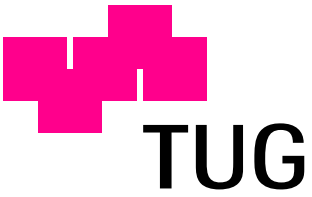
Magisterarbeit

Strategies to Automatically Test Eiffel Programs

Andreas Leitner

Institut für Softwaretechnologie
Technische Universität Graz
Vorstand: Vertragsprof. Dipl.-Ing. Dr.techn. Franz Wotawa



Begutachter: Vertragsprof. Dipl.-Ing. Dr.techn. Franz Wotawa
Betreuer: Roderick Bloem, PhD

Graz, im Dezember 2004

Ich versichere, diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubter Hilfsmittel bedient zu haben.

Abstract

Semi-automated software testing has become an increasingly popular and effective tool for raising the quality of software. Pre-, postconditions, and invariants describe the semantics of a feature (i.e., method or data member) and can help to decide whether a given test failed or passed [ARM03, Gre04, Ciu04]. Since the decision can be made without further knowledge of the input this opens up the possibility of a fully automated *push button tester*. Implementing such a system reveals both theoretical and practical challenges.

A fully automated tester is likely to run the system under test into unrecoverable states and must at least be able to recover gracefully in order to continue testing. A master/slave mechanism has been implemented to automatically resume testing after a runtime crash.

The tester must be able to satisfy the preconditions of the features under test in order to call them. Automatically satisfying preconditions in Eiffel is a challenging problem, since the contract is in practice never complete. Even if it is, its expression language is both non-functional and undecidable. Using the very simple approach of (almost) random input data yields promising results, but features with strong preconditions are left untested. A more guided approach based on a planning system has been implemented for those cases. From Eiffel source code a planning problem is generated with the goal being the precondition of the feature under test. A planner is used to generate a plan, which is then executed by an Eiffel interpreter. If the goal can be reached, the precondition of the feature under test is satisfied and thus the feature can now be tested, otherwise information from the plan execution is used to augment the planning problem and the planner is asked for a new plan.

Zusammenfassung

Semiautomatisches Testen von Software erfreut sich zunehmender Popularität. Vorbedingungen, Nachbedingungen und Invarianten beschreiben die Semantik eines Eiffel-Features, auch method und data-member genannt, und können daher verwendet werden, um zu entscheiden, ob ein Test versagt oder bestanden hat. Diese Entscheidung kann ohne weitere Kenntnis der Eingabedaten geschehen. Daher kann der Testvorgang vollständig automatisiert werden. Die Entwicklung eines solchen Testsystems erfordert die Überwindung von sowohl theoretischen, als auch praktischen Problemen.

In der Praxis verursacht ein automatisierter Tester, durch die nahezu unkontrollierte Ausführung des zu testenden Systems, oft Abstürze die den Tester nicht vom weiteren Testen abhalten dürfen. Ein Master/Slave Mechanismus wurde entwickelt der den Testvorgang robust gegenüber Laufzeitabstürzen macht.

Um ein Feature zu testen muss zuerst seine Vorbedingung erfüllt werden. Das automatisierte Erfüllen von Vorbedingungen ist ein anspruchsvolles Problem, da in der Praxis die Spezifikation nicht vollständig ist. Die Verwendung von zufälligen Eingabedaten liefert jedoch erstaunlich gute Resultate. Features mit starken Vorbedingungen können in der Regel mit dieser Strategie nicht getestet werden. Für diese Fälle wurde eine komplexere Strategie, basierend auf einem Planungssystem, implementiert. Ausgehend vom Eiffel Quelltext wird ein Planungsproblem generiert, dessen Ziel die Erfüllung der Vorbedingung des zu testenden Features ist. Ein Planungssytem generiert dann einen Plan, der auf einem Eiffel Interpreter ausgeführt wird. Kann das Ziel erreicht werden, ist die Vorbedingung des zu testenden Features erfüllt und das Feature kann getestet werden. Andernfalls wird das Planungsproblem mit Informationen aus der Planausführung erweitert und ein neuer Plan wird erstellt.

Acknowledgements

Diese Magisterarbeit wurde im Jahr 2004 am Institut für Softwaretechnologie an der Technischen Universität Graz, in Zusammenarbeit mit dem Chair of Software Engineering der ETH Zurich, durchgeführt.

Zu allererst möchte ich mich bei meinem Betreuer Dr. Roderick Bloem bedanken, der mir in nicht endenwollender Geduld mit Rat und Tat zur Seite stand. Bei Prof. Dr. Bertrand Meyer und Prof. Dr. Franz Wotawa möchte ich mich für das Zustandekommen dieser Magisterarbeit bedanken.

Für zahlreiche Diskussionen und Vorschläge möchte ich mich bedanken bei: Dipl.-Ing. Barbara Jobstmann, Gunther Laure, Wolfgang Lazian und Prof. Dr. Wolfgang Slany von der TU Graz, bei Dr. Karine Arnout, Dipl.-Ing. Till Bay, Ilinca Ciupa und Vijay D'silva von der ETH Zuerich und bei Xavier Rousselot.

Natürlich möchte ich mich bei meiner Freundin Dott.ssa Ilaria Galli bedanken, die mir in den letzten Monaten, nicht nur mit Ihren hervoragenden Englischkenntnissen beigestanden ist. Auch möechte ich mich bei meiner Familie und meinen Freunden bedanken.

Graz, im Dezember 2004 Andreas Leitner

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter presents an overview of the topics covered in this thesis. We explain the concept of contract based testing. TestStudio, an application developed at the ETH Zurich that implements contract based testing, is described. Finally extensions to TestStudio, which have been implemented as part of this thesis, are covered.

With the advent of frameworks such as JUnit [HT03] software testing has become increasingly popular. They automate the process of building and executing test cases. Software equipped with preconditions, postconditions, and invariants can in many cases be used to go one step further: test cases can be automatically derived from the contracts, eliminating the last manual step and making testing fully automatic [Aic01]. Automated test case generation supplements unit testing rather than substituting it. Counter examples found during automated testing can be added to the existing suite of unit tests.

The Eiffel language has built-in support for preconditions, postconditions and invariants, called *Design by Contract*™. When a client calls a feature (the supplier) it must establish the precondition of the supplier. The supplier then has the duty to establish its own postcondition. The invariant needs to be established by the creation procedure and must then hold at the beginning and the end of (almost) every feature call. In the case of an abnormal feature execution the guilty party can be determined:

1. If the precondition is violated the client is to blame.
2. If the class invariant is violated before the execution of the routine body the client is to blame.
3. If the postcondition is violated the supplier is to blame.
4. If the invariant is violated after the execution of the routine body the supplier is to blame.

Steps 3 and can serve as a testing oracle, i.e., determine whether a feature test passed or failed.

1.1 TestStudio

TestStudio is a contract based testing tool developed by Greber [Gre04] and Ciupa [Ciu04] at the ETH Zurich after the proposal [ARM03]. TestStudio implements the idea of contract

based testing. The concepts discussed in this thesis have been implemented as extensions to TestStudio.

TestStudio, as implemented by Greber and Ciupa, is very fragile. Chapter 4 presents the results of testing fundamental data structure classes. Even with such closed world classes, TestStudio will abort execution before gathering any test results at all. This is because every class (automatically) depends on the kernel library, which uses the *external "C"* mechanism.

Eiffel features a foreign language interface which, for example, allows C functions to be called from Eiffel. C functions in general do not validate their arguments and thus may harm the systems state when called with invalid arguments. In some situations they can even render the Eiffel runtime unusable, in which case the only option is the termination of the system's process. Furthermore, the damage may not be immediately detectable and only trigger a fatal error in subsequent feature calls.

Features that destroy the invariant of one or several objects are another reason to quit the testing process. Objects violating their invariant must not be used for further testing, because they can confuse the oracle. Such objects may be used directly or indirectly in subsequent test cases and trigger false positives. In general it is not possible to detect all objects whose invariant is violated, nor can such objects be removed.

Routines such as *EXCEPTIONS.die*, whose purpose is to terminate the current process, obviously also terminate testing and need to be dealt with.

1.2 Master/Slave Design

Testing classes from the *EiffelBase* library revealed that the problems mentioned above can render a one process based tester dead before it produces any output at all. With a one process based tester, the execution of a whole test suite (i.e.: a sequence of individual test cases) is controlled from within the same process that executes the test cases. If for some reason a test case renders the runtime of this process unusable, test cases scheduled after the harmful test case will not be executed. Before the first feature can be tested, objects to serve as arguments need to be created. Termination of the process in this stage will yield not a single test result. To overcome this problem TestStudio has been modified so that testing is split into two processes: a *master* and a *slave*. The goal of the modification is to output results for as many test cases as possible, even in the presence of test cases that harm the runtime. The master process knows what features to test and communicates with the slave process through the standard input/output streams. The slave is a minimal Eiffel interpreter which is able to create objects, execute features and report back to the master.

