Project: JADE

JAva Diagnosis Experiments

Projektleitung
Dr. Markus Stumptner
Mitantragsteller
Dr. Franz Wotawa

Institut für Informationssysteme
Technische Universität Wien, Austria, Europe
Phone: (+43) - 1 - 588 01–6129/6129
Fax: (+43) - 1 - 505 53 04
E-Mail: {mst,wotawa}@dbai.tuwien.ac.at

Vienna, March 10, 1999

Abstract

Over the last years, model-based reasoning and model-based diagnosis have become part of the standard repertoire of Artificial Intelligence techniques. The model-based approach has, for a wide spectrum of application domains, provided a solid foundation for knowledge-based systems development, while avoiding the mixing of strategy and domain knowledge that tends to impede development and maintenance of traditional rule-based expert systems.

However, while MBD has been applied successfully in many technical domains, one domain that so far has attracted little interest despite its undeniable potential is that of software debugging. Indeed, a strong research field has developed in the logic programming community based on mechanisms which are much less powerful than those provided by the model-based approach. Therefore, the application of MBD techniques to the software arena is not only a topic of high significance, it also has been little covered by the scientific community.

In a previous project, we have established the groundwork for the theory and practice of applying MBD to software debugging. The experience gained in the development of a model-based tool for supporting the debugging of the hardware specification language VHDL (which is a close relative of the programming language Ada) shows that similar concepts can be applied to functional and imperative programming languages. The project is intended to examine issues such as modeling, fault models and representation of background knowledge about system and programmers, measurement selection, and a number of other issues that have been identified as being relevant for future development of the approach. The Java language has been chosen as a vehicle for the project due to its relatively simple semantics, its ubiquity (if not now, then in the near future), and its imperative and object-oriented characteristics, which place it in the mainstream of programming language evolution over the next decade.

Contents

1 Problem Statement **3**

1.1 Introduction 3

1.2 Model-based Software Diagnosis 4

1.2.1 Basic Definitions 5

1.2.2 Extending MBD to MBSD 8

1.2.3 Diagnosing Programs - Model and Structural Correctness 11

1.3 Object-Oriented Software 12

2 Previous Research in Related Fields **15**

3 JADE Project Objectives **21**

3.1 Objectives Overview 21

3.2 Theoretical Aspects of MBSD 21

3.3 Applying MBSD principles to object-oriented (Java) programs 21

3.4 Use of alternative models for diagnosis 22

3.5 Development of an Intelligent Debugging Environment Assistant (IDEA) 22

3.6 Accumulating a set of example programs 22

3.7 Project Plan 23

3.8 Planned Publications 25

3.9 Future Activities 27

4 Place of Research and Financial Planning **32**

4.1 Place of Research 32

4.2 Staff Resources 32

4.2.1 Existing Staff 32
4.2.2 Requested Staff 33
4.3 Existing Equipment 34
4.4 Requested Equipment 34
4.5 Support for Business Trips 34
4.6 Support for Publications 34

5 Additional Details 35

6 Tables 36

1 Problem Statement

1.1 Introduction

Diagnosis has always been a traditional application area for knowledge based systems. The first and probably most famous expert system, MYCIN, was a diagnosis system that operated in the medical domain, using a knowledge base of production rules. Over the years, the interest in diagnosis as an important problem domain has not abated, although technical application areas now make up a large part of the application cases.

During the decade after the introduction of the MYCIN system, experience gained with actual use of rule-based expert systems showed that the brittleness and maintenance of rule bases presented the major bottleneck in expert system design. As a result, researchers began looking for ways in which the problem could be circumvented. In diagnosis, one way out was seen in the notion of model-based reasoning: The idea that a basic model of the problem domain could be constructed, and this model together with a set of observations about a malfunction of a system or organism would then allow the derivation (via a problem-independent inference mechanism) of the causes of the malfunction. This concept is generally recognized as sound and forms the basis for a large amount of successful research as well as for successful application of fault diagnosis in human-produced artifacts, from electronic appliances to cars and ships.

However, a particular application area that has so far gained little attention is the location of faults in software, or in other words, software debugging. Whereas the examination of relatively simple formal models of debugging is a strong topic in the logic programming community, so far little work has been done for software debugging in general, in particular using the powerful model-based diagnosis techniques.

The authors have been involved in a successful project that provided model-based debugging support for electrical engineers who design integrated circuits using hardware specification languages, a process which is highly similar to software development (and indeed the specification languages used are powerful derivatives of general-purpose or systems programming languages). The tool developed in this project provides the ability to search for the source of errors in actual lifesize hardware designs and identify the statements responsible.

The goal of the JADE project is to extend and examine the application of these techniques to programming languages in general, and in particular to object-oriented languages due to their widespread usage and their position at the forefront of programming language research. The vehicle chosen for the project is the language Java, for a number of reasons:

- Java is a general-purpose language with widespread usage predicted in the coming years
- Java is strongly typed and has relatively simple semantics, in particular compared to C++, the most widespread OOPL.
- Java provides specific features for developing and running applications over the World Wide Web, which is expected to present new challenges for the MBD approach
- Java is expected to be widely used in the future by people with little programming experience as it becomes the language of choice for developing small, ad-hoc WWW applications. In this context, a knowledge-based debugger that guides an inexperienced user through the debugging cycle would be especially important.

The goal of project JADE is the examination of the formal underpinnings required for using Model-Based Diagnosis in a software development environment, and the development of an experimental system for solving actual diagnosis cases with Java programs.

1.2 Model-based Software Diagnosis

Model-based Software Diagnosis (MBSD) is the application of model-based diagnosis to software debugging. As an introduction to model-based diagnosis in general, see [Rei87, dKW87, GSW89] regarding consistency-based and [CDT89, CT90] regarding abductive diagnosis. Initially, the concept of MBSD was formulated in [CFD93] for a pure logic programming language. Instead of considering a system model of the physical device, we use the program or, to be exact, a model of the program which is derived using the semantics of the language (see Figure 1.2). From this model, we derive the actual behavior which has to be compared with the intended behavior of the program. The intended behavior is necessarily provided by an outside source (called an *oracle* by Shapiro [Sha83]). If differences (so-called discrepancies) to the intended behavior are found, the computation of diagnoses is initiated. The oracle itself can be an informal or formal specification, the user, or the result of comparing the behavior of the design for particular test runs with test runs of a second program implementing the intended behavior.

