Flexible, Distributed and Adaptive Resource Management in MoDiS *

P.P. Spies, C. Eckert, M. Pizka, C. Czech, J. Geiger, C. Rehn

Abstract

Currently, a shift of paradigm from sequential to distributed computing can be observed. Great efforts are needed to cope with the challenging demands that are inherent to this transition. The most important issues and requirements concern two main areas. First, programming of distributed applications should be *significantly simplified* by hiding as many details of the underlying physical distribution of the hardware configurations as possible. Second, the *performance* of distributed systems has to be enhanced by providing an efficient resource management for distributed systems that is completely transparent to the application level. The research project SFB #324/A8 dealt with a novel language- and object-based approach called MoDiS to cope with these demands. Distributed systems are developed and transformed into executables following a systematic, top-down driven method. Language-based is intended to mean that a high-level programming language is used to develop operating system services as well as user-level applications. The language-level concepts employed are based on objects and support advanced structuring features and incremental system construction in a controlled manner. Structural dependencies between objects are implicitly determined at the application level and exploited by the distributed resource management system to transform applications into efficient executables.

Keywords: Distributed System, Resource Management, Programming Language, Incremental System Construction, Reflective Management

*We would like to thank the German Research Council DFG for funding this work as part of the research project SFB #342/A8.

1 Introduction

Currently, a shift of paradigm from sequential to distributed computing can be observed. Great efforts are needed to cope with the challenging demands that are inherent to this transition. The most important issues and requirements concern two main areas. First, programming of distributed applications should be significantly simplified by hiding as many details of the underlying physical distribution of the hardware configurations as possible. Second, the performance of distributed systems has to be enhanced by providing an efficient resource management for distributed systems that is completely transparent to the application level.

From the developers point of view, a programming environment that offers concepts, features and tools to ease the development of large-scale distributed applications is a requirement. The developer should not be explicitly concerned with the realization of details, such as using socket addresses or port numbers of services to implement distributed object communication, binding object names or keeping track of the current load situation. Details of the underlying hardware configuration and the operating system being used should be hidden from programmers. Moreover, application developers should not be forced to handle new concepts (e.g. different RPC semantics) and tools (e.g. interface definition languages). Hence, homogeneity of concepts offered by the programming environment is required.

Object technology seems to provide promising features to cope with these demands. Among other advantages, object orientation supports component reuse, object interaction via well-defined interfaces, inheritance, and encapsulation. The object paradigm supports the modeling of real world problems in a natural way. Moreover, objects define appropriate units to be managed by the underlying distributed resource management system (e.g. distribution of single objects or replication of objects). Hence, object technology is and will be widely used to develop large-scale applications as well as operating systems.

Complex application systems like multi-media applications or parallel numeric algorithms occupy large amounts of system resources and claim individual operating system support like security, fast context switching or real-time processing. Therefore, the underlying distributed operating system should offer a distributed object management that can be customized to application-specific requirements in a transparent way. Moreover, to cope with varying requirements of applications the resource management system must be able to dynamically adapt its strategies (for instance scheduling policy), depending on changes in overall resource utilization and application-specific needs.

To fulfill these requirements the resource management system needs appropriate information about application-level objects. We claim that the majority of this information can be extracted from structural dependencies between objects provided that such relationships can be expressed by the programmer. Structural dependencies like object nesting or communication patterns can be exploited, for instance, to cluster objects into units which can be placed on the same computing node, or to migrate single objects if the number of remote accesses to the object exceeds a given limit.

Obviously, we require that the distributed management provides for scalability and introduces only low overall management overhead. The performance has to be reasonable compared to sequential and centralized software solutions, as well as it should provide relevant speed-ups if additional computing resources are available.

Within the project A8 as part of the SFB 342 we have developed a new *top-down* driven and *language based approach*, called MoDiS [EP99], to meet the requirements stated above. MoDiS provides abstract concepts to construct an object-based, distributed operating system for a cluster of loosely-coupled workstations. The resulting system (MoDiS-OS) integrates applications as well as tools and classical OS features.

The approach is centered around our object-based, high-level programming language called INSEL which offers advanced structuring features and language concepts allowing the programmer to specify parallel and cooperative applications on a high level of abstraction. Structural dependencies between objects are determined at the application-level. Hence, the top-down approach brings ease to the task of programming distributed systems. The programmer is able to specify the distributed system with a homogeneous repertoire of language concepts without having to cope with the details of the physical realization such as making decisions about the placement of a specific object.

The MoDiS approach is characterized by a tight coupling of all transformational steps (e.g. compiler, linker, resource management) involved. This allows the preservation of application-specific structuring information throughout all of these steps. That is, information gathered by the compiler, for instance, is passed to the resource management system and is enhanced with information that is gathered dynamically at runtime. This information flow provides a basis for improving resource management for distributed systems.

The top-down orientation combined with a language-based approach leads to a single system spanning operating system functionalities and user-level applications. This integrated view together with structuring concepts offers new opportunities for system-wide resource management. To start new applications and to enhance the system with new functionalities MoDiS provides the ability to dynamically modify the running system. Within MoDiS we have elaborated concepts to construct the system in an incremental, but nevertheless controlled, manner. We are able to dynamically extend the system functionalities as well as dynamically remove services that are no longer required.

Both, the dynamic modification of the running system and the selection between alternative realizations are supported by the flexible and incremental linker and loader *FLink* which is an essential part of the overall management in MoDiS. In addition to classical linker and loader tasks , FLink provides the ability for incremental and dynamic system construction and offers extended flexibility using different policies for symbol resolution. Using FLink enables the management to choose from different techniques to resolve symbol references, to change symbol references and even to reverse the decisions for a program in execution without stopping and restarting it.

The remainder of the paper is organized as follows. Section 2 investigates related work in the area of distributed object-based systems. Section 3 introduces the top-down driven approach that aims to overcome some of the deficiencies revealed. Sections 4 and 5 elaborate on the main language concepts and structuring facilities of the approach. An overview on the new adaptive and scalable distributed resource management system is presented in section 6. Section 7 discusses incremental system construction which is an important aspect in MoDiS due to the single system approach. Closely related to incremental system construction is the extended management flexibility presented in section 8 as both are realized in MoDiS using the linking concepts and techniques of section 9. Section 10 reports on implementations of the MoDiS concepts before the main results of the paper are summarized.

2 Related Work

This section first investigates existing distributed environments to elaborate their benefits and deficiencies with respect to the requirements stated above. Since adaptive and transparent resource management is the intrinsic task of a distributed operating system, current research activities in this area are discussed afterwards.