The reflection facilities from ISE Eiffel are not complete enough to allow building an Eiffel interpreter. At the time of the writing, they only provide value retrieval and assignment for attributes. Routine invocation specifically is not possible. Due to this limitation we implemented a reflection library generator. The Gobo [Bez03] Eiffel parser is used to parse the system under test, then an ERL [Flu04] implementation is generated from the parsed AST.

The slave, mainly consists of the generated reflection library. It is a simple interpreter that listens on the standard input for requests, executes those requests and prints infor-

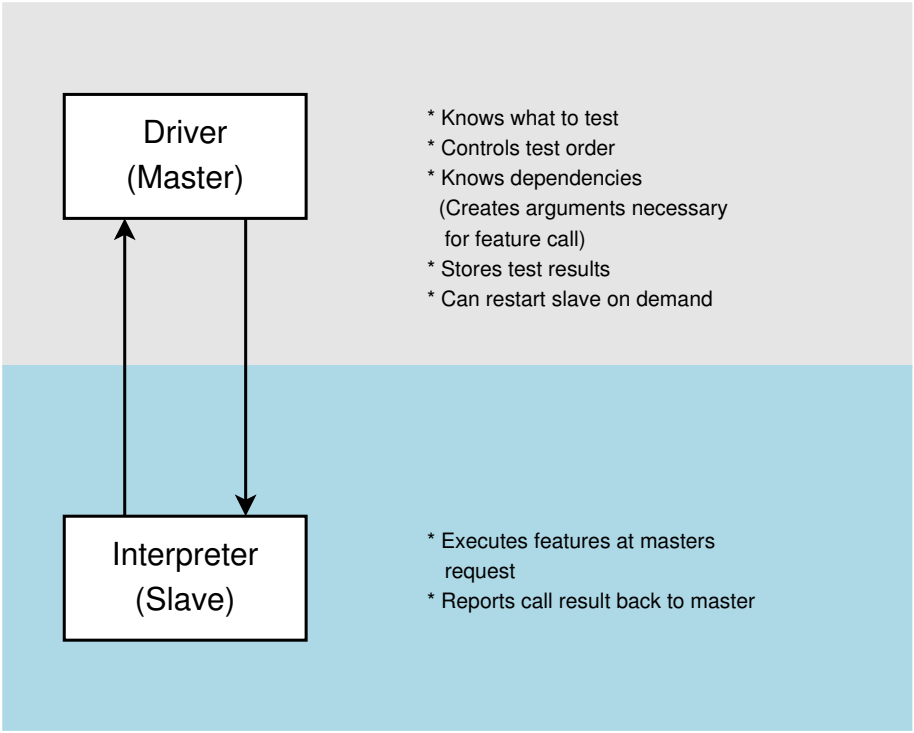


Figure 1.1: Master Slave Concept

mation about the execution to the standard output stream. Exceptions are caught and also reported to the standard output stream.

If a fatal error is detected, the slave terminates. The master starts a new client and continues testing the feature under test. If a feature under test causes a fatal error repeatedly, the master moves on to the next feature under test. The test result information is stored by the master so that a restart of the client does not cause any loss of information.

In order to test a feature, its precondition must be satisfied. In general it is not possible to automatically construct objects which deterministically satisfy a given precondition because of the following two issues:

- In practice, the assertions provided by the programmer rarely form a complete specification. Most source code is written with postconditions that only state changes, leaving the frame problem open [1].
- Even though it is encouraged to write side effect-free functions, functions are not checked for this property by the compiler.

Nevertheless strategies can be applied to solve the problem of satisfying preconditions for many situations. This thesis discusses two strategies in detail: a random and a guided approach. A third one using genetic programming has been implemented as a proof of

[1] [BMR95] defines the frame problem as *"the inability to express that a procedure changes only those things it has to, leaving everything else unmodified"*

concept. The random approach, first implemented in the one process based tester by Nicole Greber [Gre04], has been re-implemented in the master/slave tester. The guided approach uses a planner.

1.3 Random Strategy

The idea of the random approach is to call the features under test with random arguments. The following steps are taken to test a given feature *f* of type *T*:

1. Create an object of type *T*. A random creation procedure of *T* is selected. Store created object in the object repository.
2. Create objects suitable as actual arguments for *f*. Again a random creation procedure is selected. Store created objects in the object repository.
3. Select a random object from the repository and invoke a random feature on it (this step is justified below).
4. Retrieve conforming objects from the object repository for the target object and all actual arguments. If several objects conform for a given argument or the target object, select one randomly.
5. Invoke feature with selected objects.

Several things are worth noting here:

- Steps 1, 2 and 3 are recursive and terminate only stochastically. To invoke a regular feature or creation procedure new arguments may be needed.
- Step 3 may seem superfluous, but it serves the purpose to modify the objects in the repository. This is useful if a given feature is tested repeatedly. Due to step 2 chances are that the feature will be called from different states, thus potentially changing the code path taken by the feature and in turn increasing test coverage.

A side effect of calling a certain feature under test is that other features (and creation procedures) might be called. Those might not even be selected for testing by the user. But since they are executed potential bugs might be discovered.

The above strategy is rather unguided due to the heavy use of randomness. Others, for example [MK01], have used more complex strategies. Nevertheless this strategy detected several bugs in the production quality libraries *EiffelBase* and the *Gobo Eiffel Structure Library*. Detecting bugs being the primary goal of a tester this is clear evidence of the strategy's effectiveness. Detailed results are presented in Chapter 4.

One weakness of the random strategy is that it leaves features with complex preconditions untested since all modifications made to the object state are due to random feature invocations and not influenced by the precondition of the feature under test. Thus, the more complex/stronger the precondition the less likely it gets satisfied.

1.4 Planning Strategy

A more guided strategy is needed to satisfy complex preconditions. We developed a planner based strategy, whose idea is to use the information given by the contracts. As noted above the contracts are rarely complete, but they can still serve as an indication of what features do.

The actual execution and evaluation of the feature under test is taken from the first strategy; only the creation and modification of the objects needed to execute the call to the feature under test is different. Figure 1.2 shows the concept of this strategy.

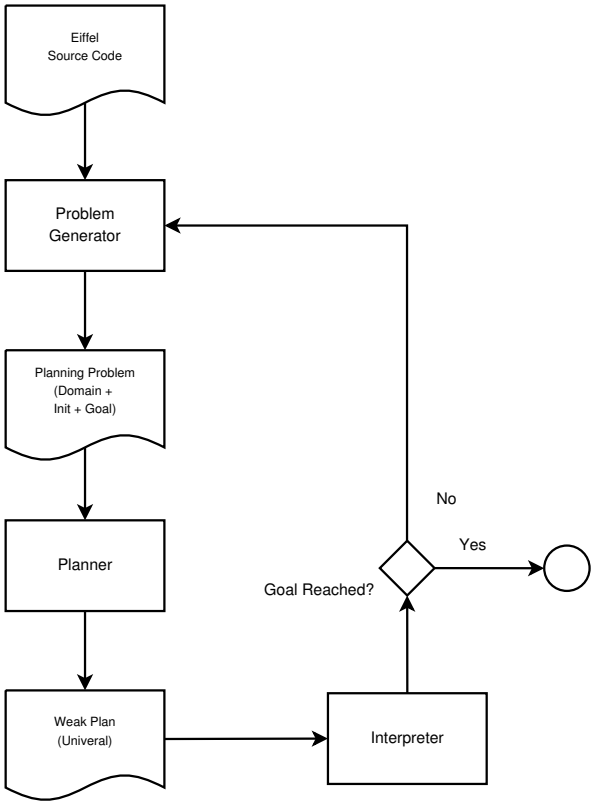


Figure 1.2: Planning Flowchart

For example the feature *LINKED_LIST.remove*, which removes the item under the current cursor position, requires the list to be *not off* (i.e.: the cursor must be positioned on a element and not *before* or *after*). For the cursor to be on an element there must at least be one element in the list so a possible plan for testing *LINKED_LIST.remove* would be to:

1. Create an object of type *LINKED_LIST*.
2. Insert the *Void* reference into the list.
3. Move the cursor to the position of the just inserted element.

4. Call *LINKED_LIST.remove*.

The planner based strategy abstracts the Eiffel source code to a planning problem. Attributes are mapped to state variables, routines to actions. The goal state is the abstracted precondition of the feature under test. The initial state is a state where no object has yet been created. Actions have a notion of precondition and effect, which are mapped from the Eiffel routines precondition and postcondition. The routine body is not taken into account at all.