The work in [CFD93] was of relatively limited scope, and by concentrating on pure logic programming, avoided a number of problems that have to be dealt with when discussing the application of MBSD in a real-world environment.

In this section we first give some basic definitions for MBD. Then we show how these principles can be applied to programs, by converting a short JADE example to a system description usable for diagnosis.

1.2.1 Basic Definitions

The use of model-based diagnosis is based on a few basic principles. First, systems are assumed to be composeable, i.e., a system can be described in terms of a set of of components and their interconnections. Second, there must be a theory describing the behavior of the components, i.e., a model. Note that these conditions can be considered to hold for program debugging.

Definition 1.1 (Diagnosis System (Reiter)) *A (diagnosis) system is a pair* $(SD, COMP)$ *where:*

$$
\begin{tikzcd}
\text{Actual Program} \arrow[d] & \text{Diagnosis} & \text{Oracle} \arrow[d] \\
\text{Behavior of the program} \arrow[r] & \text{Behavior Differences} \arrow[u] & \text{Intended Behavior} \arrow[l]
\end{tikzcd}
$$

Figure 1: Program Diagnosis

1. *SD is the system description, describing the behavior of components and their interconnections.*

2. *$COMP$, the system components, is a finite set of constants.*

The input/output behavior of components must be defined at least in the case where a component is assumed to behave correctly. It is also possible to define behavior for specific cases where the component is faulty, and to this end a function is introduced associating components with behavior modes.

Definition 1.2 (Component Modes) *$MODES$ is a finite set of constants including at least* undef *and* ok*.*

The set $MODES$ contains all behavior modes for components. At least the two distinct modes $undef$ and ok must be element of $MODES$. We also assume the existence of a function *modes* which associates a subset of $MODES$ with every component (again, including $undef$ and ok for every component).

$modes : COMP \mapsto 2^{MODES} \cup \{ok, undef\}$

Usually only the predicates ab or ok are used to indicate whether a component works correctly or not (with $ok = \neg ab$ and vice versa). It is worth noting that the system description may contain a behavior description any given mode, but this is not necessarily the case. The only mode that is always assumed to have a defined behavior is ok.

As an example, we look at the system description for an and gate with the two fault modes stuck-at-one (em s1) and stuck-at-zero (*s0*).

$$
\begin{aligned}
&ok(C) \wedge andGate(C) \Rightarrow out(C) = and(in_1(C), in_2(C))\\
&s1(C) \wedge andGate(C) \Rightarrow out(C) = 1\\
&s0(C) \wedge andGate(C) \Rightarrow out(C) = 0
\end{aligned}
$$

A diagnosis problem is stated by giving a system description and a set of observations. If the observations contradict the expected behavior there must be a subset of $COMP$ whose incorrectness would explain the observations.

Definition 1.3 (Diagnosis Problem) *A diagnosis problem is a triple $(SD, COMP, OBS)$ where SD is the system description of the system to* *be diagnosed, COMP is the set of components in the system, and OBS is a finite set of observations.*

Returning to the discussion of software errors from Section 1.1, we note that since a misbehavior is the result of an incorrect part of the program, it follows that a natural way of classifying errors is to look at how the program must be altered to remove the error. Thus, for example, omitting an operator (as in writing "`i`" instead of "`i+1`" is a different kind of error from using the wrong variable (as in writing "`a[i]`" instead of "`a[j]`"). The way in which an expression has been incorrectly used in a program can therefore be likened to the fault modes of hardware diagnosis (and, in the same manner as a fault mode, a particular assumption for the "corrected" statement limits the difference in behavior between the correct and incorrect system). We therefore define the notion of component modes for a program (where the components are the statements or expressions in the code of the program). The behavior of components must be defined at least in the case where a component is assumed to behave correctly, but in general, specific fault modes will also imply a specific behavior.

Definition 1.4 (Mode Assignment) *A mode assignment* μ *is a set of ground facts of the form* $m(c)$ *where* $m \in MODES$ *and* $c \in COMP$*, such that no component is assigned two different modes.*

Definition 1.5 (Diagnosis) *Let* $(SD, COMP, OBS)$ *be a diagnosis problem. A mode assignment* Δ *is a diagnosis* $(DIAG(\Delta))$ *iff* $SD \cup OBS \cup \Delta \cup \{ok(c) | c \in COMP \setminus \gamma(\Delta)\}$ *is consistent.*

The task of a diagnosis engine is to determine mode assignments for every component which explain the error. For the sake of brevity, we assume in this paper that OBS corresponds to the one particular output value expected from a query. To limit the search space, we will only look for minimal diagnoses in the spirit of [Rei87], and define an appropriate order on mode assignments.

Definition 1.6 (Mode Assignment Subsumption) *Let* MA_1, MA_2 *be two mode assignments. We say that* MA_1 *subsumes* MA_2 $(MA_1 \sqsubseteq MA_2)$ *if the following holds:*

$\forall m(c) \in MA_1 : m(c) \in MA_2 \vee undef(c) \in MA_2$

The above definition prefers mode assignments with smaller cardinality and mode assignments which avoid the undefined behavior mode[1].

Definition 1.7 (Minimal Diagnosis)
Let Δ be a diagnosis for $(SD, COMP, OBS)$. We say Δ is minimal iff there exists no diagnosis subsuming Δ.

$MIN(\Delta) \Leftrightarrow \nexists \Delta' : DIAG(\Delta') \wedge \Delta' \sqsubseteq \Delta).$

The aim is now to determine minimal diagnoses for a given system and set of observations. To do this we can use a slightly modified version of Reiters [Rei87] diagnosis algorithm (corrected by [GSW89]) which is described in [Wot96].

In our *and* gate example, $s0(and)$ is a minimal diagnosis (and preferred over $undef(and)$).