2.1 Distributed Programming Environments

Computing environments for distributed applications like OSF's Distributed Computing Environment (DCE) [OSF92] or ANSA [ANS89] offer tools and services to support a procedural programming paradigm providing an RPC mechanism. Products based on the Common Object Request Broker Architecture (CORBA) [OMG95] 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.

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

But Programming within these software environments is not as simple 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 and 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 his/her own services. The programmer has to cope with interface definition languages (e.g. DCE-IDL, OMG-IDL) to specify interfaces. Access to remote objects must be handled in a different way than local objects, 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, the programmer 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 bugs that could be avoided by supporting a conceptual homogeneous environment instead. 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 Language-based Environments

Several research projects, such as COMANDOS [Con92], GUIDE [BLR94], or Emerald [SEJ95] — to name only few — tried to provide homogeneous distributed programming environments by following a language-based approach. The employed languages offer high-level concepts to develop distributed object-oriented programs. Unfortunately, all of them lack appropriate structuring features. Though specific runtime systems are implemented on top of existing operating systems, these runtime systems are not able to gain information about structural dependencies between application objects in order to enhance resource management. Hence, neither the development of structured large-scale applications, nor their application-adapted efficient execution is supported with these approaches.

With ORCA [Bal94] major steps towards application-specific resource management were taken. Tools, such as the compiler are tailored to ORCA and enable static and dynamic analysis. The results of these analysis 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.

2.3 Distributed Operating Systems

Performance issues with respect to efficient resource management have not been a major issue in the design and implementation of existing distributed programming environments. Consider the CORBA environment. Transparent resource management is the task of the Object Request Brokers (ORBs), but actually the existing ORBs limit their services to locating objects and performing parameter marshaling. If, for instance, an ORB is able to exploit application-specific information, like access patterns, the realization of these accesses may be optimized by using the available resources more efficiently. Because efficient resource management is the task of operating systems research activities in this field are investigated in the following paragraphs.

Object-oriented technology provides appropriate means to configure operating systems at a pre-execution time. That is, operating systems can be statically customized to application needs (e.g. Choices [CIRM93]) with low overhead. However, an application-specific resource management capable of dynamically adapting its management decisions to the changing requirements of various applications demands for more flexibility.

Current research performed within the area of reflective architectures [Yok92, CHJ$^+$94, LYI95, CAK$^+$96] pursues promising ideas to solve the problem of adaption. Reflective operating systems offer different policies to perform resource management, for instance, different scheduling algorithms. Each application is enabled to select a policy which matches its specific needs. This selection is performed dynamically at runtime using "Meta-level" protocols [KdRB91, WS95]. Appropriate information about the application itself concerning, for instance, its access behavior, is needed to optimize these decisions. Such information can be gained by analyzing structural dependencies between application-level objects. Unfortunately, due to the lack of structuring facilities offered by the underlying object-oriented programming languages only very restricted analysis can be performed. Hence, only a small bit of information can be extracted from the application-level to support an efficient application-specific resource management. Moreover, selections of proper strategies has to be done by the application programmer explicitly, which is contrary to the required transparency.

Kernel-based systems currently devote a lot of attention to satisfy the requirement of adaptability and extensibility of operating system services. Approaches like SPIN [BSP+95], Paramecium [vDHT95] and VINO [SESS94] are characterized by specific features that allow dynamic integration of application-level code into the protection domain of the kernel. To exploit adaptability features programmers are burdened with performing resource management tasks explicitly. This aggravates the programming of distributed application considerably.

These observations of related research activities can be summarized in following results. A programming environment that offers appropriate concepts to develop well-structured distributed systems on a high level of abstraction is a requirement. In addition, these distributed systems should be executed on top of interconnected workstation clusters where all required resource management actions (e.g. load distribution, migration, distributed memory management) are automatically performed by the underlying operating system. The management must be adaptable to dynamically changing requirements of distributed applications and fully be transparent to the application programmer. To meet these requirements a new top-down oriented, language-based and integrated approach – the MoDiS approach – is proposed.

3 The MoDiS Approach

The acronym MoDiS stands for Model oriented Distributed Systems and emphasizes that abstract concepts and models [Spi96] are the foundation of this project.

3.1 Top-Down driven

Existing operating systems are characterized by the bottom-up construction of their services and programming interfaces. They aim at enhancing the simple functionality provided by the hardware to more powerful services offered to the application-level. Due to the bottom-up oriented construction, the properties of the application-level and the language-level are not taken into account. For instance, language-level concepts may require efficient support for fine-grained persistent data objects instead of coarse-grained files. Bottom-up orientation usually leads to general purpose resource management features and some basic services that do not match the needs of applications. For example, UNIX systems offer heavy-weight processes to the application level, which are inadequate to realize fine-grain parallelism. To overcome this lack of flexibility, systems like Mach [Acc86] introduced threads. But again, this is another fixed and general purpose abstraction. 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 these deficiencies we follow a top-down oriented approach, deriving low-level facilities from requirements of the application level. Within the MoDiS project, abstract concepts to construct structured distributed systems are developed. Top-down orientation in this sense means that the construction of a distributed system starts with the specification of the system at a high level of abstraction. The programming language used, INSEL [Win96a], provides language concepts that are well adapted to the 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. The top-down approach brings ease to the task of programming distributed systems. The programmer is able to specify the distributed system at a high level of abstraction with a homogeneous repertoire of language concepts. He does not have to cope with the details of the physical realization such as making decisions about the placement of a specific object or explicitly using operating system services like threads and semaphores.

Realization of a specified system on a given hardware platform is done by stepwise refinement of the abstract properties towards more concrete ones. The more concrete the properties are the more they are influenced by existing physical resources like storage capacities and available processors. With each transition from one level of abstraction n to a lower level of abstraction

$n-1$, mappings from more abstract properties to more concrete ones have to be found [Gro96]. The facilities 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 information locally or remotely, in main or secondary memory, or 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 make appropriate decisions.

Some of these transformation steps are performed statically at compile-time, whereas others have to be done dynamically at runtime. Hence, the process of refinement encloses the complete life-cycle of a distributed system. The resource management in MoDiS provides the opportunity to adapt refinement decisions according to the dynamically changing demands of the software system. This is a significant difference to common systems which for instance, often separate the 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 information for all levels of abstraction to enable efficient resource management and macerate the usually hard separation of statics and dynamics. The top (high level of abstraction) to down (low level of abstraction) approach allows efficient performance of application specific resource management because important high-level structuring information such as access behavior or life-time dependencies gathered statically or dynamically is available.