We use the planning tool bifrost, described in [Jen03], to extract a universal plan from the problem. A universal plan is a state-action table that associates a set of actions to a state that can be executed in such state [CPRT03]. Due to the uncertainty of the planning problem (which is in turn caused by non-complete contracts), in general, no *strong solution* can be extracted. A strong solution is defined by [CPRT03] as follows:

> Strong solutions are plans that are guaranteed to achieve the goal in spite of nondeterminism: all the sequences of states corresponding to its execution reach the goal.

A weak solution is defined by [CPRT03] as:

> Weak solutions are plans that may achieve the goal, but are not guaranteed to do so: at least one of the many possible sequences of states corresponding to plan execution reaches the goal (i.e., the final state is a goal state).

A weak solution can in practice always be found. The output of the planner, the weak solution, is parsed by the tester and then used to drive the interpreter. Due to the nature of the weak solution, we need the interpreter to decide if a given plan really leads to the goal state (the satisfied precondition) or not. If we can reach the goal state the feature under test can now be tested. If the goal state has not been reached, the original planning problem is augmented with information gained during interpretation. For every feature invocation the object state before the feature invocation ($\text{state}_{\text{old}}$) and the state after the feature invocation ($\text{state}_{\text{new}}$) are recorded. The action corresponding to the feature called has its postcondition augmented with the expression $\text{state}_{\text{old}} \rightarrow \text{state}_{\text{new}}$. The augmented plan now has stronger postconditions and thus reduced uncertainty, which will in many cases lead to a better weak solution. This assumes that Eiffel objects will act deterministically, which is not true in general. Nevertheless many real life objects will indeed act deterministically, because code that acts deterministically is much easier to understand.

Genetic Programming Strategy A third strategy has only been implemented as a proof of concept. The idea is to use genetic programming to breed test cases. The concept and the experimental results are presented in Chapter 3.

Chapter 2

Preliminaries

The content of this chapter is preliminary for the rest of this thesis. We give an overview of the implementation and target language of the tester described in this thesis. Then the chapter explains the concept of software testing in general. Finally, planning, which is the basis for the strategy described in Chapter 5 is introduced. This chapter may be skipped by readers already familiar with the topics mentioned above.

2.1 Eiffel

The automatic tester is implemented in Eiffel. The language targeted by the tester is also Eiffel. For this reason, the following gives a brief overview of the language. Eiffel is a pure object oriented programing language. The main features of the language are:

- Static type system
- Multiple inheritance
- Constrained genericity
- Design By Contract

Listing 2.1 shows the Eiffel version of the popular *Hello World* example. The only class in this example is *HELLO_WORLD*. This class has only one feature: the procedure *make*. The implementation of this procedure prints the string *Hello world!* followed by a new line character to the standard output stream. In Eiffel every class implicitly or explicitly inherits from the class *ANY*. The routine *print* is defined in this class for convenience reasons. The procedure *make* is also marked to be a creation procedure. Since it has no arguments it can serve as the programs entry point.

2.1.1 Eiffel Terminology

Eiffel has its own naming convention which often diverges from the conventions used in languages such as C++, Java or C#. Throughout this thesis the Eiffel naming convention is used. Loryn Jenkins wrote a terminology mapping from Eiffel to C++ [Jen04]. Table 2.1 is based on the mapping from Jenkins and shows the entries relevant for this thesis.

Listing 2.1: Hello World in Eiffel

```
1 class HELLO_WORLD
  create
    make
4 feature
    make is
      do
7       print ("Hello World!%N")
      end
  end
```

2.1.2 Design By Contract

The perhaps most distinguished feature of Eiffel is its support for Design By Contract. Design By Contract is a a methodology for designing computer software. It provides benefits in the following processes:

Design – Interfaces equipped with contracts ensure that their purpose is stated clearly.

Documentation – Contracts make source code easier to understand. External documentation tends to get out of sync with the actual implementation. Contracts are part of the source code and thus are less likely to go out of sync.

Debugging – Enabling assertion checking during runtime reveals bugs earlier and closer to the source of the bug.

Design By Contract is derived from the *correctness formulas* or *Hoare triples* used in formal program verification [Mey97]:

> Let A be some operation (for example an instruction or a routine body). A *correctness formula* is an expression of the form:
>
> $\{P\}A\{Q\}$
>
> denoting the following property, which may or may not hold:
>
> **Meaning of a correctness formula** $\{P\}A\{Q\}$
>
> *"Any execution of A, starting in a state where P holds, will terminate in a state where Q holds."*

In Eiffel correctness statements are supported via the following constructs:

Precondition – Boolean expression which must hold before every feature call.

Postcondition – Boolean expression which must hold after every feature call.

Class Invariant – Boolean expression which must hold before and after every feature call.

Check Instruction – Boolean expression which must hold at a certain position in a statement block. The *check* instruction is similar to the *assert* instruction from Java.

Eiffel	C++
ace file	Makefile
	An ace file is descriptive, while a makefile is imperative
ancestor class	x1 superclass
attribute	data member
CAT	Change of availability or type
	For CAT calls see Section 2.1.3
check	assert
child class	derived class
cluster	namespace
	Clusters group classes.
	Two classes from two different cluster must not have the same name
command	virtual function
	with void return type
create x.make	x = new X
creation feature	constructor
deferred feature	pure virtual function
deferred class	abstract base class
descendent class	subclass
feature	function
	function = method
feature	data member
feature	method
	method = function
fully deferred class	pure virtual base class
function	virtual function
	with non-void return type
generic class	class template
manifest object	literal constant
parent class	base class
Precursor	super()
	excepting that super() is regularly used
	to call any feature of the superclass so desired
Result	return

Table 2.1: Eiffel to C++ terminology mapping

Loop Invariant – Boolean expression which must hold at the beginning and the end of each loop iteration

Loop variant – Integer expression which decreases with every loop iteration.

Following defines the semantic for the most important clauses (i.e.: precondition clause, postcondition clause, and class-invariant clause) [Mey97]:

Let C be a class, INV its class invariant. For any routine r of the class,

call $\text{pre}_r(x_r)$ and $\text{post}_r(x_r)$ its precondition and postcondition; x_r denotes the possible arguments of r, to which both the precondition and the postcondition may refer. (If the precondition or postcondition is missing from the routine text, then pre_r or post_r is just True.) Call Body_r the body of routine r. Finally, let Default_C be the assertion expressing that the attributes of C have the default values of their types.

Definition: class correctness :

A class is correct with respect to its assertions if and only if:

- For any valid set of arguments x_p to a creation procedure p: $\{\text{Default}_C$ **and** $pre_p(x_p)\}$ Body_p $\{\text{post}_p(x_p)$ **and** INV$\}$
- For every exported routine r and any set of valid arguments x_r: $\{\text{pre}_r(x_r)$ **and** INV$\}$ Body_r $\{\text{post}_r(x_r)$ **and** INV$\}$

Runtime assertion-checking can be enabled via the ace file. The *rescue* clause can catch exceptions similar to the C++/Java/C# *catch* clauses.

2.1.3 CAT Calls

Eiffel allows for covariant redefinitions in descendants. A covariant redefinition, is a redefinition which makes a class more restrictive than its parent. An often used example which contains a covariant redefinition is shown in Listing 2.2.

Listing 2.2: Example of covariant redefinition

```
  class FOOD
  end
3
  class GRASS
  inherit FOOD
6 end

  class ANIMAL
9 feature
    eat (f: FOOD) is
      do
12      -- ...
      end
  end
15
  class COW
  inherit ANIMAL
18  redefine eat end
  feature
    eat (f: GRASS) is
21    do
        -- ...
      end
24 end
```

These redefinitions are source for an error which is hard to detect at compile time: the *CAT call* (*CAT* stands for *Changing Availability or Type*). For example the call *a.eat (f)* from Listing 2.3 is a CAT call. Per language definition CAT calls are invalid. Unfortunately checking for CAT calls statically is undecidable. Worse, most compilers do not check for CAT calls at runtime. There, a CAT call results in undefined behavior, which in practice often means that the runtime panics and terminates. Gelint, a tool from the GOBO suite [Bez03], implements the dynamic type set mechanism as described in

[Mey92]. This tool checks for CAT calls statically. It is worth noting that this mechanism is pessimistic (it reports false negatives).

Listing 2.3: Example of CAT call

```
local
  a: ANIMAL
  c: COW
  f: FOOD
do
  create c
  create FOOD
  a := c
  a.eat (f)
end
```

2.1.4 *LINKED_LIST*

Most examples presented throughout this thesis will be related to the class *LINKED_LIST* from the EiffelBase library [Mey94]. As the name of the class suggests it is a class representing lists. It is implemented via one way linked cells. The list is navigated via a cursor, which can be moved:

- before the beginning of the list
- to the first item of the list
- to the last item of the list
- after the end of the list
- forth one item
- back one item

Figure 2.1 shows a list with 6 items and the 4 fundamental cursor positions. Note that even an empty list has the concept of the before and after position. The interface of the class is shown in Section A.3. The features relevant for this thesis are:

make – Create a new list.

first – The first item in the list.

item – The item at the current cursor position.

last – The last item in the list.

count – The number of items in the list.

after – Is the cursor positioned after the end of the list?

before – Is the cursor positioned before the beginning of the list?

is_empty – Is the list empty?

isfirst – Is the current cursor positioned on the first element of the list?

islast – Is the current cursor positioned on the last element of the list?

off – Is the cursor not on a valid position? (i.e.: Is the cursor *before* or *after*?)

back – Move the cursor one position back.

finish – Move the cursor to the last element in the list.

forth – Move the cursor one position forth.

start – Move the cursor to the first item in the list.

force – Add an item to the end of the list.