1.2.2 Extending MBD to MBSD

The authors possess significant experience in the application of MBSD to large, real world programs, as presented in [FSW96, SW96d]. The basis of the success of this project which dealt with the diagnosis of errors in the programs written in the language VHDL, was the use of an abstract model of program behavior. In [SW96c, SW96b], we presented a more detailed diagnosis model for VHDL programs (but covering only a subset of the whole language). In [SW96a], several approaches are described and compared for applying MBSD to a simple functional programming language . However, strategies for efficient software debugging and the application to a more sophisticated language are still open research issues.

To use MBD for debugging, programs must be converted into a system description, and testcases (specifying the I/O behavior) into observations. We give an example for this conversion using the following *Java* program implementing an or-function.

```
public boolean myOr(boolean i1, boolean i2) {
     return not(and(not(i1), not(i2))); };
```

[1] Since the *undef* mode does not explain why a component may fail.

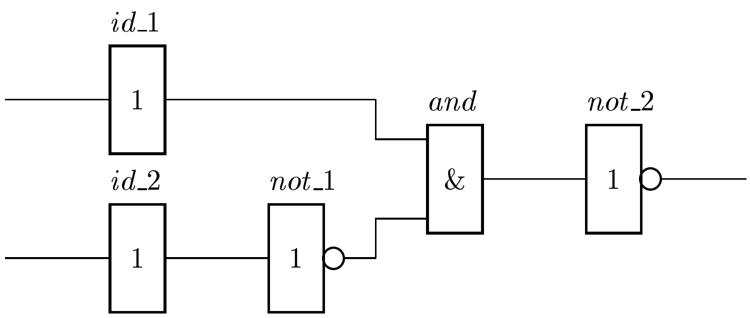

Figure 2: The `myOrW` function as digital circuit

Assume further the testcase *i1 = false, i2 = false* is executed with the following, incorrect implementation:

```
public boolean myOrW(boolean i1, boolean i2) {
     return not(and(i1, not(i2))); };
```

Executing `myOrW` using the testcase leads to the wrong result *true* instead of the expected *false*. To diagnose `myOrW` we must convert it to a system description. This conversion can be done compositionally by converting functions and variable accesses into their associated components. In our case we denote the first `not` by *not_1*, `and` by *and*, `i1` by *id_1*, the second `not` by *not_2*, and finally the variable access `i2` by *id_2*. The result of the conversion of `myOrW` is shown in figure 1.2.2 in form of a digital circuit. The model of each used component is given by the underlying function.

$ok(F) \wedge idFunc(F) \Rightarrow (out(F) = in(F))$
$ok(F) \wedge notFunc(F) \Rightarrow (out(F) = not(in(F)))$
$ok(F) \wedge andFunc(F) \Rightarrow (out(F) = and(in1(F), in2(F)))$

The interconnection can be formalized by

$out(id\_1) = in1(and)$
$out(id\_2) = in(not\_1)$

$out(not\_1) = in2(and)$
$out(and) = in(not\_2)$

and observations OBS (related to the given testcase) by

$in(id\_1) = false$
$in(id\_2) = false$
$out(not\_2) = true$

By defining that *idFunc(id_1)*, *idFunc(id_2)*, *notFunc(not_1)*, *notFunc(not_2)*, *andFunc(and)* we can derive a diagnosis using the standard MBD algorithm:

{ *undef(id_1)* }, { *undef(and)* }, { *undef(not_2)* }.

Given several additional assumptions, we can restrict the result further.

1. Assume that *and* is the only binary function defined on *boolean* values. In this case, we cannot find a simple replacement for it.
2. The replacements must cause only small program changes. This, together with the previous assumption, eliminates the second diagnosis.
3. Replacing *not_2* by an identity function solves our problem for this testcase.

An advantage of MBD (and MBSD) is that replacements can be naturally used by associating them with fault modes and their associated fault models. A replacement of function calls can be expressed by $replace_R(E)$ where R denotes the new function and E the function which is to be replaced. An appropriate model is given by

$replace_{idFunc}(E) \wedge notFunc(E) \Rightarrow (out(E) = in(E))$
$replace_{notFunc}(E) \wedge idFunc(E) \Rightarrow (out(E) = not(in(E)))$

Note that such models must be introduced for every replacement which is to be considered for diagnosis. One important point within the JADE project is to guide the search for the best replacements. The set of modes also depends on the set of operations and functions used. For example, specific modes for *conditionals*, *loops*, *recursive functions* are possible. Therefore, the second important aspect of the project is the development of program models suitable for MBSD. As a matter of, fact software diagnosis must in principle deal with an unlimited number of fault models including structural faults. This is different from hardware diagnosis where only a limited number of fault modes is available.

1.2.3 Diagnosing Programs - Model and Structural Correctness

In a certain respect the problem of diagnosing programs is unique in the realm of model-based reasoning. In conventional model-based diagnosis, the system description is an exact specification not only of the overall behavior of the system, but of its individual parts. For example, when diagnosing the hardware implementation of a 16-bit adder, the adder's system description will describe the behavior of the logical gates from which the adder is composed. A fault is assumed to occur because one of the components does not act according to its specification.

In debugging a program, however, the assumption that the specification will be a complete representation of the structure of the artifact is obviously invalid – if that were the case, the program would be correct and need no debugging. In our case, the internal structure and the way in which the behavior is described may differ widely between the specification and the implementation code. We are therefore forced to base our model on analysis of the code of the implemented program itself. That implies, however, that it is the model that reflects the incorrectness of the system (program) and whose output (the results of a test run) is confronted with observations that are correct (the specification), whereas in traditional diagnosis problems, the model is correct and it is the observations, made from the behavior of the actual system, that reflect on the incorrect behavior. In addition, the question of how a design defect may manifest itself in the model leads us to the related issue of so-called structural faults.

Structural faults are faults that do not occur because a component is functioning incorrectly, but because there is a missing or additional connection between two components, as in a bridge fault in electrical engineering. The bridge fault problem was first noted by Davis [Dav84] and mainly excluded from consideration in subsequent work on diagnosis. However, it is very relevant when diagnosis is applied to designs, and in particular, software. The use of an incorrect argument in an expression (e.g., by using a different variable name, switching the ordering of arguments), or the omission of part of a complex expression constitute typical examples of such faults.