3.2 Single system Approach

The top-down orientation is combined with a language-based approach [EW95a] that leads to a single system spanning the operating system functionalities and user-level applications. With the notion of *'system'* we refer to a structured set of objects that realizes the operating system 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 capabilities to utilize operating system services.

The programming language INSEL (see Section 4) is used to develop operating system (OS) services as well as user-level applications. This has important consequences:

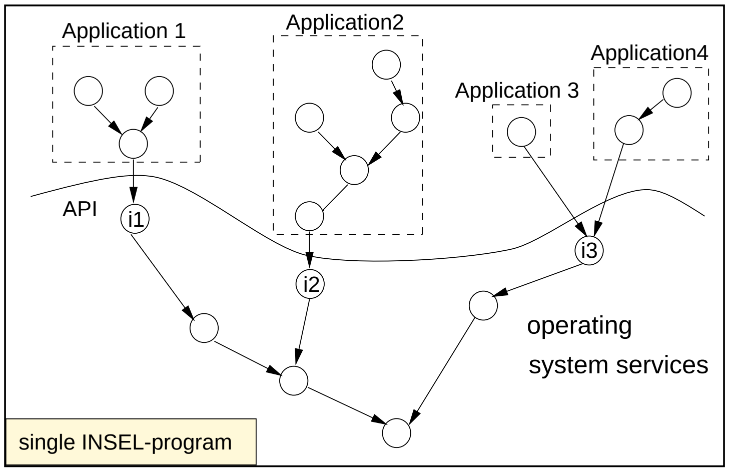


Figure 1: Single system view of a distributed system

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 for themselves are structured according to the 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 objects at the OS level.

This integrated view together with structuring concepts offers new opportunities for system-wide resource management.

4 Programming language INSEL

According to the formal concepts [Spi96], language concepts for distributed programming have been derived. INSEL (Integration and Separation Language) is an imperative, object-based and type-safe high-level programming language with an Ada-like syntax [Ada83]. An object-based style of programming was chosen since objects support reusability, structuring of complex systems and modeling real world problems in a natural way. At the operating system level, objects can serve as units of management and distribution and therefore facilitate global resource management.

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

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.

INSEL generators can be compared to *classes* in C++ [Str91]. The difference is that generators are integrated in the system just like any other object. The structuring of generators in INSEL predetermines dependencies between object instances. 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. INSEL is object-based in the sense that it supports encapsulation but no inheritance.

INSEL objects can either be 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 may contain declarations of local objects, methods, or nested generators. The statement part can be compared with a *constructor* in common object-oriented languages. Both active and passive objects encapsulate data and services to access the data. The interface of an object is determined by *exported* access methods.

Actors serve as 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, 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 object-oriented languages. By creating a passive object, the flow of control of the creator switches to the newly created passive object in order 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 interact later with the terminated object via its access methods.

INSEL distinguishes between *named* and *anonymous* objects. Named objects are known at compile-time. Anonymous 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. Furthermore, explicit deletion of objects is not supported in INSEL. An INSEL object is automatically deleted if its computation has ended and it is no longer accessible by any other object.

Active objects may cooperate directly in a client-server style by synchronous rendezvous or indirectly by using shared passive objects. Hence, INSEL supports message passing as well as the shared memory paradigm, which is a contrast to platforms like ORCA [Bal94] 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.

Figure 2 illustrates some of the concepts, described in this section. Line 1 starts the definition of a generator for actors of class `system`. The generator for passive objects of class `D_t` which comprises a declaration of the variable `v` (3), a declaration of the access method generator `get` (5) and a statement part (8) starts on line 2. Line 10 declares a generator for pointers to anonymous objects of class `D_t`. Line 12 starts the definition of the generator for actors of class `T_t`, which offer the service `coop`. The statement part of objects of class `system` starts on line 33.

5 Language-Level Structuring Concepts

The following subsections present the main concepts to structure distributed object-based systems. These structures characterize the dependencies among the objects of a system and serve as the knowledge basis for resource management decisions.

Structural dependencies within an INSEL program are implicitly determined by the programmer himself. For instance, nesting of generators, ordering of declarations or definitions of pointer generators express specific dependencies which can be analyzed and exploited by the underlying resource management.

```
MACTOR system IS                              -- actor generator
  DEPOT D_t(x : IN INTEGER) IS -- passive object generator
    v : array [1..x] OF INTEGER;            -- object declaration

    FUNCTION get ...                  -- access method generator
      RETURN x;

  BEGIN ... END D_t;                           -- statement part

  TYPE D_t_ptr IS ACCESS D_t; -- pointer generator declaration

  CACTOR T_t IS                               -- actor generator
    c : INTEGER;
    e : D_t;
    dp : D_t_ptr;

    PROCEDURE coop IS     -- rendezvous operation generator
    BEGIN
      ... count := count + 1; ...
    END coop;

  BEGIN
    dp := NEW D_t(42);
    SELECT
      ACCEPT coop;
    END SELECT;
    ...
  END T_t;

  d : D_t(8);
  t : T_t;

BEGIN                                 -- statement part of system
  WriteLn(d.get);
  t.coop; ...
END system;
```

Figure 2: Sample INSEL program

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 1 (δ – 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 information about visible objects and generators on each level of nesting. Given a certain object O, the δ structure is first investigated 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 $U(O)$. The computation of $U(O)$ is continued 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 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 also executed by sequentially embedded operations of the requested service into the flow of control of the caller. These sequential relationships are recorded by the σ-structure.

Definition 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 P to object O.

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 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.

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 the operating system to co-locate objects on either the same node or at least close to each other.

Definition 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 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.

After the creation of an anonymous object it can be referenced and identified by a pointer value. This is recorded by the *zeta*-structure.

Definition 7 **(ζ – structure)**
An anonymous object O is ζ-dependent on object P — $\gamma(O, P) \Leftrightarrow$ a pointer V exist which references O and V is declared in the declaration part of P.

5.5 Termination Structure

Combining the λ and the γ structures described above, a termination dependency for passive and active INSEL-objects is defined, which simplifies memory management considerably [EW95b, Win95].

Definition 8 (ϵ – structure)
Object O is termination dependent on object P —

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

The termination dependency is basically used 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.