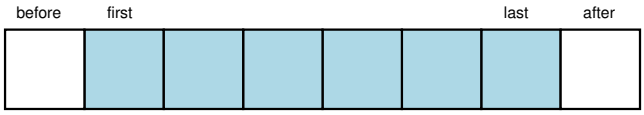


Figure 2.1: Cursor positions of *LINKED_LIST*

2.2 Testing

There are several kinds of testing. Throughout this thesis software testing means checking whether a given implementation conforms to its specification. In order to completely assure the quality of a software system, it needs to be tested against its complete input space. In practice even for small systems, the input space is too large for complete testing. This well known fact has been summarized by Dijksta in 1972 as "Program testing can be used to show the presence of defects, but never their absence!" [DDH74]. Therefore it is of great importance to select representative inputs for testing. The popular Extreme Programming approach [Bec00], puts testing in a central position. With Extreme Programming, before implementing the routines of a class, automatic test procedures must be written. In traditional software process models testing is often done as one of the last steps and then left out completely because of exceeded resources. By writing the test cases before the actual implementation and automating the execution of the test cases, Extreme Programming promises to avoid this problem. Since the test cases are written before the implementation, they additionally serve as a(n) (incomplete) specification. In [Aic01], Aichernig formalizes this relation and shows not only that test cases are formal specification but also that they can be derived from traditional specification.

2.3 Planning

The testing strategy described in Chapter 5 is based on a planner called Bifrost [Jen03]. For this reason the following gives a brief overview of planners in general, and Bifrost in particular.

To solve a planning problem a sequence of actions must be found. The actions are required to change the state of a given domain from an initial state to a goal state. A typical example for a planning is the *Blocks World*. The blocks world consists of a flat surface and several blocks. A robotic arm can pick up, lift and stack one block on top of another. The robots task then is to place certain blocks on other blocks. A planning problem consists of a domain theory, one or more initial states, and one or more goal states. The STRIPS language [FN71] is one of the best known languages to represent a domain theory. Classical planners can be categorized into:

State Space Planners search in the state space of the domain.

Plan Space Planers search through the plan space.

Recently research has gone into using new encodings for the planning problem algorithms. The *Model Based Planner* (MBP) [CGGT97] first encoded a planning problem as a non deterministic finite automaton (NFA). The NFA itself is represented via Binary Decision Diagrams (BDD) [Bry86]. Generating universal plans, this technique can be successfully used in non deterministic domains. The OBDD represents the transitions of the NFA. The algorithms used to verify CTL properties in model checking [CGP99] can be used to extract universal plans from the planning problem. A universal plan is a state-action table describing in which state what actions are applicable to reach the goal. In [CPRT03], three kinds of solutions are distinguished:

Weak solutions are plans that may achieve the goal, but are not guaranteed to do so: at least one of the many possible sequences of states corresponding to plan execution reaches the goal (i.e., the final state is a goal state).

Strong solutions are plans that are guaranteed to achieve the goal in spite of nondeterminism: all the sequences of states corresponding to its execution reach the goal.

Strong cyclic solutions are plans that are guaranteed to achieve the goal under a "fairness" assumption. Their execution can result in an infinite sequence of states, i.e., execution can loop forever. However, this happens only if some action is executed infinitely often in a given state and some of its outcomes (the ones leading to the goal) never occur. We say that these executions are "unfair".

2.3.1 Bifrost

The Bifrost planner, like MBP, is based on BDDs. It uses the extNADL language to describe planning problems. For this thesis, we choose Bifrost because both the precondition and effect clause support arbitrary expressions. This makes the conversion from Eiffel source easier. (As will be explained in Chapter 5, a converter from Eiffel to extNADL was implemented as part of this thesis.) The planner supports the strong, strong-cyclic, and weak solution extraction algorithms.

extNADL

The Bifrost planner uses the extNADL problem language. In Chapter 5 several planning problems formalized in extNADL will be studied. For this reason the following briefly describes this language. An extNADL problem consists of four parts:

1. **Variables**: The variables span up the state space of the NFA. A variable is declared by providing a type and name. Possible types are *bool*, which represents boolean values and *nat (x)*, which represents positive numbers with a precision of x bits.
2. **Actions**: The actions represent the transitions between the states. An action consists of a name, a modifies clause, a precondition clause, and an effect clause. The modifies clause specifies which variables can be changed by the action. The precondition clause specifies the set of states from which the action can be performed. The effect clauses states how the action can change the state.
3. **Initial States**: The initial states specify the valid starting states.
4. **Goal States**: The goal states specify the set of states that are valid end states.

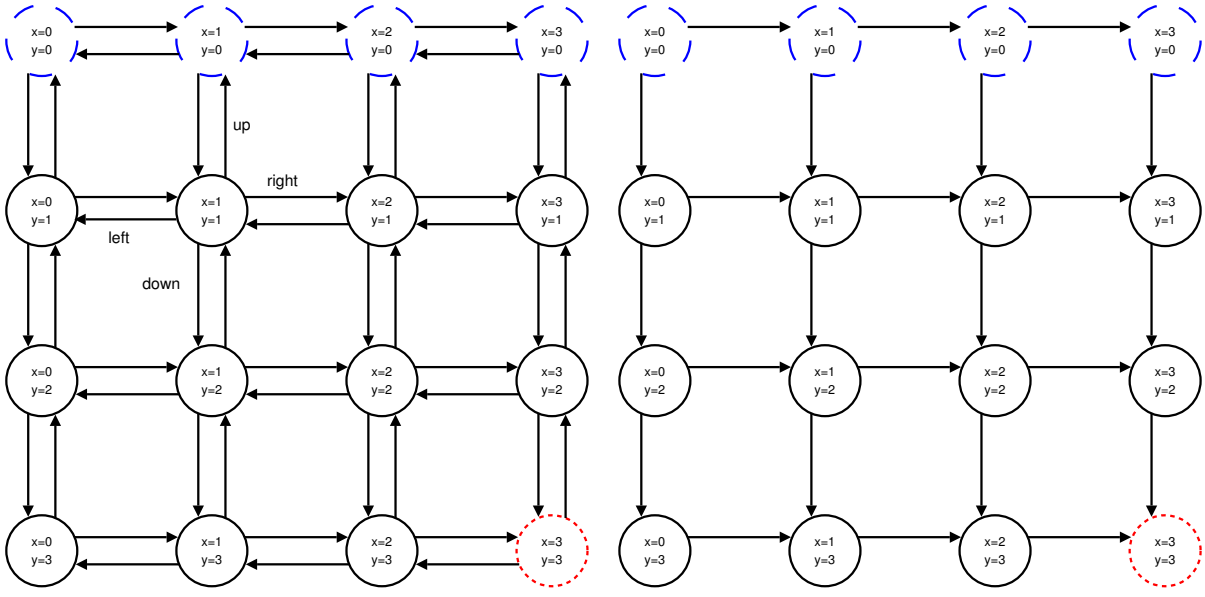
Note that extNADL also supports multiple agents, which can act concurrently. This feature is not explained because it is not used in this thesis. Listing 2.4 shows a simple planning problem. The two variables (x and y) represent the current position of the robot on a 4 by 4 grid. The robot has four actions (*move_up*, *move_down*, *move_left*, and *move_right*). Note that in the effect clause, a primed variable ("′") refers to the value of the variable after the execution of the action, while a variable not followed by a prime refers to the value of the variable before the execution of the action. The start states are all states in which $x = 0$ holds. The goal state is the state where $x = 3 \wedge y = 3$ holds. Figure 2.2(a) shows the state machine representing this problem. The extracted strong solution is shown in Table 2.2. Note that a solution viewed as a state machine (see Figure 2.2(b)) is a subset of the problem state machine.

x	y	
⋆⋆	0⋆	move_down
⋆⋆	10	move_down
0⋆	⋆⋆	move_right
10	⋆⋆	move_right

Table 2.2: Strong solution for simple extNADL example

Listing 2.4: Simple extNADL planning problem

```
VARIABLES

    nat(2) x
    nat(2) y

SYSTEM

  agt: robot

  move_up
    mod: y
    pre: y > 0
    eff: y' = y - 1

  move_down
    mod: y
    pre: y < 4
    eff: y' = y + 1

  move_left
    mod: x
    pre: x > 0
    eff: x' = x - 1

  move_right
    mod: x
    pre: x < 4
    eff: x' = x + 1

ENVIRONMENT

INITIALLY
  x = 0

GOAL
  x = 3 /\ y = 3
```



(a) Planning Problem (b) Strong Solution

Figure 2.2: Planning problem and solution as state machines

Chapter 3

Related Work and Alternative Approaches

This chapter briefly describes two related research projects: TestEra [MK01] and Korat [BKM02]. Both try to automate test case generation and target the language Java. An alternative strategy to test Eiffel programs based on genetic programming is introduced. While we implemented a prototype tester based on genetic programming it was not the main focus of this thesis.