The usual way for dealing with structural faults is to assume the existence of a different, complementary model that allows to reason about the likelihood of such faults (i.e., modelling of spatial neighbourhood in the case of bridge faults). In software, such models could take the shape of considering name misspellings, variable switchings, or attempts to repair expressions (i.e., synthesize missing parts) to provide correct functionality. The possibility of expressing structural faults through fault modes has been recently examined in [Böt95]. This is an open research issue.

1.3 Object-Oriented Software

Object-oriented concepts have achieved widespread recognition in the last decade. The term was first used to describe the programming language Simula[BDMN73], but object-oriented concepts were also developed over time in such diverse fields as Artificial Intelligence, data modelling, and user interface design. Widespread use of object-oriented concepts was engendered by the release of object-oriented programming languages (OOPL's) suitable for real-world application development (such as C++[Str86], Smalltalk[GR83], and Objective C[Cox86]) in the 1980s. The basic foundation for these languages is the notion that a program can be viewed (and designed) as a system of interacting objects. An object represents the encapsulation of data together with the operations used to process these data. There is no exhaustive, generally accepted definition of an object-oriented system. However, there is a consensus about certain properties that are expected from an object-oriented language.

Encapsulation: An object is considered to be an instance of an abstract data type. It provides an interface (defined in terms of operations and their signatures) to the outside world. Hidden from the outside are the internal state of the object (except insofar as it can be accessed through operations defined in the interface) and the implementations of the operations. The internal state can possess a complex structure, consisting of multiple so-called instance variables. (This term is derived from Smalltalk usage, and many languages differ, e.g., use of the term "data member" in C++.) Instance variables can contain atomic data values or references to other complex objects. Direct access to the structure is usually discouraged as being contrary to the encapsulation principle.

Object Identity: Each object is considered to have an identity, i.e., possess a property that distinguishes it even from other objects with the same structure and the same state. This is often represented by a so-called Object ID (OID). The identity of an object remains unchanged throughout its lifetime, regardless of changes in its state.

Classification: Similar objects are grouped together in classes. A class thus can be considered to be the definition of a set of objects with similar properties. Some authors define classes as "implementations of abstract data types". Since the class of an object defines the structure and operations of the object, the notion of class is related to the notion of a type in conventional programming languages. In fact, some authors define a language as "class-based" when it allows user-defined types (i.e., classes) that are in all respects treated the same as the built-in types of that language, e.g., they can be used as return types for functions, as element types for sets and arrays [Str87, Weg87]. However, in OO languages that support strong typing, the notions of "class" and "type" are usually considered separate, even if they often coincide in practice. (See below.)

Inheritance: The existence of an inheritance relationship means that a given class A (called the *superclass*) can share parts (or all) of its definition with a second class B (called the *subclass*). Class B therefore *inherits* the definition of class A. However, inheritance is not prescriptive, i.e., the definition of class B can be extended or modified with respect to class A, with the programmer or designer adding new instance variables and operations, or overriding the implementation of existing operations. This relationship is therefore incremental, i.e., the unchanged parts of the specification and implementation of B do not need to be repeated in code or design. Redundancy is kept to a minimum. Often, classes are defined that provide a specification and partial implementation for a certain problem domain, leaving the implementation of the remaining methods to individual subclasses. Such classes that cannot be meaningfully instantiated without first filling the gaps in their definition through subclassing are called *abstract classes*. Inheritance can be an effective tool for describing the relationships between different classes in design and implementation. A large part of the work done in the field of object-oriented software production deals with the correct uses of inheritance relationships in design and implementation.

The existence of an inheritance relationship is considered the crucial difference between object-oriented languages and languages supporting modules and abstract data types such as Modula-2 and Ada 83 (which are often subsumed under the designation *object-based*, as opposed to object-oriented).

Different kinds of inheritance relationships are supported by the languages in use today. Originally, inheritance was based on the concept of a hierarchy (i.e., a tree) where each class has at most a single superclass. Newer languages often support the generalized form of multiple inheritance, where the inheritance relationships form a directed acyclic graph.

Polymorphism In an object-oriented context, Polymorphism denotes the ability to apply the same operation to instances of different classes in a system. These classes may provide different implementations for the operation, so that different code is executed depending on the class of the object. This concept (runtime selection of the code to be executed) is called *dynamic binding* or *late binding*. The separation between operation name and choice of implementation gave rise to the designation *message passing* for the call of a dynamically bound operation. The implementation of the operation that is executed is called a *method*.

Polymorphism is especially convenient in combination with inheritance, since inheritance relationships can be used to specify the messages supported by a class (called its *protocol*), whereas the implementations can be modified in subclasses.

In principle, object-oriented programs can be considered a system of encapsulated, interacting classes. Object-oriented languages thus have a modular structure, with classes taking the role of modules, or packages, in object-based languages. Organizing the structure of a software system around the data that it handles instead of the particular functions used for processing is considered an approach that results in a more stable, maintenance-friendly software architecture. Use of inheritance reduces the redundancy of code (i.e., software reuse in the small) and provides a semantic structure for applications by delineating similarities between classes.

Experience has shown that while reuse of whole classes and their partial modification through inheritance is a significant factor in furthering software reuse, classes are too small a unit to fully realize the possibilities of reuse. Research in reuse of object-oriented software now focuses on frameworks or patterns - groups of classes that cooperate to provide certain services or application functions, and are instantiated or refined in concert.

2 Previous Research in Related Fields

1. [Sha83] introduces a theoretical framework for *algorithmic program debugging* and several algorithms. In the introduction several arguments of why testing and debugging can not be replaced by program verification are given. The two goals *bug identification* and *bug correction* of debugging are identified. Although three types of incorrect program behavior can be distinguished, (1) a program can terminate with a wrong output, (2) a program can terminate with no output, and (3) nontermination. For all errors [Sha83] provides algorithms to fix it. Prolog implementations, computational and query complexities for all algorithms are given. In addition this paper shows how debug algorithms can in principle be used for program generation out of examples.