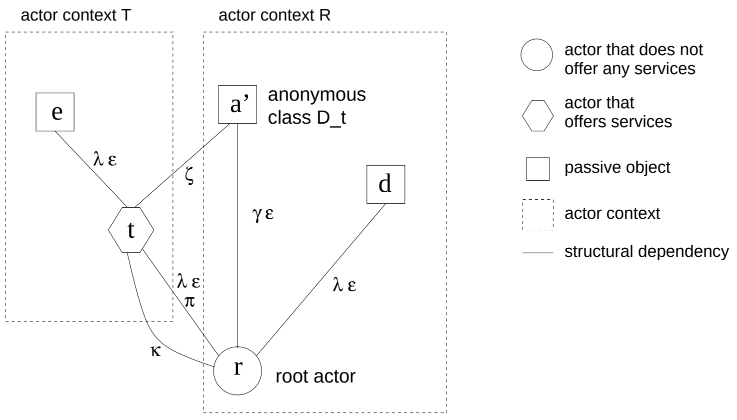


Figure 3: Snapshot of an INSEL system

Figure 3 illustrates some of the structural dependencies. It shows a snapshot of a simple INSEL system at runtime that evolved from the execution of 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.

The structural dependencies introduced reflect application-level properties. These dependencies are implicitly determined by the programmer employing the INSEL language concepts. The structural information can be exploited to enforce automated application-adapted resource management. Hence, the programmer is not burdened with having to specify hints to the resource management system in addition to writing his application program. Furthermore, as most of the explained structural dependencies are based on properties of INSEL generators, they are easy to analyze by software tools such as a compiler.

6 Resource Management Model

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

The following subsections will present the basic approach and implementation issues to develop a distributed resource management system that adapts to changing requirements and provides scalability to varying sizes of the distributed system and of the underlying hardware configuration.

6.1 Management Architecture

To enforce transparent, scalable, and adaptable distributed resource management, a model of a reflective resource management architecture [EW95b] was developed. Based on the termination dependency (ϵ-structure), objects are clustered into *actor contexts* (see figure 3), which are essential units of management. An actor context comprises exactly one actor and all its termination dependent passive objects. By associating an abstract manager with each actor context, a reflective distributed manager architecture is constructed. Just as objects in INSEL can be created and deleted dynamically, managers are created and deleted dynamically as well. The task of each manager is to enforce actor context-specific resource management. Besides fundamental tasks like allocating appropriate memory for all objects within its associated actor context the manager might also enforce context-specific access restrictions, maintain the consistency of replicated objects, or perform load-balancing.

Managers cooperate to enforce global resource management or to solve conflicting situations. For instance, stack overflows resulting from parallel allocations by different managers are handled by communication and cooperation between the managers. The set of managers is structured as well, because the application-level structural dependencies between the actor contexts (the *pi*-structure) is transfered to the manager. This tree-structure determines cooperation and communication links between managers. That is, to handle conflicting situations, a manager will contact its parent manager or its children, according to the tree-structure induced by the underlying *pi*-structure between actor contexts.

It is obvious that managers need application-level information to adapt to the requirements of actor contexts. This information is provided by analyzing the structural relationships at compile-time as well as at runtime. The interaction between the 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 objects that are linked to actor contexts. Such a rigid implementation of managers would introduce an enormous management overhead that 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 its 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 an analysis of the execution environment $U(A)$. If the analysis shows that A will not cooperate with other actors, then communication facilities are omitted which 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 associated actor context. More advanced managers provide services to maintain consistency of replicated objects or to perform specific access controls.

Hence, the associated abstract managers have to be implemented using alternatives adapted to the specific requirements of the managers.

6.2 Implementation Issues

The key idea of the MoDiS approach is to systematically incorporate management functionalities into the software tools involved in the implementation like compiler, linker, loader and the operating system kernel. Figure 4 depicts basic alternatives to implement management facilities. To improve the execution speed and to reduce the size of the target representation, we incorporate management functionalities into the compiler or the operating system kernel instead of employing layering techniques or runtime libraries.

The implementation alternatives can roughly be classified as static or

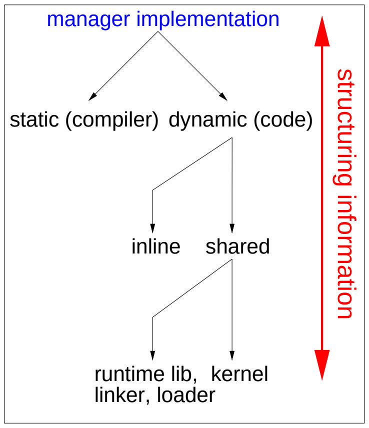


Figure 4: Implementation of the distributed manager architecture

dynamic. The static part is mainly concerned with the generation of manageable entities and comprises compiler as well as linker functionalities. Advanced linking techniques (see Section 9) are used in the linker to support adaptability and extensibility. This technique requires appropriate compiler support. For the INSEL front-end [Piz97] the GNU compiler system *gcc* was also modified to omit the generation of so called trampoline code [PEG97] for references to nested functions. At the same time, the code generator was extended to arrange stack frame specific linkage tables. The linker may independently decide either to bind open references with immediate addresses of code modules or to produce intermediate trampoline code and bind its address instead for enhanced flexibility. Execution of the latter results in a jump to the actual representation of the linked object. Hence, adaption and exchange of object representations can easily be handled using this linking technique.

The dynamic part is concerned with the execution and management of distributed applications at runtime. It splits up into an inline part comprising management functionality especially generated by the compiler for a specific set of objects. The shared part comprises usual operating system and runtime library services. For example the management of the *virtual single address space* [Reh00] has mainly been implemented as part of the runtime system. Basic facilities of hardware nodes are provided by the kernel part which manages resources associated with the local node. Moreover, the kernel enables communication with other nodes.

Figure 4 illustrates the tight integration of all management facilities involved in resource management. Based on this integration all management facilities cooperate and exchange information to achieve a distributed and efficient global management. Here the benefits of the top-down approach together with the structuring features of the programming language INSEL become evident. Structural information about overall system behavior as well as application-specific information is gained from static and dynamic analysis. In contrast to usual approaches, attributes of compile-time analysis are preserved and made usable for the dynamic resource management part. We use an extended attributed syntax tree in the compiler where attributes representing runtime properties of applications are stored and analyzed as well.

7 Incremental System Construction

Due to the single system approach (see Section 3.2) a MoDiS system is described by a single program specifying INSEL components which depend on each other according to the language-level structures introduced in section 5. In order to create a long-lasting system the user must be able to hook applications into the running system and also remove them later if necessary. Traditional systems facilitate the starting of applications at the system-call interface. A running program is not fully integrated into the operating system but has very limited relations to other applications in execution as structural dependencies are restricted in non language-based systems. Relationship between applications is neither specified nor known to the resource management and therefore can not be exploited for decision-making. In addition, the extension of a running system is limited to coarse-grained units. Fine-grained adaption of applications already in execution is not possible. Without the knowledge of global dependencies, a distributed operating system 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 simultaneously 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. This can be achieved by combining a top-down and a language-based approach with the concepts elaborated in this section, allowing a fine-grained and dynamic adaption and evolution of a MoDiS system.