3.1 Related Work

3.1.1 TestEra

TestEra [MK01], is a tool for test case generation. TestEra targets software written in Java. Specification written in Alloy [Jac02] needs to be provided for the software under test. Up to a maximum number, all non-isomorphic instances that conform to the specification are built. The instances are translated to Java and the software is tested. The output produced by the tests is mapped back to Alloy and is checked against a correctness criteria also given in Alloy.

By systematically constructing input instances, TestEra is able to get good data-coverage in closed world systems. Open world systems, however are potentially problematic. A predicate, which can be made true not by construction but by execution of an external method cannot be dealt with automatically. For example, a class representing a video surface is likely to have a property that decides whether the surface has been initialized or not. Drawing routines then, will likely require an initialized surface. To initialize a surface and make the property in question true an external routine needs to be called. The automatic abstraction and concretization translations that convert between Alloy and Java cannot generate test cases for such domains.

3.1.2 Korat

Korat [BKM02] too automates test case generation. Similarly to TestEra, all non-isomorphic inputs conforming to the precondition and invariant of the method under test are generated and used for testing. Unlike TestEra, Korat lets the programmer write the specification

in any language as long the specification can be translated to Java predicates. A translator for JML [LBR00] has been implemented. Objects up to a given bound are created and checked against the precondition and invariant. The precondition and invariant as mentioned above are available as Java predicates. Each generated object is thus checked against those predicates for validity. Monitoring the read accesses during the predicate execution, fields which do not influence the predicate are detected. This information is used to prune the search space. The generated input objects are used to call the method under test. Similar to TestStudio the postcondition and invariant are used as an oracle.

Both TestEra, Korat and TestStudio try to automatically generate test cases from preconditions, postconditions and invariants. While TestEra and Korat target the language Java, TestStudio targets Eiffel systems. A more fundamental difference however is that TestEra and Korat focus on testing selected methods with as many different inputs as possible. On the other hand the focus of TestStudio was to test as many methods as possible.

3.2 Alternative Strategies

3.2.1 Genetic Programing

Genetic programing [Hol92] tries to automatically find computer programs that best perform a certain task. It uses evolutionary algorithms to optimize a population of programs. A fitness criterion is used to decide whether a program should survive an iteration or not. Genetic programming is typically used for problems where the best solution is hard to find, but a good solution is acceptable. For the purpose of testing, genetic programming could be used to search through the space of test cases.

As part of this thesis, we implemented a proof of concept program which uses genetic programing for testing. The program is based on the *Genetic Algorithm classes* by Ikram [Ikr03]. In this program a population, as can be seen in Figure 3.1, consists of many individuals. Each individual represents a sequence of instructions. The fitness criterion increases the value of those sequences whose executed features either violate a postcondition or throw no exception at all. The goal of testing is to find bugs. Via genetic programming this goal cannot be approached directly. A sequence of instructions does not trigger half a bug and can then be "optimized" to trigger a full bug. However, sequences of instructions that throw no exception are likely to create objects in "interesting" states. Such a sequence can be mutated by adding, removing, or changing an instruction, which could then trigger a bug. Thus the fitness criterion used is not only based on whether a postcondition is thrown or not. It also favors instruction sequences that do not throw exceptions at all. After execution, the energy level of an individual is determined as the sum of the energy of its instructions:

- For every type correct instruction, the energy is increased by 1.
- For every type correct instruction that did not trigger any exception, the energy is increased by 2
- For every type correct instruction that triggered a postcondition violation, the energy is increased by 3

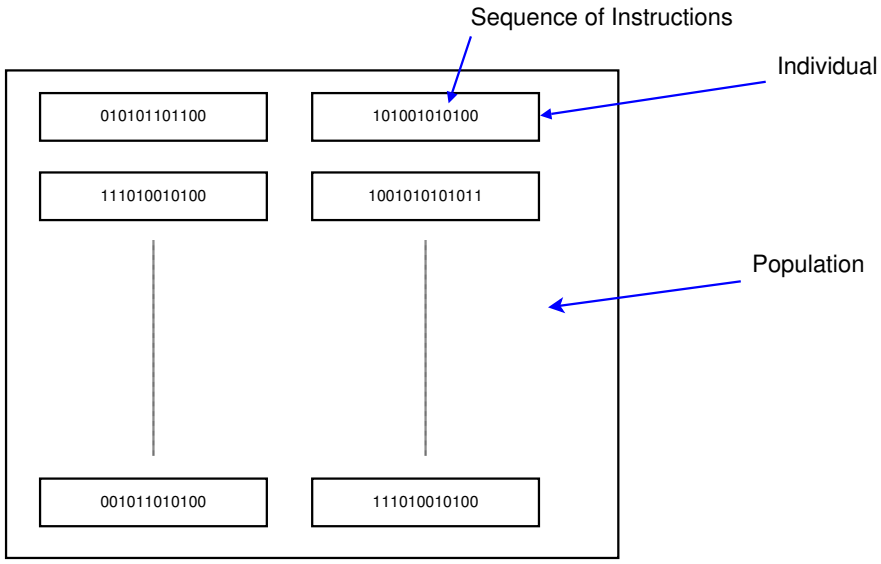


Figure 3.1: Population of Individuals

Figure 3.3 shows the mean and average energy for 2409 iterations on. The class under test was modeled after *LINKED_LIST* (due to technical reasons class *LINKED_LIST* could not be used directly). The population consisted of 300 individuals. Each bit had a probability of 0.05 to mutate. Children were only produced by mutation, not by cross-over. (The genetic programming library used does not support cross-over). A bug has been implanted in the list. It was triggered when calling a certain feature on a list with more than two items. This bug has been found only in the 2409th iteration. As Figure 3.3 shows neither the mean nor the maximal energy level rise steadily. We believe that the reason for this is the way the instructions were represented and mutated. As mentioned above, an individual consists of a sequence of instruction. The i-th instruction is defined as $\text{target}_i := \text{feature}_j(\text{arguments}_i)$. Both feature invocation instructions and creation instructions can be easily mapped to this notation. The identifier feature_j is a number identifying either a creation procedure or a regular feature of a certain type. Figure 3.2 shows the symbolic representation of an example individual. The individual in this figure consists of three instructions. The identifiers feature_{35} and feature_{70} each represent a creation procedure with no arguments. The identifier feature_{20} could stand for a regular feature whose target object is the value returned from the first instruction execution and whose sole argument is the value returned from the second instruction execution. If feature_{20} represents a creation procedure, both target_1 and target_1 serve as arguments because no target object is needed.

After an instruction was executed, its result (if any) is stored in target_i. The identifier arguments_i is the list of actual arguments (for a creation instruction) or the target object and the list of actual arguments (for a feature invocation instruction). If an element in the list has the value x, it refers to the value of target_x. If an argument refers to an target that has not been assigned to yet, it is assumed to be ***Void***. An instruction needs to be *type-correct* to be executed. Instructions that are not type correct, are not executed

$$\begin{aligned} \text{target}_0 &:= \text{feature}_{35}() \\ \text{target}_1 &:= \text{feature}_{70}() \\ \text{target}_2 &:= \text{feature}_{20}(\text{target}_0, \text{target}_1) \end{aligned}$$

Figure 3.2: Symbolic representation of an example individual

and said to be *sleeping*. An instruction can be *type-incorrect* because its arguments are of wrong number and/or type.

An individual is the concatenation of the bit representation of its instructions. An individual consists of a fixed number of bits. An instruction however varies in size depending on the number of arguments its invocation requires. Thus the number of instructions per individual varies as well. If a mutation changes the feature or creation procedure of an instruction, the number of arguments required may change as well. Thus such a mutation may have an effect not only of the local instruction but on all following instructions of the same individual. A modification of the first instruction may render all subsequent instructions type-incorrect. We believe that this drastic effect explains the problems with the mean and maximal energy mentioned above.

The genetic program based tester has been implemented as a proof of concept and is not the main topic of this thesis. The rest of this thesis concentrates on the robust random strategy and the planning based strategy.