   Comments. *[Sha83] describes program debugging not only for logic programs but limits the applicability to programs consisting out of* procedures*. The proposed algorithms only fix faults on this procedure level (and not inside procedures). On the other hand all algorithms are interactive and need heavy user interaction to find a fault. Automatic program debugging which is one of the aims of* JADE *is therefore not possible using Shapiro's approach.*

2. [Fer87] and [Llo87] Both papers presents declarative algorithms for software debugging of *Prolog* programs. Declarative Prolog constructs such as cuts are not considered. In [Fer87] variables are used in the semantics, and therefore no information potentially contained in logic programs is lost, e.g., consider a goal `G(x)`. A semantics not using

variables cannot formalize the difference between a substitution result θ(x) = 1 and θ(x) = x.

Comments. *The restrictions of [Sha83] related to the application domain still remains in both papers.*

3. [HM93] and [NF93] Based on [Sha83] both authors presents way of using *algorithmic debugging techniques* for debugging of functional programs such as *Lisp* or *Miranda*. In [HM93] a graph showing all function calls during execution is created and used for debugging. To optimize search a way of computing optimal pathes is presented. [NF93] on the other hand uses [Sha83] and applies the basic ideas to lazy functional programs where as difficulty computations do not take place in an order one might expect.

   Comments. *Both papers uses standard* algorithmic debugging techniques *for locating faults in functional programs. The drawbacks of [Sha83] still remains but interesting ideas can be located. One idea is the use of dependencies between program entities, i.e., functions or variables to debug large programs. However, debugging small programs, i.e., function bodies or procedures is not provided by them without user interaction. Since both papers relay on [Sha83] automation of debugging is not possible.*

4. [Joh86] describes the PROUST system for the interactive tutoring of programming novices, based on an approach called *intention-based Diagnosis*. Instead of attempting to find faults in the structure of the program, PROUST attempts to analyze the goals of the programmer instead. Design goals were a high degree of correctness and detailed explanation of why the error found constitutes a bug. The requirement for correctness also leads to the interesting feature that PROUST explicitly refrains from making estimates when the analysis of the intentions embodied in the program does not match.

   Comments. *The research goals behind PROUST are somewhat different from ours; they aim at diagnosis of very small programs (on the order of a page of source code) in a strictly controlled environment, where the scope of student programs is known and predefined. These assumptions are significantly more restrictive than the assumptions behind the* JADE *project.*

5. Like the previous entry, the work described in [Mur88] aims at providing support specifically to novice programmers. The **Talus** system examines semantic properties of students' LISP programs by attempting to deduce equivalence theorems with a predefined reference function. A Boyer-Moore style theorem prover is used for the deduction process. The system is able to handle considerable differences in structure and decomposition between the reference and student programs.

   Comments. **Talus** *is able to handle a wide range of alternate student programs as answers to a given problem. However, the goals are completely different from our project. The basic assumption is that a full implementation of the problem (in the same language) must exist as a reference. The system is therefore, even stronger than PROUST [Joh86], only applicable in a classroom environment. The requirement that a complete and correct implementation must exist makes it effectively useless for non-tutoring purposes.*

6. In [CFD93] the first application of model-based diagnosis to the software domain has been proposed. This paper introduces a way of using MBD by removing and adding new horn clauses to prolog programs. In difference to [Sha83] less user interaction must be performed to fix the fault. Although not only fault localization is done. Instead a possible correction of the faulty program is computed. To restrict computation and to focus on the most possible fault [CFD93] introduces a definition for minimal program variants.

   Comments. *This paper presents one way of using MBD to software debugging but restricts the view to logic programs, e.g,* Prolog *programs. An evaluation of the proposed method including an implementation and an application to larger programs is missing, too. On the other hand [CFD93] only focus of logic programs consisting out of horn clauses. Negations and non-declarative prolog predicates are not considered. Therefore only basic ideas can be used for* JADE *after adapting them.*

7. [BP94] and [Bon94] extents [CFD93]. Failures related to definitions are corrected and an implementation is presented. The problem of fixing a bug in logic programs using a standard consistency-based diagnosis approach is solved.

Comments. *Similar to [CFD93] no practical experiences in applying the presented approach to larger program has been given. Also repair has not been tackled and left for further research.*

8. [CWH94] describes a diagnosis approach for gate-level (boolean) circuits. Possible locations for errors are found by tracing lines between gates backwards (similar to the functional dependency analysis in [FSW96]). The system then attempts to repair the error by replacing parts of the circuit with other constructs based on a set of heuristic error types which have been described in the verification literature as covering almost all actual cases of design errors at gate level. Equivalences between different types of repairs are computed. The paper also presents performance results for circuits with up to 4000 gates.

   Comments. *The approach is interesting because it combines error location and repair, but works only for gate level boolean circuits, both in terms of semantics and because it requires exact descriptions for almost all types of designer errors, the handling of which is then hardwired into the diagnosis algorithms. This would be impossible in a general software environment. Finally, the work is strictly limited to single faults.*

9. [GC89, AC91, SSGP92] are a family of papers which describe the use of model-based diagnosis in a design environment. A functional model of the design is produced, and if the design does not live up to the specification, MBD techniques are used to attempt to locate defective places in the design.

   Comments. *This work is interesting insofar as it also considers the use of MBD in a design environment, with a resulting relaxation of a number of strict assumptions that are usually made in diagnosis, for example the notion of structural correctness. However, the work is based on the existence of a complete prespecification of all functions that might be provided by particular constructions in the design environment described. The corresponding notion (e.g., spell out all reasons why a programmer would want to use an if statement in a program), is clearly not feasible in a programming environment.*

10. [Liv94] discusses the use of a functional representation in the debugging of software. Reasoning about function is supposed to reduce the problem of structural faults (when a superfluous or missing component is the source of the error). The approach relies on symbolic execution of the functional specification to find which requirements or post-conditions will not be fulfilled and point at an error.

    Comments. *The paper does not discuss the applicability of the approach to life-size programs. The example only shows the method applied to a single loop. As in the previous case, the required existence of a functional specification limits the applicability of the work.*

11. [Rie93] describes the application of Model-Based Diagnosis to network communication protocols. The system attempts to locate and identify errors in OSI networks by examining a model of the different protocol entities expressed in terms of *finite transducers*, a special kind of finite automata. A constraint-based implementation is used to find diagnoses. The thesis also discusses further issues such as designing for diagnosis or integrating the diagnostic tools into communications architectures.