On the one hand the single-program approach provides flexibility and structuring information describing dependencies between all components of the distributed system. On the other hand it necessitates the incremental extensibility of the system. Therefore concepts have been elaborated in the MoDiS project which make the dynamic system modification easy to handle for the user and at the same time controllable for the integrated management.

To demonstrate the abilities of incremental system construction in MoDiS figure 5 shows a sample INSEL program. The source code of an INSEL component, for example of procedure b, determines the properties of all b-generators and b-incarnations. We denote the set of generators (incarnations) specified by the source code of a component b by G_b (I_b).

```
 1 MACTOR system IS                                  -- actor generator
 2   PROCEDURE a(I : IN INTEGER) IS
 3     PROCEDURE b(J : IN INTEGER) IS                -- nested in a
 4     BEGIN
 5       ...
 6     END b;
 7   BEGIN
 8     b(I);                                         -- create b incarnation
 9     IF I > 0 THEN
10                     a(I - 1);                     -- recursive call
11     END IF;
12   END a;
13 BEGIN                                             -- statement part of root actor
14   a(42);
15 END system;
```

Figure 5: Dynamics of generator recursion

Definition 9 (Abstract properties of a component)
The abstract properties $P(b)$ of a component b of a MoDiS system are substantially determined by its signature, declaration and statement part which are described by the INSEL source code of b.

When executing this example program a generator $A \in G_a$ is created in the declaration part of actor *system*. In the statement part of actor *system* an incarnation $a_1 \in I_a$ is generated from this generator A. The generator $B_1 \in G_b$ is created in the declaration part of incarnation a_1 and an incarnation $b_{1_1} \in I_{B_1}$ of B_1 is generated in the statement part of a_1. In line 10 there is a recursive call to procedure a, so an incarnation $a_2 \in I_a$ is generated from generator A. This results in the creation of a new generator $B_2 \in G_b$, $B_2 \neq B_1$ in the declaration part of a_2.

Definition 10 (Extension/reduction of properties)
Let b be a component of a MoDiS system with properties $P(b)$. We call the modification of b to b' an extension *(*reduction*) of the properties of b if $P(b) \subset P(b')$ $(P(b) \supset P(b'))$.*

When modifying the program of figure 5 dynamically, the extension/reduction of a b-incarnation (e.g. b_{1_1}), a b-generator (e.g. B_2) or all b-generators B_* which are created in recursive calls of function a would be reasonable. Therefore generators and incarnations are not sufficient to describe all candidates for modification. This leads to the introduction of an additional component category, the *generatorfamily*. The effect of a modification (extension/reduction) of the generatorfamily of procedure b in line 3 results in a modification of all b-generators $B_* \in G_b$ created each time the recursive procedure a incarnates. This observation leads to the following definition to distinguish different kinds of component categories in a MoDiS system.

Definition 11 (Component categories)
Let S be a system. S consists of components which belong to one of the following categories:

- *Generatorfamily*
  $\mathcal{G}$ denotes the set of generatorfamilies in S.
- *Generator*
  G denotes the set of generators in S.
- *Incarnation*
  I denotes the set of incarnations in S.

As we have seen in the example above, properties of a generatorfamily, a generator or an incarnation in MoDiS may at first be specified incompletely and be completed at a later point in time when the system is already in execution. Therefore all components posses an attribute which stores their state of completeness. In the following this attribute is written as an upper index ^{c} for a completely and ^{i} for an incompletely specified component.

Definition 12 (Completeness of components)
A component of a MoDiS system is called complete *if all abstract properties of it have been specified or* incomplete *otherwise.*

Definition 13 (Completion of properties)
A completion *of properties is an extension which results in a complete component.*

In the following we abstract from incremental extension of a component and combine multiple extension steps leading to a complete specification of abstract properties in a single completion step.

Depending on the component category and the type of modification we can distinguish the following modifications of abstract properties:

- Generatorfamily completion (GFC)
- Generatorfamily reduction (GFR)
- Generator completion (GC)
- Generator reduction (GR)
- Incarnation completion (IC)
- Incarnation reduction (IR)

A relationship of dependence between properties of components belonging to different categories exist which provides a framework for completion and reduction. These relationships must be well-founded to reduce the large number of possible modifications and make incremental system construction controllable. Therefore the modification of INSEL components is restricted according to the following dependency requirement:
Let $\mathcal{A} \in \mathcal{G}_a$ be a generatorfamily.
$\forall A \in G_a, a_i \in I_a : P(\mathcal{A}) \subseteq P(A) \subseteq P(a)$ [1]
This requirement implies that in a MoDiS system a generator created from an incomplete generatorfamily may be complete or incomplete, an incarnation generated from an incomplete generator may be complete or incomplete but the generatorfamily of an incomplete generator must be incomplete and the generatorfamily and generator of an incomplete incarnation both must be incomplete.

The following conditions ensure that modifications of components preserve the restriction of relationships between properties:

- GFC: $\mathcal{A}^i \overset{extend}{\rightarrow} \mathcal{A}^c \Rightarrow \nexists A \in G_{\mathcal{A}}$
- GFR: $\mathcal{A}^c \overset{reduce}{\rightarrow} \mathcal{A}^i \Rightarrow \nexists A \in G_{\mathcal{A}}$
- GC: $A^i \overset{extend}{\rightarrow} A^c \Rightarrow \nexists a \in I_A$
- GR: $A^c \overset{reduce}{\rightarrow} A^i \Rightarrow \nexists a \in I_A$

[1]for the exact definition of the $\subset$ relation on component properties see [Reh98]

The extension and reduction of components presented so far is summarized in the state transition diagram in figure 6. In the diagram b denotes a component which creates the generator $A \in G_{\mathcal{A}}$ from the generatorfamily $\mathcal{A}$ and an incarnation $a \in I_A$ from the generator A. The vertices represent cartesian products of component categories with the attribute complete or incomplete. Vertical arrows denote extensions or reductions of components and horizontal arrows stand for the creation or deletion of generators and incarnations.