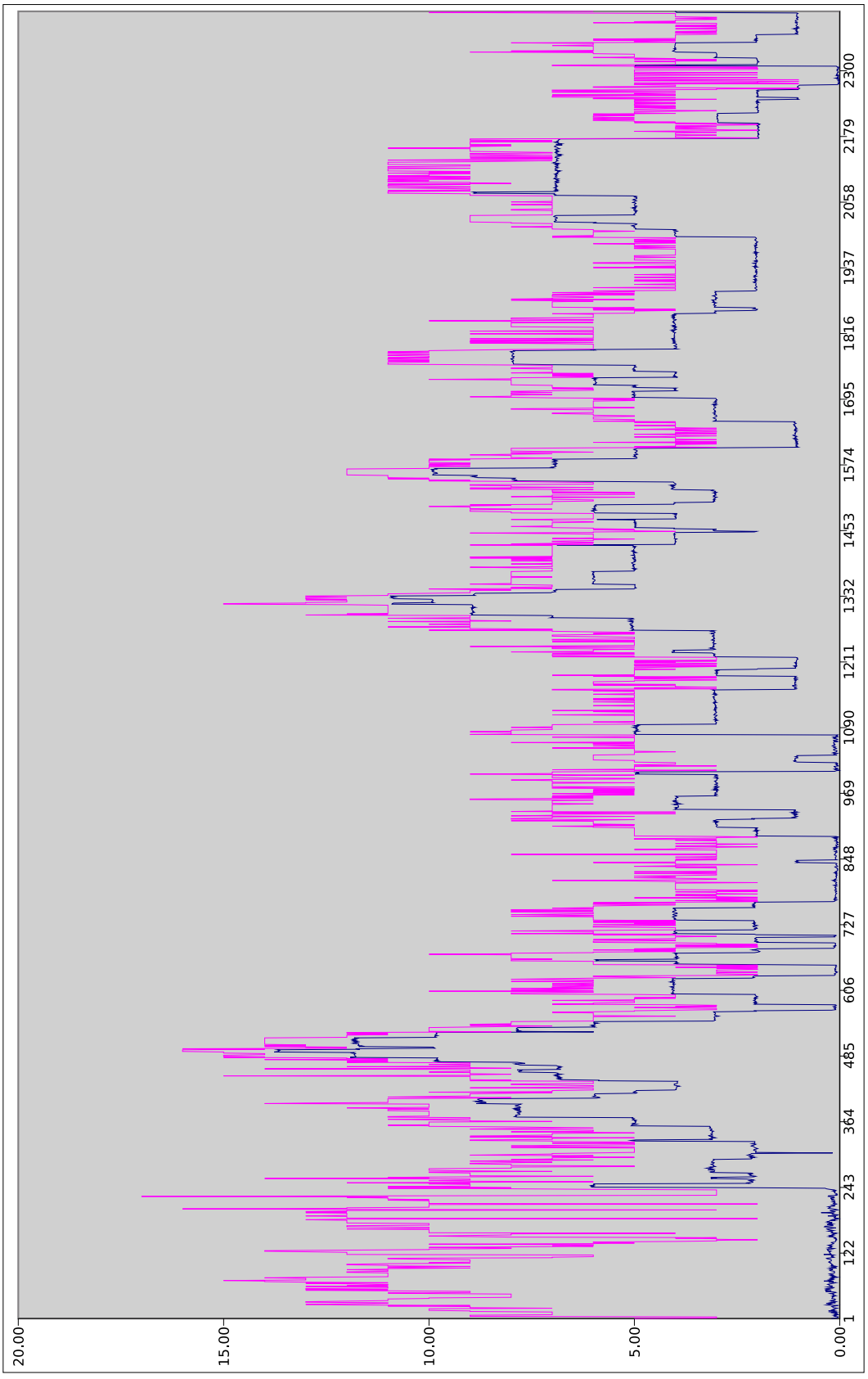


Figure 3.3: X-axis: iteration; Y-axis: mean and maximal energy

Chapter 4

Master/Slave Design

This chapter explains the idea of contract based testing and presents two usage scenarios for it. We describe TestStudio, an application implementing the idea of contract based testing. The chapter explains the main limitation of TestStudio: fragility. A solution (i.e.: the master/slave concept) implemented for TestStudio is presented.

4.1 Contract Based Testing

As mentioned in Section 2.1.2, Design By Contract provides benefits during the design, documentation and debugging processes. Furthermore, the preconditions, postconditions, and invariants can be used for automatic testing. During the debugging process the assertions help to narrow down the source of bugs. To debug a software system an entry point and input data are needed. Libraries do not provide either. Instead, test cases are often used to prevent regressions bugs. Testing frameworks such as JUnit [HT03] greatly simplify the testing process. Such frameworks automate testing, report generation and publishing. The test cases, however, need to be written and maintained manually. Software written with contracts can be used in many cases to generate test cases automatically. The test cases can be used in addition to the existing unit tests. The counter example of bugs revealed by automated testing can be added to the suite of existing unit tests. The following describes two possible usage scenarios for contract based testing:

Push Button Testing – Embedded in the developers' integrated development environment, automatic testing can be started pushing a button. Working in the absence of the developer, the results are presented on her/his return. In this scenario it is important that there is no or only minimal initial configuration that has to be done. The developer should be able to start testing without being asked many questions. Testing should possibly be breadth first. All features should be tested sequentially. Once all features have been tested, the process can start from the beginning with different input data. This makes it possible to abort testing at any time (whenever the developer returns) and still delivers testing results for many features of many classes.

Stand-alone Testing – The automatic tester is installed on a non-interactive computer. It is coupled with the source code version control system. Periodically, the tester

retrieves the latest version of the source code from the version control system and starts testing. Similar to auto-build servers like [Cla04], which periodically compile and test the latest version of the source code, the tester publishes its results (for example: via a web server or email). In this scenario, initial configuration is less of a problem because it is only done once during the installation.

4.2 TestStudio

TestStudio, developed by Greber [Gre04] and Ciupa [Ciu04] at the ETH Zurich, uses contracts for automated testing. The purpose of this thesis was to research and implement improvements to TestStudio. Therefore, TestStudio and its architecture are briefly explained: TestStudio is an application with a graphical user interface. Via this interface the user can create and configure a test project. A test project consists of an Eiffel system, provided via its ace file. The user can select the cluster, classes, and features to test. Several parameters such as the stress level (how often to test a feature) or the test order can be selected. The user can choose those parameters per feature, class, or cluster. After the project has been configured, TestStudio generates, compiles, and executes the test cases. After the test-case execution the tool presents the user the test result statistics. The statistics tell in which features bugs have been found and show the exception for every bug.

4.2.1 Random Strategy

The testing strategy employed by TestStudio is mainly based on randomness. A feature under test is called with random objects. The tester only ensures that the actual arguments conform to the formal ones. The execution of the test cases happens in three phases:

Context Pool Creation – Types that are needed as target objects and arguments for calling the features under test are instantiated. If a type exports multiple creation procedures, one is selected randomly. The context pool stores the objects that will later be used to call the features under test.

Context Pool Modification – To get objects in different states the objects from the pool are modified. Random commands are called on every object from the pool in the hope to change its state.

Testing – The features under test are called using the objects from the pool.

4.2.2 Limitations

The main limitation of TestStudio was its fragility during testing. For example, testing the class *LINKED_LIST* from the EiffelBase library using the TestStudio project-file shown in Section A.1 resulted in the output shown in Listing 4.1.

During testing the runtime crashed and could not recover. Specifically, it was not able to:

- Report what exception caused the crash.

Listing 4.1: Output of original TestStudio when testing *LINKED_LIST*

```
generated_test: PANIC: unexpected harmful signal ...

generated_test: system execution failed.
Following is the set of recorded exceptions.
NB: The raised panic may have induced completely inconsistent information:
```

- Print a stack trace of the moment the exception happened.
- Report test case results.

From the output of the tester the user can only infer that there was some fatal error that triggered a runtime panic. Closer inspection revealed that this particular crash happened during the phase in which commands were called on the previously created context-pool objects. This explains why no test results at all were generated. Several things can cause unexpected behavior of the testing process:

- Eiffel features an *external C* mechanism. Via this mechanism external C functions can be called from within Eiffel. C functions, in general, do not check for invalid function arguments. Calling a function with invalid arguments usually results in undefined behavior. In practice such calls often render the process and/or the Eiffel runtime into an unrecoverable state. This can result in the immediate termination of the process. Alternatively, the process continues but crashes during the execution of a later, potentially perfectly valid, routine call. In the latter situation it is very complicated to locate the source of the problem.
- A bug may cause an Eiffel routine not to terminate. For example the routine could be stuck in an infinite loop. For practical purposes, an infinite loop is not even required: a routine which terminates, but takes an unreasonable amount of time to complete, may give the same impression. The user will likely terminate testing by hand.
- Eiffel features that influence the running process are potentially problematic. For example, the feature *EXCEPTIONS.die* causes the current process to terminate immediately. If this command is called during the modification of the context-pool objects or during the execution of test cases the process terminates immediately. All test cases scheduled after this call will not get executed.
- As explained in Chapter 2, until recently it was not possible to check for the presence of CAT-calls statically. With many compilers CAT-calls are not even detectable at runtime. There the behavior of a CAT-call is undefined and often results in a runtime crash. The tester could check each call it makes, whether it is a CAT-call. This would however only prevent direct CAT-calls. A routine called from the tester could itself perform CAT-call. Detecting direct and indirect CAT-calls is a complex problem and is better left to specialized tools.