    Comments. *The thesis is interesting in that it covers a domain that has a number of similarities to software (its entities are expressed in terms of software after all): Not only the behavior, but the actual structure, communications patterns, and architecture of the systems diagnosed may be at fault. However, the protocol area has the advantage of limited expressibility (exemplified by the use of finite state machines as the underlying model), which means the results and approaches are too weak and not directly applicable to the software domain.*

12. [BH93, BH95] presents another approach to program debugging using probability measurements to guide diagnosis. The program debugging process is divided into parts. In the first one, program parts that may cause a discrepancy are computed. This is done by tracing the incorrect output back to the inputs and collecting the involved statements. To improve the diagnosis result, i.e., focus on the most probable statements, a belief network (Bayesian Network) is used. Statement types

and fault types are related by a probability which is given by an "expert". This probability can be used to compute a probability of failure for every statement or execution path. Burnell et al, present the result that their program diagnosis is about 75 percent correct.

Comments. *The high value of diagnosis correctness is only related to the diagnosis of one program and it is not very likely that diagnosing another program will also lead to such a high value. This proposition can be argued by saying that a general fault probability which can be assigned to every component depending neither on the programmer nor on the specific programming language used, is very unlikely to find. Therefore the approach of Burnell et al can only be used for maintaining an existing (large) program. Using the same approach for another one requires a recomputation of probabilities and a fine tuning.*

3 JADE Project Objectives

3.1 Objectives Overview

The JADE objectives can be summarized as follows:

1. Development of a theory of model-based software debugging.
2. Description of Java semantics in terms of logical models usable for diagnosis.
3. Examination of the use of alternative models for scalable diagnosis and their flexible use in multi-model in an diagnosis environment.
4. Development of an intelligent debugging environment for Java programs based on the theoretic results.
5. Collection of example programs and establishment of a set of benchmarks for diagnosis evaluation.

3.2 Theoretical Aspects of MBSD

This part of the project will aim at the development of a theory of model-based software debugging, building on the results gained in our previous work [FSW96, SW96d, SW96c, SW96b, SW96a]. Examination of different approaches to the modelling of programming language constructs, in particular with regard to object-oriented languages. The results gained will enter into the design of the actual diagnosis system for debugging Java code.

3.3 Applying MBSD principles to object-oriented (Java) programs

The use of MBSD for Java requires the description of Java semantics in terms of logical models usable for diagnosis. Depending on a the characteristics of individual programs, different models may be developed for use in different situations (see next section). This part of the project will also deal with any required extensions to the general diagnosis principles for dealing with special Java language constructs. (An example for this would be the way in which the model defined in [FSW96] is based on the concurrent programming constructs of the VHDL language.)

3.4 Use of alternative models for diagnosis

A topic of high relevance for the practical application of diagnosis algorithms to programs is the choice of the correct representation. This part of the project will examine the use of alternative models for diagnosis as mentioned in the previous section. For example, large programs may require a more abstract representation to make diagnosis feasible. A goal is also the definition and examination of multi-model diagnosis algorithms which can, e.g., switch between different levels of abstraction for a more efficient and flexible diagnosis process.

3.5 Development of an Intelligent Debugging Environment Assistant (IDEA)

This is the most strongly implementation-oriented part of the project. It will deal with the development of an intelligent debugging environment for Java programs based on the theoretic results from the previous sections. This would also require (for experimentation and instrumentation purposes) the implementation of a Java interpreter.

Work on this part of the project would proceed in parallel with the more theoretical stages to guarantee that the environment will be ready for use when needed for experimenting with different diagnosis models and algorithms.

3.6 Accumulating a set of example programs

Evaluation and testing of the diagnosis algorithms will require the compilation of a set of example programs of different size, complexity, and application domain. This set of programs will be used as the base of a set of benchmarks for the (performance- and quality-related) evaluation of different diagnosis approaches.

3.7 Project Plan

One part of the project will be devoted to theoretical aspects, the second half will be more implementation-oriented and will aim at building a prototype Intelligent Debugging Environment Assistant (IDEA) based on the concepts developed in the theoretical part of the project. Work on the collection of benchmark diagnosis problems will proceed in parallel.

It is necessary that all research assistants start their activities with the start of the project, because at the beginning they will be involved in theoretical research and the development of the Java debugging environment.

The topics listed above imply that the project will be divided into a number of theoretical (**T**) and implementation-oriented (**I**) tasks.

T1. Theoretical investigation of MBSD. This includes several subtasks:

1. Compiling an overview of the existing literature
2. Developing a classification of programming errors, with input from the literature on programmer cognition: types and occurrences of errors, effects of different types of errors etc.
3. Development of a representation for causality in programs
4. Development of a representation of imperative languages for diagnosis
5. Adaptation and development of MBD algorithms for MBSD

T2. Applying MBSD principles to object-oriented (Java) programs.

1. Examination of representation of the semantics Java constructs in a diagnosis context. This includes data types and control structures, as well as distribution and possibly security aspects
2. Examination of class boundaries as abstraction borders for diagnosis. This is of general interest for OO languages, and applies to Java as well.
3. Modeling of the standard Java class library for MBSD. This might include functional models describing the intended use of these classes in application programs.

T3. Use of alternative models for diagnosis.

1. Evaluation of implementation results gained in the initial stages of task **I1** (see below).
2. Examination of the effects of extending models by explicit representation for functional dependencies in programs.
3. Development of additional fault models and estimates on the occurrences of different fault models.

Task T1 will last throughout the first year of the project. T2 will start after six months (when the groundwork has been laid by T1) and will last till the end of the project (to allow for reevaluation and improvements based on experience with the actual implementation). T3 will start after T1 is concluded and last up to the end of the project.

The more practical aspects of the project comprise tasks **I1** and **I2**.

I1. Development of an Intelligent Debugging Environment Assistant (IDEA). This task will consist of a number of subtasks. The writing of design and system documentation will occur in parallel with the different subtasks.

1. Implementation of the Inference and generic MBD class library
2. Java representation specific classes
3. Java compiler classes
4. Testing compiler on example programs from **I2**
5. MBSD classes
6. Testing and evaluation of diagnosis
7. User interface development (implementation+testing)
8. Global system test
9. User manuals
10. Implementation of alternate models developed in **T3**.