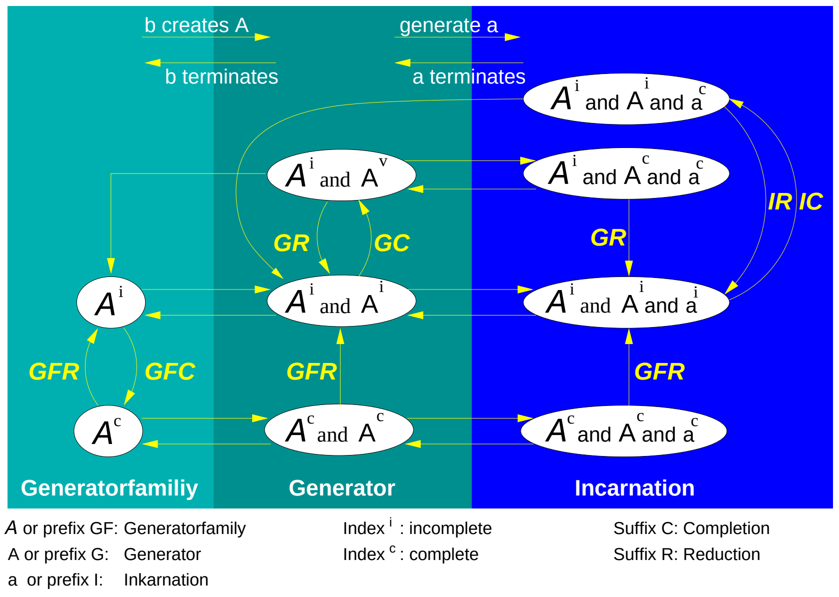


Figure 6: State transition of component evolution

The dynamic modification of component properties presented in this section is realized by the linker and loader FLink which is part of the overall management in MoDiS. The concepts and binding techniques used by FLink are described in section 9.

8 Extended Management Flexibility

The flexibility of resource management in traditional systems is usually limited to the runtime system and kernel. Decisions made by the compiler, linker and loader can not be revised dynamically. For example the mapping of virtual to physical memory may change dynamically but the decision of the compiler whether to create a thread of execution for a component or not can not be revised at runtime. Likewise symbol references resolved by the linker usually do not change while a program is executed although techniques introduced for dynamic linking [Lev00] available in modern operating systems provide the potential to increase flexibility.

In MoDiS flexibility of the resource management covers the complete span from compiler through linker, loader and runtime system to the operating system kernel. Decisions of the compiler or linker based on static analysis only are improved due to information gathered dynamically at runtime. Based on the additional information, the compiler may generate alternative implementations of components which can be integrated in the running system. Alternative implementations are managed by the linker and loader transparently to the application level.

To avoid the overhead of uniform and maladjusted implementations in a distributed system multiple alternative implementations for a single INSEL component may exist simultaneously. According to the execution environment the resource management chooses an implementation which seems best suited in a given situation or may even restart the compiler to generate an additional implementation.

An example for the use of alternative implementations in the context of distributed systems is the realization of access to remote passive objects by dynamic replication, migration and RPC [Win96b]. Figure 7 shows another example to exploit the flexibility of alternative implementations for a local and remote method call. A, B and C represent objects of the distributed system. A and B are realized on node $R1$ whereas C is realized on node $R2$. The call of method X of object B may be implemented as a local function call for A but as a RPC call for C. In a MoDiS system two alternative

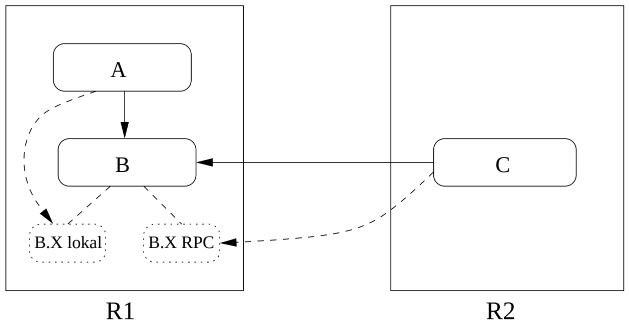


Figure 7: Example for the use of alternative implementations

implementations for $B.X$ may exist simultaneously. The integrated management makes sure that a proper implementation is chosen for local and remote method calls.

To structure the possible uses of alternative implementations MoDiS distinguishes three different kinds of alternatives:

- *Standard alternative* which is the default implementation
- *Node alternative* to replace an implementation on a special node of the system
- *Actor context alternative* to replace an implementation of a component for all references from a special actor context.

A detailed description of flexible management in MoDiS using alternative implementations can be found in [Jek99].

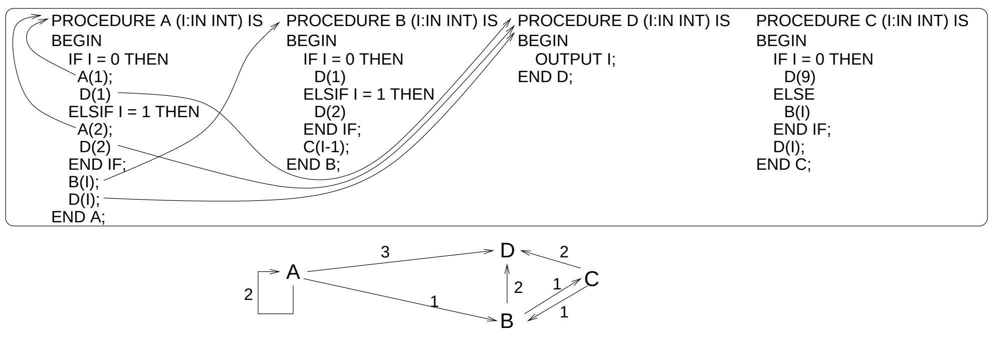
9 Linking and Loading

Both, dynamic modification of the running system presented in section 7 and the choice between alternative realizations described in section 8 are supported by the flexible and incremental linker and loader *FLink* which is an essential part of the overall management in MoDiS. In addition to classical tasks of the linker and loader, FLink provides the ability for incremental and dynamic system construction and extended flexibility using different binding techniques with varying advantages and drawbacks. Using FLink enables the management to choose from different alternatives to resolve symbol references, change symbol references and even reverse the decisions for a program in execution without stopping and restarting it.

9.1 Binding Techniques

The INSEL compiler generates implementations of INSEL components which may have symbol references to other components of the system on the source code level. In order to resolve the references on the assembler level the linker uses different techniques. The use of these techniques is transparent to other parts of the resource management which allows easy composition of management functionality and maximum flexibility. The choice of a binding technique has effects on the performance and flexibility. Figure 8 shows a part of a sample INSEL program together with source code level references which is used to demonstrate the effects of choosing a particular binding technique below.