The problems mentioned above have been solved by separating the testing process into a master and a slave. The master/slave design is described in detail in Section 4.3.

Another limitation was that test cases were generated as Eiffel source code, compiled and then executed. Compiling the test cases to C and then into executable code with ISE Eiffel takes, on modern hardware, several minutes. This makes it hard to alter the test case for a given feature depending on the information gained on previous tests of the same test run. Changing the test strategy also involves changing the code generator. To solve this inflexibility, a special purpose interpreter has been implemented. It is described in detail in Section 4.3.2.

4.3 Master/Slave

4.3.1 Concept

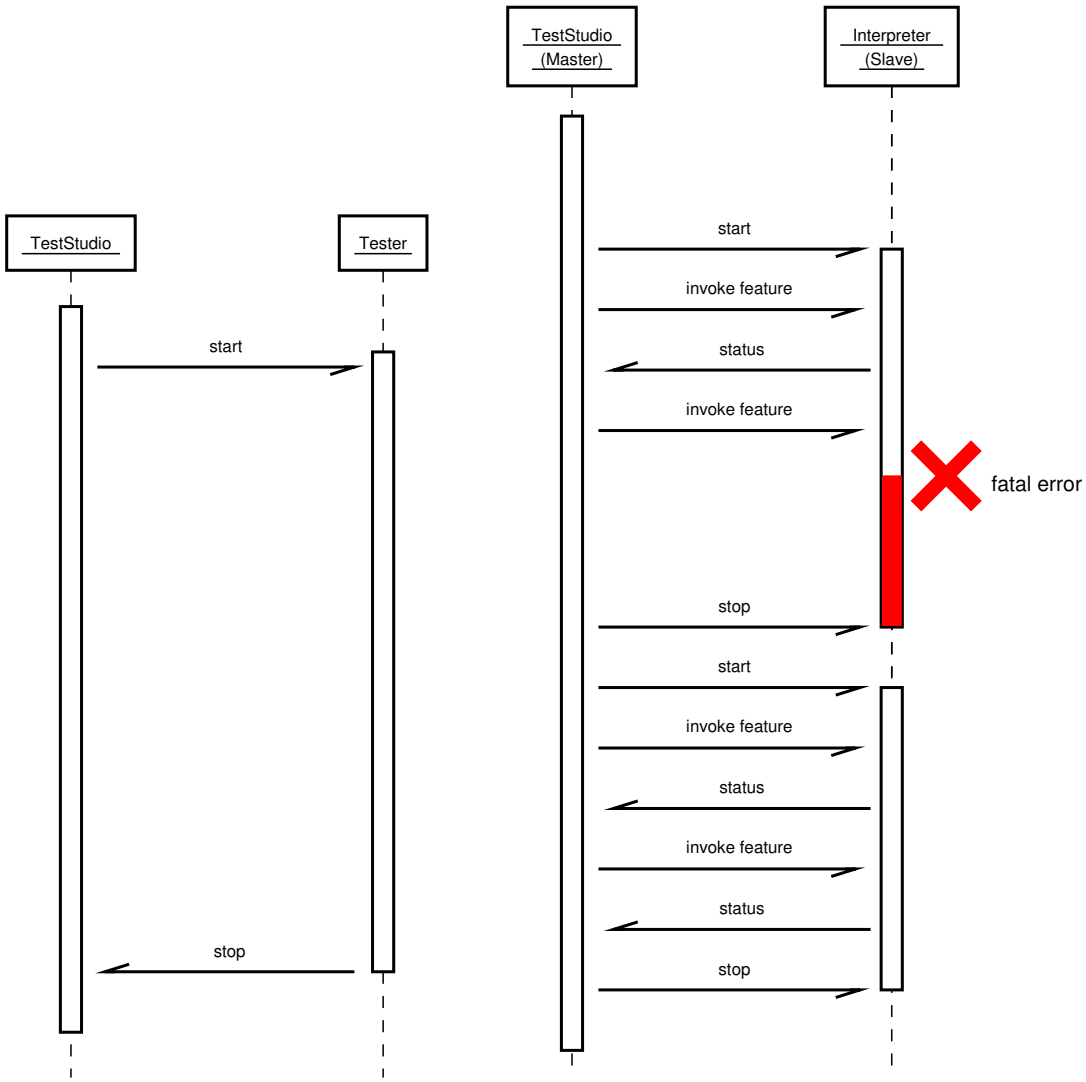
Figure 4.1(a) shows the sequence diagram of the original TestStudio graphical user interface process and the tester process. The user starts the TestStudio application and configures a certain Eiffel system to test. The Eiffel system is parsed, the test cases are generated and compiled. Then the GUI launches the compiled tester as a child process. During the execution of the child process the GUI and the tester do not interact. The GUI waits for the tester to finish and then closes the child process. If a fatal error happens during testing, the GUI process cannot determine during what phase or test case the error happened. The user can only restart the tester. No test after the one triggering the fatal error can be executed. If the fatal error happens during the pool creation or modification phase no test results can be obtained at all.

To eliminate the problem described above, testing is split into two processes: a master and a slave. Determining the test-strategy and the actual test-execution is separated. The master determines which features to call with which arguments in which order. The slave only executes those features. The less work and responsibility the slave has, the less dramatic the consequences of its crash. Figure 4.1(b) shows the interaction between master and slave: the server starts the client and instructs it to invoke a feature. The slave executes the feature and reports the call status back to the server. The master instructs the slave to execute another feature. During the execution of the second feature the slave process crashes. The master gets an bad response (or no response at all) from the slave process. The master times out waiting for a response and finally restarts the slave. The master can now retry or skip the problematic feature call. Note that the slave is instructed to execute feature by feature and not test case by test case. This leaves the master process full flexibility. It can change test cases without recompilation of the slave. The testing-strategy implemented is described in Section 4.3.4.

4.3.2 Interpreter

As described in the last section, the slave needs to be able to execute and create features as requested by the master. The slave will alternate between reading execution requests and executing those requests. In principle, this makes the slave a special purpose interpreter. The interpreter is implemented using a reflection library. Due to limitations in the Eiffel reflection library from ISE Eiffel a reflection library generator has been implemented. The generator is described in the next section.

Table 4.1 shows the commands understood by the interpreter. The commands that



(a) Original TestStudio Sequence Diagram

(b) Master/Slave Sequence Diagram

Figure 4.1: Sequence Diagrams of old and new TestStudio implementation

evaluate preconditions (*pre_create* and *pre_invoke*) are needed by the planner based strategy described in Chapter 5.

If an execution was requested the interpreter responds with the messages *pass* or *fail*. The first message means that the execution was completed successfully. After a *pass* message, it prints the return value (if any) of the execution. If the type of the return value is primitive (e.g.: *INTEGER*, *CHARACTER*, *BOOLEAN*), the value is returned directly. Reference types are stored in an object-repository where they are assigned an slot id. If the return value is of a reference type, its slot id is returned. If an execution was interrupted by an exception, the message *failed* is returned followed by a description of the exception and the stack trace. Note that the object repository is initialized with one object already present: slot 1 contains the object referenced by ***Void***. To invoke a

Request	Arguments	Result
create (Create new object)	• type • creation procedure • target slot • arguments	No result
invoke (Invoke feature on object)	• type • routine • target slot • arguments	Reference or manifest object (depending on feature return type)
pre_create (Evaluate precondition of creation procedure)	• type • creation procedure • arguments	Boolean
pre_invoke (Evaluate precondition of routine)	• type • creation procedure • target slot • arguments	Boolean
quit (Quit the interpreter)		No result

Table 4.1: Interpreter Commands

feature, the target object and any actual arguments needed to call a routine need to be provided depending on their type, just like values are returned:

- Primitive types are provided as constants. For example, the integer *6* is passed as *INTEGER"6"*.
- Reference types are provided via their repository id. For example, the object in slot *10* is provided as *REFERENCE"10"*.