I2. Compilation of a set of example programs

1. Definition of a catalog of examples
2. Establishing an ordering of implementation tasks according to complexity and relevance of the examples

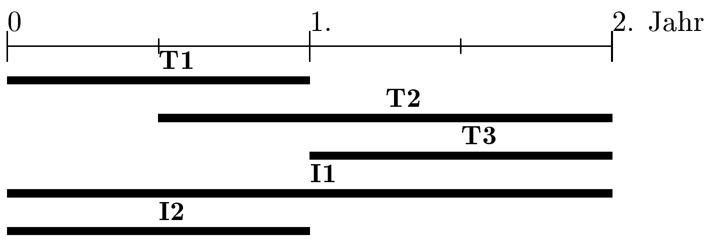

Figure 3: Project Plan

3. Implementation of simple programs (e.g., nonrecursive, simple tasks like sorting and searching)
4. Implementation of complex example programs

Task **I1** will start at the beginning of the project, so that the basic development environment framework will be ready for experimentation in the second year. The results of task **I2** will be required for experimentation, and therefore the task will need to be largely finished after the first project year.

Task **I1** will be the responsibility of a research assistant, while task **I2** will be largely covered by student projects under the supervision of a research assistant.

3.8 Planned Publications

Task T1 should result in about 3 refereed scientific publications. Task T2 is of a somewhat interdisciplinary nature and can be expected to result in about 4 publications. Task T3 is expected to result in 1 or 2 publications.

Each of the tasks I1 and I2 should also result in a publication, either in a journal or a conference.

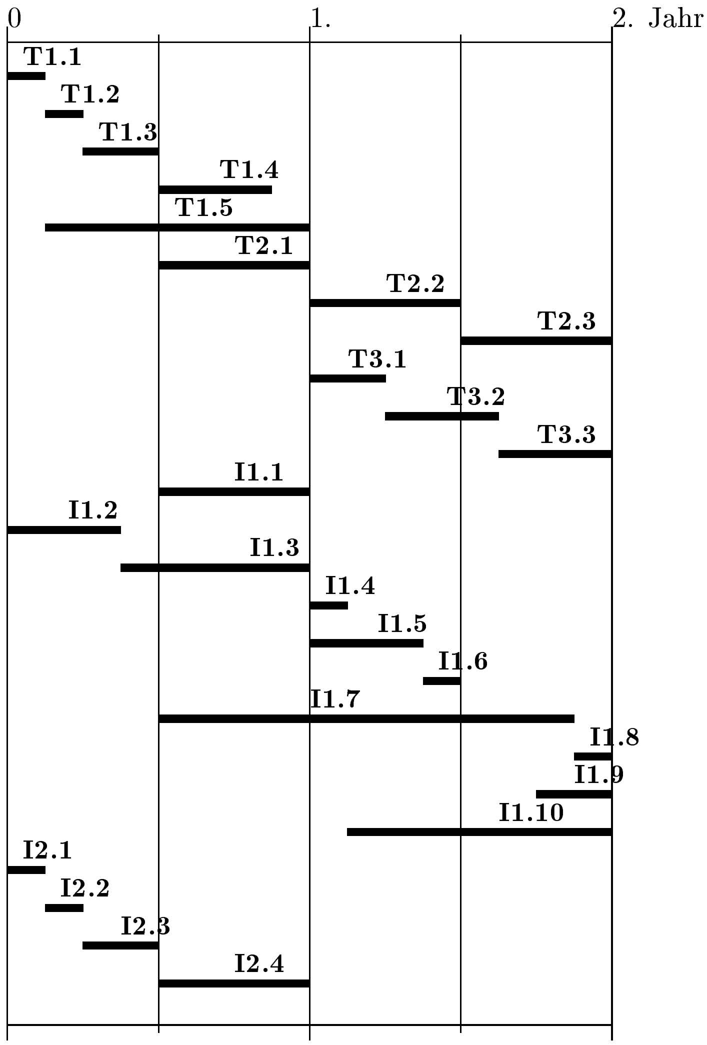

Figure 4: Detailed Project Plan

3.9 Future Activities

An obvious goal for future research will be the generalization of the results of the project to other programming languages with similar semantics (e.g., Smalltalk, C++, Objective C).

References

[AC91] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *Proc. IEEE Knowledge-Based Software Engineering Conference*, pages 136 – 143. IEEE, 1991.

[BDMN73] G. Birtwistle, Ole Johan Dahl, B. Myhrtag, and Kristen Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973.

[BH93] Lisa Burnell and Eric Horvitz. A Synthesis of Logical and Probabilistic Reasoning for Program Understanding and Debugging. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, pages 285 –291, 1993.

[BH95] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, pages 31 – 41, 1995.

[Bon94] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[Böt95] Claudia Böttcher. No faults in structure? How to diagnose hidden interaction. In *Proc. IJCAI*, Montreal, August 1995.

[BP94] G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.

[CDT89] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. A theory of diagnosis for incomplete causal models. In *Proc. IJCAI*, pages 1311–1317, Detroit, August 1989. Morgan Kaufmann.

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann.

[Cox86] Brad J. Cox. *Object Oriented Programming – An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[CT90] Luca Console and Pietro Torasso. Integrating models of correct behavior into abductive diagnosis. In *Proc. ECAI*, pages 160–166, Stockholm, August 1990. Pitman Publishing.

[CWH94] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320–332, 1994.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[Fer87] Gerard Ferrand. Error Diagnosis in Logic Programming, An Adaption of E.Y Shapiro's Method. *Journal of Logic Programming*, 1987:177–198, 1987.

[FSW96] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Budapest, August 1996.

[GC89] Ashok Goel and B. Chandrasekaran. Functional representation of designs and redesign problem solving. In *Proc. IJCAI*, pages 1388–1394, Detroit, August 1989.

[GR83] A. Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison-Wesley, May 1983.

[GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[HM93] Jonathan E. Hazan and Richard G. Morgan. The Location of Errors in Functional Programs. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging, AADEBUG*, pages 134–170, Linköping, Sweden, May 1993. Springer-Verlag.

[Joh86] W. Lewis Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Pitman Publishing, 1986.