To resolve symbol references FLink uses the four alternatives described in the following subsubsections.



```
PROCEDURE A (I:IN INT) IS
BEGIN
   IF I = 0 THEN
     A(1);
     D(1)
   ELSIF I = 1 THEN
     A(2);
     D(2)
   END IF;
   B(I);
   D(I);
END A;
```

```
PROCEDURE B (I:IN INT) IS
BEGIN
   IF I = 0 THEN
     D(1)
   ELSIF I = 1 THEN
     D(2)
   END IF;
   C(I-1);
END B;
```

```
PROCEDURE D (I:IN INT) IS
BEGIN
   OUTPUT I;
END D;
```

```
PROCEDURE C (I:IN INT) IS
BEGIN
   IF I = 0 THEN
     D(9)
   ELSE
     B(I)
   END IF;
   D(I);
END C;
```

Figure 8: Example source code and references

9.1.1 Direct Binding

When using *direct binding*, symbol references are directly bound to the virtual address of the referenced component. Figure 9 shows the assembler level bindings between components when using only direct binding to link the INSEL program in figure 8. The advantage of direct binding is efficiency. The

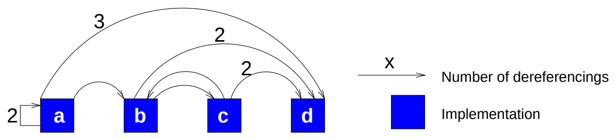


Figure 9: Direct binding

number of references on the assembler level is kept to a minimum and equals the number of references in the INSEL source code. The drawback of this alternative is the lack of flexibility. For example to replace the implementation of component d (of figure 9) by an implementation d', 3+2+2 symbol dereferencings have to be corrected[2]. Due to the high efficiency direct binding is often used in popular linkers.

9.1.2 Indirect Binding

The second binding technique supported by FLink is *indirect binding.* Symbol references are bound to the virtual address of *trampolines.* A trampoline consists of a small number of assembler instructions resulting in a jump to the virtual address of the component referenced on the source code level. The linker generates a trampoline $tr(x)$ for each component x which is bound

[2]provided that d' can not be placed at the same virtual address as d

indirectly. This is illustrated in figure 10 where all components are bound indirectly. The disadvantage of indirect binding is the loss of efficiency due

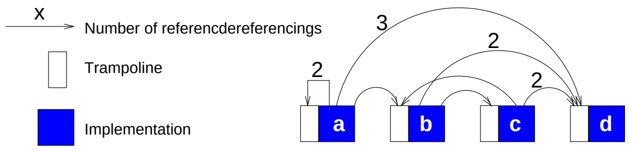


Figure 10: Indirect binding

to the indirection introduced by the trampoline code. The advantage of this binding technique is the increase of flexibility compared to direct binding. All references to an implementation of a component, e.g. d in figure 10, are bundled at the assembler level in a trampoline, $tr(d)$. To replace the implementation d by d' only the trampoline $tr(d)$ has to be replaced by a trampoline $tr(d')$. This is possible as all trampoline codes have the identical number of assembler instructions and therefore occupy the same amount of memory.

9.1.3 Weak Binding

Weak binding is the most flexible binding alternative supported by FLink. A broker consisting of a table and a search method is used to find a referenced component. To replace an implementation the linker only has to adapt the corresponding position in the table. In addition the broker can keep account of the number of referencings or be used for access control in order to temporary block the reference to a particular component. The gain of flexibility correlates with a heavy loss of performance as the search method used in the broker is much more complex than the trampoline code of indirect bindings.

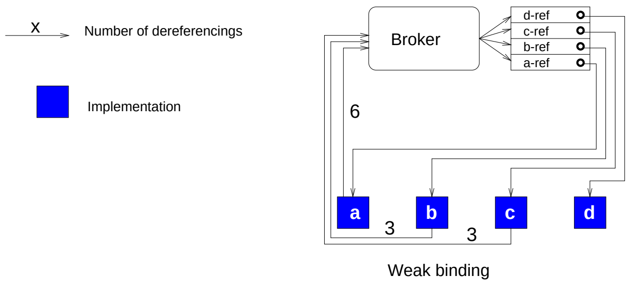


Figure 11: Weak binding

9.1.4 Fail-safe Binding

The last variant supported by FLink, called *fail-safe binding*, is a special form of weak binding. Fail-safe bindings are marked in the broker table and mainly used to reference unknown components. When calling a fail-safe bound component the user is asked to solve the problem interactively by completing the component's properties according to the concepts introduced in section 7.

9.1.5 Costs and gains of alternative binding techniques

The choice of a particular binding technique depends on the required flexibility and efficiency. To compare the advantages and drawbacks we have measured the costs of function calls to a very simple function which just returns its incremented integer argument using the binding techniques described above. The results are summarized in table 1

	Time for n function calls in μs			
n	direct	indirect	weak	fail-safe
100	25	38	2380	4400
1000	230	360	24000	44100
10000	2400	3500	242000	440000

Table 1: Performance differences of binding techniques

10 MoDiS Implementations

10.1 Prototypes

Currently three prototype implementations of the programming language INSEL exist. One is an interpreter integrated in an analyzing and visualization tool, called *DAViT*. It is capable of visualizing all of the 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 through a 10 MBit Ethernet network. For the same platform we realized another implementation, called *EVA* [Rad96], 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* [Win96a] performs a translation of INSEL into C. It focuses on distributed memory management techniques and strategies, such as migration and replication of passive objects.

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 operating system features, as management of users, I/O-system, accounting and so on.

10.2 INSEL Operating System

The prototypical INSEL implementations demonstrated that existing tools to construct software systems do not match our requirements, since they are mostly tailored for UNIX environments. We have implemented a native INSEL to machine-code compiler [Piz97] and a new dynamic linker as sketched in section 9. As a base for the implementation of the INSEL compiler the freely available GNU compiler *gcc* was chosen.

Using these new compiler and linker facilities, some basic services of a distributed operating system in INSEL (distributed scheduling, dynamic loader, etc.) have been implemented. One of the main task of this base system is to support the dynamical extension of the running system at runtime.

This system is currently implemented on a cluster of 16 PCs, interconnected by a 100 MBit/s Fast-Ethernet running a modified Linux kernel.

To visualize the structural dependencies described in section 5 a Java visualization front-end has been programmed.

11 Conclusion

This paper presented the MoDiS approach to cope with the challenges coming along with the shift of paradigm from sequential to parallel and distributed computing. The required homogeneous programming environment is established by following a language-based approach using the object-based, high-level programming language INSEL. The language concepts introduced allow the programmer to solely concentrate on specifying algorithms without being concerned with resource management tasks, as for instance implementing communication facilities or requesting addresses of objects. Key features of the approach are the language-level structuring concepts that were presented in detail.

