource management tasks, as for instance implementing communication facilities or requesting addresses of objects. Second, we demonstrated that distributed applications, specified on such a high level of abstraction can efficiently be executed. This task is accomplished by consequently exploiting structural dependencies for resource management.

The key to these features is the unique combination of top-down orientation with a language-based approach. Starting with abstract concepts to construct structured distributed object-based systems, we develop language concepts and well-adapted resource management tools. This means that all of our concepts and strategies enclosing for instance compiler technology down to low-level services offered by our micro-kernel *DYCOS*, are perfectly adapted to the requirements of our abstract concepts.

Interoperability with other distributed OSs is feasible, by incorporating interface components, comparable to ORBs. Such components should be enhanced with facilities to exploit application-specific structural dependencies for efficient realization of objects and requests.

References

[1] Ada. *The Programming Language Ada Reference Manual*, volume 155 of *LNCS*. Springer–Verlag, Berlin, 1983.

[2] ANSA. An engineers introduction to the architecture. Technical Report TR-03-02, APM Ltd., Cambridge, England, November 1989.

[3] H. E. Bal. Report on the programming language Orca. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1994.

[4] R. H. Campbell and N. Islam. Choices: A Parallel Object-Oriented Operating System. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programmi ng*. MIT Press, 1993.

[5] D. C. Cann. Sisal 1.2: A brief introduction and tutorial. Technical report, Lawrence Livermore National Laboratory, 1992.

[6] C. Consortium. *The Comandos Distributed Application Platform*. 1992.

[7] C. B. Czech. Designing reconfigurable microkernels for distributed operating environments. submitted to SPDP '96.

[8] C. Eckert and H.-M. Windisch. A new approach to match operating systems to application needs. In *Proceedings of 7th IASTED - ISAMM*, pages 499–503, October 1995.

[9] J. B. C. et al. Implementation and performance of munin. Technical report, Computer System Laboratory, Rice University, Houston, Texas, 1991.

[10] M. A. et al. Mach: A New Kernel Foundation For UNIX Development. Technical report, CS Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[11] P. S. et al. Concepts for the construction of distributed systems. SFB-Bericht 342/09/96 A TUM-19618, Munich Institute of Technology, March 1996. german.

[12] S. Groh. Designing an efficient resource management for parallel distributed systems by the use of graph replacement system. In *Proceedings of PDPTA 96*, August 1996.

[13] L. Gunaseelan and R. L. Jr. Distributed Eiffel: A Language for Programming Multi–Granular Distributed Objects on the Clouds Operating System. In *Intern. Conf. on Comp. Lang.*, San Francisco, April 1992.

[14] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[15] B. Meyer. *Object-oriented Software Construction*. Prentice–Hall International (UK) Ltd., 1988.

[16] OMG. The common object request broker: Architecture and specification. Technical report, Object Management Group, July 1995.

[17] OSF. *Introduction to OSF DCE*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[18] S. K. R. Balter, J-P. Banâtre. *Construction des systèmes d'exploitation réparties*, volume Collection didactique. INRIA, July 1991.

[19] R. Radermacher. *An Execution Environment with Integrated Load Balancing for Distributed and Parallel Systems*. PhD thesis, Munich Institute of Technology, 1996. german.

[20] D. C. Schmidt and S. Vinoski. Object interconnections. *SIGS C++ Report magazine*, May 1995.

[21] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, MA, 2nd edition, 1991.

[22] H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In *Proceedings of PDPTA 95*, pages 680–688, November 1995.

[23] Y. Yokote. Kernel Structuring for Object–Oriented Operating Systems: The Apertos Approach. Technical report, Sony Computer Science Laboratory Inc., 3-14-13 Higashigotanda, Shinagawa-ku, Tokyo, 141 JAPAN, July 1993.