[Liv94] Beat Liver. Modeling software systems for diagnosis. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis*, pages 179–184, New Paltz, NY, October 1994.

[Llo87] J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5:133–154, 1987.

[Mur88] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Pitman Publishing, 1988.

[NF93] Henrik Nilsson and Peter Fritzson. Lazy Algorithmic Debugging: Ideas for Practical Implementation. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging, AADEBUG*, pages 117–134, Linköping, Sweden, May 1993. Springer-Verlag.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Rie93] Marc Riese. *Model-based diagnosis of communication protocols.* PhD thesis, EPFL Lausanne, 1993.

[Sha83] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[SSGP92] E. Stroulia, M. Shankar, A. Goel, and L. Penberthy. A model-based approach to blame-assignment in design. In *Proceedings Artificial Intelligence in Design*, pages 519–537, Pittsburgh, 1992.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.

[Str87] Bjarne Stroustrup. What is "object-oriented programming"? In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, LNCS 276, pages 51–70, Paris, France, June 15-17 1987. Springer Verlag.

[SW96a] Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In *Proc. DX'96 Workshop*, Val Morin, Canada, 1996.

[SW96b] Markus Stumptner and Franz Wotawa. Model-based diagnosis of hardware description languages. In *Proc. CESA'96 IMACS Multiconf.*, Lille, 1996.

[SW96c] Markus Stumptner and Franz Wotawa. Model-based program debugging and repair. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Fukuoka, 1996.

[SW96d] Markus Stumptner and Franz Wotawa. A model-based tool for finding faults in hardware designs. In *Proceedings Artificial Intelligence in Design*, Stanford, 1996.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, pages 168–182, December 1987. Published as Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, number 12.

[Wot96] Franz Wotawa. *Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages.* PhD thesis, Technische Universität Wien, 1996.

4 Place of Research and Financial Planning

4.1 Place of Research

The project will be carried out at the Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme, Technische Universität Wien (Information Systems Department, Group for Databases and Expert Systems) headed by *Prof. Dr. Georg Gottlob*). The department was founded in 1984 and currently employs 2 professors, 7.5 assistant professors, 1 hardware technician, 1 system administrator, and 2 secretaries, as well as a varying number of researchers working on industry-funded projects.

The group has been involved with model-based diagnosis research continuously since 1986. The department's infrastructure (network, file and mail servers, printers, library) can be used for the project, and the administrative staff of the department (secretaries, system administrator etc.) will be available for support. Rooms for the new project members will also be available.

4.2 Staff Resources

This section lists all anticipated members of the JADE Project.

4.2.1 Existing Staff

Two assistant professors will be members of Project JADE . Both are members of the Institut für Informationssysteme.

Dr. Markus Stumptner (project leader): Dr. Stumptner will be primarily concerned with organizing and coordinating the project, including the integration of the results from different tasks, and in guiding the more theoretical parts of the project. He will be involved with the theoretical investigation of MBSD (**T1**), as well as the application of MBSD to Java (**T2**), and the evaluation of alternate models (**T3**). He will also support students working on their master's thesis (diploma) within the project.

Dr. Franz Wotawa: Dr. Wotawa will likewise be strongly involved in the theoretical aspects of the project, tasks **T1**, **T2**, and **T3**. He will be strongly involved in exercising control over the implementation-oriented tasks – the IDEA implementation (**I1**) and the development of the example programs (**I2**). He will also support the work on master's theses within the project.

4.2.2 Requested Staff

Two research assistants and three students working on their master's thesis are required.

Since one can expect that the thesis work will be more difficult compared to average theses, in all cases the maximum possible financial support should be provided (AS 60.000,–).

The additional motivation may be helpful, since the theses will take longer to finish than usual ones and the students can earn much more money in industry.

RA1: RA1's task will be T1, i.e., investigating the theoretical aspects of MBSD with regard to modelling of programs, and T3, i.e., evaluation of and experimentation with alternate models for diagnosis (together with Dr. Stumptner and Dr. Wotawa).

RA2: RA2's task will be T2, i.e., applying MBSD principles to Java, and I1, the actual development of the IDEA system (together with Dr. Stumptner, but mainly Dr. Wotawa).

D1: D1's task is to contribute to task I1. The diploma will be concerned with the implementation of inference algorithms.

D2: D2's task is to contribute to task I1. The diploma will be concerned with the implementation of the IDEA user interface.

D3: D3's task will be task I2. The diploma will be concerned with the development and classification of example Java programs

4.3 Existing Equipment

The Information Systems department owns several workstations and PCs, which are available for use by the permanent staff members, and for student projects on a shared basis. Additional equipment for the full-time research assistants funded by the project will be required, however.

4.4 Requested Equipment

Two PCs should be paid by project funds. The operating system *Linux* as well as other required software will be installed on each of them. The two PCs are considered as workstations for the two research assistants. We estimate AS 90.000,– as total cost for the two PCs.

4.5 Support for Business Trips

Since financial support for business trips to conferences cannot be provided by the department, at least 3 business trips per year should be funded to present the results of the project to the scientific community (two to Europe, one to overseas). We assume a sum of AS 10.000,- for European trips and AS 20.000,- for overseas trips.

Conference fees can be funded by the Information Systems department.

4.6 Support for Publications

Costs arising from publications can be funded by the Information Systems department.

5 Additional Details

This project is not funded and will not be funded by other institutions.

6 Tables

Temporal Evolution of Costs

Type of Costs	1. Year	2. Year	Sum
Staff	776.000,–	716.000,–	1.492.000,–
Equipment	124.992,–	0,–	124.992,–
Material	0,–	0,–	0,–
Business Trips	40.000,–	40.000,–	80.000,–
Other	0,–	0,–	0,–
Sum	940.992,–	756.000,–	1.696.992,–

Temporal Evolution of Staff Costs

Staff	1. Year	2. Year	Sum
RA1	328.000,–	328.000,–	656.000,–
RA2	328.000,–	328.000,–	656.000,–
D1	30.000,–	30.000,–	60.000,–
D2	30.000,–	30.000,–	60.000,–
D3	60.000,–	0,–	60.000,–
Sum	776.000,–	716.000,–	1.492.000,–

No national or international cooperations within the JADE project are planned. No work on demand of third parties will be done within JADE
.