It was demonstrated that well-structured, 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. According to the top-down oriented approach INSEL-programs are systematically transformed into executables by stepwise refinement. A main characteristic of MoDiS is the integration of all resource management activities involved into a integrated system-wide distributed resource management. The tight coupling of compiler, linker, operating system as well as low-level services offered by the kernel enables to exchange information between resource management parts primarily concerned with static management tasks (e.g. compiler) and those parts performing dynamic management tasks (e.g. operating system kernel).

Incremental system construction and extended management flexibility are of prime importance in MoDiS. Necessitated by the single program approach, MoDiS allows fine grained extension of a system in execution. In order to adapt management decisions to the dynamically changing requirements of the distributed system alternative implementations of components can be integrated. The concepts and techniques to realize both, the incremental system construction and the extension of management flexibility have been presented.

Though we demonstrated the benefits of language-level structuring concepts by means of a specific programming language and the adapted resource management system, we want to emphasize that the underlying concepts and the developed techniques are not unique to this approach. For instance, the dynamic linking techniques can be transfered to other multi-threaded and distributed systems to improve resource management in these systems as well.

References

[Acc86] Mike Accetta. Mach: A new kernel foundation for unix development. Technical report, CS Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[Ada83] Ada. *The Programming Language Ada Reference Manual*, volume 155 of *LNCS*. Springer–Verlag, Berlin, 1983.

[ANS89] ANSA. An engineers introduction to the architecture. Technical Report TR-03-02, APM Ltd., Cambridge, England, November 1989.

[Bal94] Henri E. Bal. Report on the programming language Orca. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1994.

[BLR94] R. Balter, S. Lacourte, and M. Riveill. The Guide language. *The Computer Journal*, 37(6):519–530, 1994.

[BSP$^+$95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safty and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[CAK$^+$96] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems*, Annapolis, 1996.

[CHJ$^+$94] V. Cahill, Ch. Hogan, A. Judge, D. O'Grady, B. Tangney, and P. Taylor. Extensible systems – the Tigger approach. In *Proceedings of the SIGOPS European Workshop*, pages 151–153, 1994.

[CIRM93] R.H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing CHOICES: an object–oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[Con92] Comandos Consortium. *The Comandos Distributed Application Platform*. 1992.

[EP99] C. Eckert and M. Pizka. Improving resource management in distributed systems using language-level structuring concepts. *The Journal of Supercomputing*, 13(1):33–55, January 1999.

[EW95a] C. Eckert and H.-M. Windisch. A top-down driven, object-based approach to application-specific operating system design. In Marvin Theimer Luis-Felipe Cabrera, editor, *Proceedings of the IEEE International Workshop on Object-Orientation in Operating Systems, August 14th - 15th 1995, Lund, Sweden*, pages 153–156, August 1995.

[EW95b] Claudia Eckert and Hans-Michael Windisch. A new approach to match operating systems to application needs. In *Proceedings of*

7th International Conference on Parallel and Distributed Computing and Systems, pages 499–503, Washington, D.C. USA, October 1995.

[Gro96] S. Groh. Designing an efficient resource management system for parallel distributed systems by the use of a graph replacement system. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 215–225, August 1996.

[Jek99] Michail Jekimov. Weiterentwicklung des inkrementellen und verteilten binders flink. Technischer Bericht, TU-München, Systementwicklungsprojekt, 1999.

[KdRB91] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Lev00] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 1 edition, 2000.

[LYI95] R. Lea, Y. Yokote, and J. Itho. Adaptive operating system design using reflection. In *Proceedings of the 5th Workshop on Hot Topics on Operating Systems*, pages 95 – 100, 1995.

[OMG95] OMG. The common object request broker: Architecture and specification. Technical report, Object Management Group, July 1995.

[OSF92] OSF. *Introduction to OSF DCE*. Prentice Hall, Englewood Cliffs, NJ:, 1992.

[PEG97] M. Pizka, C. Eckert, and S. Groh. Evolving software tools for new distributed computing environments. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 87–96, Las Vegas, NV, July 1997.

[Piz97] Markus Pizka. Design and implementation of the GNU insel–compiler gic. Technical report, TU München, April 1997. SFB–Bericht 342/09/97 A TUM–I9713.

[Rad96] Ralph Radermacher. *An Execution Environment with Integrated Load Balancing for Distributed and Parallel Systems*. PhD thesis, Munich, Department of Computer Science, 1996. in german.

[Reh98] Christian Rehn. Inkrementelles und dynamisches Binden in einer verteilten Umgebung. Technischer Bericht, TU-München, Diplomarbeit, 1998.

[Reh00] Christian Rehn. Top-Down Development of a Decentralized Single Address Space Management . In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'2000*, pages 573–579, Las Vegas, NV, June 2000.

[SEJ95] B. Steensgaard and E. E. Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–78, Dezember 1995.

[SESS94] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the vino-kernel. Technical Report TR-34-94, Computer Science, Harvard University, 1994.

[Spi96] P.P. Spies. Concepts for the construction of distributed systems. SFB-Bericht 342/09/96 A TUM-19618, Technische Universität München, Institut für Informatik, March 1996. in german.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison–Wesley, Reading, MA, 2nd edition, 1991.

[vDHT95] L. van Doorn, P. Homburg, and A. S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Proceedings of the 5th Hot Topics on Operating Systems (HotOS) workshop*, pages 86–89, Orcas Island, WA, May 1995.

[Win95] H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applicationsof (PDPTA)*, pages 680–688, University of Georgia, November 1995.

[Win96a] H.-M. Windisch. The distributed programming language insel – concepts and implementation. In *First International Workshop on High-Level Programming Models and Supportive Environments*, pages 17 – 24, April 1996.

[Win96b] Hans Michael Windisch. *Speicherverwaltung fr konzeptionell strukturierte verteilte Systeme*. PhD thesis, Technische Universitt Mnchen, 1996.

[WS95] Z. Wu and R.J. Stroud. Using metaobject protocols to structure operating systems. In Marvin Theimer Luis-Felipe Cabrera, editor, *Proceedings of the IEEE International Workshop on Object-Orientation in Operating Systems, August 14th - 15th 1995, Lund, Sweden*, pages 228 – 231, August 1995.

[Yok92] Y. Yokote. The Apertos reflective operating system – the concept and its implementation. In *Proceedings of the Conference on Object-Orientated Programming Systems, Languages and Applications (OOPSLA)*, pages 414 – 434. ACM Press, 1992.