Communication with the interpreter happens via the standard input and output streams. To avoid confusion when reading the output of the interpreter, it prints a mark before and after a feature invocation. This allows the reader of the output to skip over any potential output of the executed feature. Features like *ANY.print*, whose purpose is to print a message on the standard output channel, can thus be tested without confusing the master. If a feature was called that printed the message marking the end of the execution of that feature, the master would still be confused. However, chances for this event are much lower than the chances for a routine printing an arbitrary string. Using a pipe instead of the standard output would not cure the problem completely either, since the pipe could itself be used by features under test. The standard input and output streams are used because they offer portable non-blocking input, whereas pipes do not.

An example session of the interpreter is shown in Listing 4.2:

- An object of type *LINKED_LIST [ANY]* is created and stored in slot 2. The creation terminates successfully and the interpreter reports back that the result of the creation is an object of type *LINKED_LIST [ANY]* and that it is stored in slot 2.
- The feature *force* from type *LINKED_LIST [ANY]* is called on the object in slot 2. The first argument is the integer 55. The routine call terminates successfully.
- The feature *i_th* from type *LINKED_LIST [ANY]* is called on the object in slot 2. The first argument is the integer 12. This triggers a precondition exception. The exception is reported together with the stack trace.

Listing 4.3 shows the log file the interpreter wrote during the session shown in Listing 4.2.

Gobo Based Eiffel Reflection Library

As mentioned earlier the reflection library that comes with ISE Eiffel (as of version 5.5) is too limited to build an interpreter on top of it. The Eiffel Reflection Library (ERL) [Flu04], defines a complete reflection API for Eiffel. The library includes both a complete reflection API and a partial implementation based on the ISE Eiffel reflection facilities. The API is well suited to build a complete interpreter upon. Since the implementation is based on the reflection facilities from ISE Eiffel, it is lacking in the same regards. A complete implementation requires changes to the underlying compiler. Gobo based ERL (GERL) has been implemented as part of this thesis. While GERL is also an incomplete ERL implementation, it is more powerful than the ERL implementation and suffices to build the special purpose interpreter described above. GERL consists of a generator and a runtime library. The generator parses the Eiffel system that should be reflect-able. It generates source code that together with the runtime library provides an ERL implementation that makes the parsed library reflect-able. The generator needs to be provided with an Eiffel system and a list of types that should be reflect-able. Since the generated library needs to be consistent, all types that are recursively needed by those that have been marked to be reflect-able need to be reflect-able too. This is why the generator generates source code not only for the types from the user's list, but also from the type closure of this list.

Listing 4.2: Example Interpreter Session

```
create:"LINKED_LIST [ANY]"."make" (REFERENCE"2")
---multi-line-value-start---
---multi-line-value-end---
status:"pass"
result:REFERENCE"2""LINKED_LIST [ANY]"
done:
invoke:"LINKED_LIST [ANY]"."force" (REFERENCE"2"INTEGER"55")
---multi-line-value-start---
---multi-line-value-end---
status:"pass"
done:
invoke:"LINKED_LIST [ANY]"."i_th" (REFERENCE"2"INTEGER"12")
---multi-line-value-start---
---multi-line-value-end---
status:"fail""3""i_th""CHAIN""valid_key"
---multi-line-value-start---
-------------------------------------------------------------------------------
Class / Object      Routine                 Nature of exception          Effect
-------------------------------------------------------------------------------
LINKED_LIST         i_th @2                 valid_key:
<0000000040025278>  (From CHAIN)            Precondition violated.       Fail
-------------------------------------------------------------------------------
GERL_TYPE_IMP_2     feature_i_th @1
<000000004001B848>                          Routine failure.             Fail
-------------------------------------------------------------------------------
FUNCTION            rout_obj_call_function
<000000004006BAB8>                          Routine failure.             Fail
-------------------------------------------------------------------------------
FUNCTION            apply @3
<000000004006BAB8>                          Routine failure.             Fail
-------------------------------------------------------------------------------
ERL_FUNCTION_IMP    call @3
<000000004006BA18>  (From ERL_ROUTINE)      Routine failure.             Fail
-------------------------------------------------------------------------------
ERL_FUNCTION_IMP    call @2
<000000004006BA18>  (From ERL_FUNCTION)     Routine failure.             Fail
-------------------------------------------------------------------------------
TS_RT_INTERPRETER   call_routine @6
<0000000040019030>                          Routine failure.             Rescue
-------------------------------------------------------------------------------
---multi-line-value-end---
done:
quit:
done:
```

Figure 4.2 shows the main GERL classes and their relations. The classes *ERL_UNIVERSE*, *ERL_TYPE* and *ERL_FEATURE* are part of the ERL interface classes and represent Eiffel universes, types and features respectively. An Eiffel universe is the sum of all types and classes from a given Eiffel system (application or library). The class *GERL_UNIVERSE* is an implementation of *ERL_UNIVERSE*. It can be used to retrieve objects that implement *ERL_TYPE* for all types that are reflect-able. For every reflect-able type the generator creates an implementation of *ERL_TYPE*. In this example the implementations of *ERL_TYPE* are: *GERL_STRING_TYPE*, *GERL_FOO_TYPE*, and *GERL_INTEGER_TYPE*.

4.3.3 Oracle

In a testing system, the oracle decides whether a certain test case passed or failed. A test case that passed means that the execution of the implementation under test behaved as expected. If a test case fails, the implementation did not behave as expected and a bug has been found. The oracle in TestStudio makes its decision based on the contracts of the feature under test. Since every feature has a contract (even if it is implicitly *True*), every feature call can be seen as a test case:

pass – A feature call passes if the execution of the feature triggered no exception.

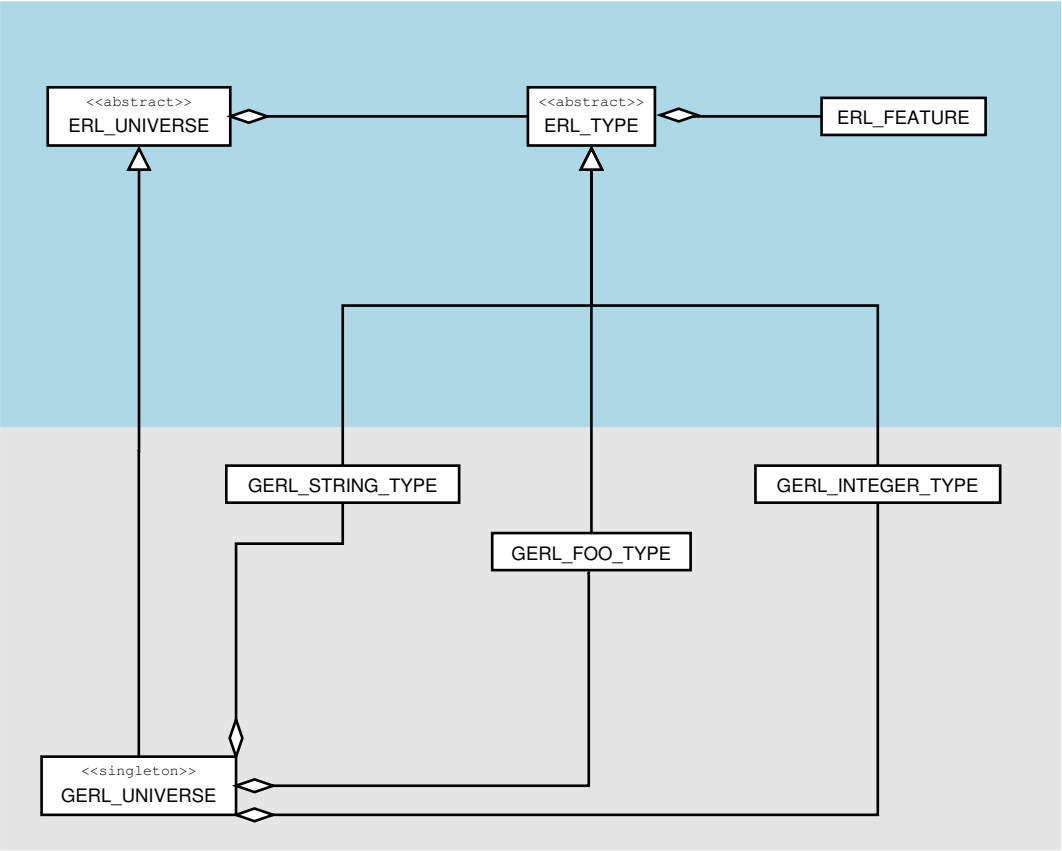


Figure 4.2: GERL UML class diagram

invalid – A feature call is invalid if the precondition of the feature under test was violated or the class invariant was violated prior to the feature execution.

fail – A feature call fails if the execution of the feature call triggers a:

1. Postcondition violation of any feature.
2. Class invariant violation of any feature during or after the execution of the feature under test. Due to aliasing a class invariant violation can go unnoticed. If later an object with a violated invariant is used for testing, the feature under test must not be blamed.
3. Precondition violation occurring in a feature other than the feature under test. A routine that calls another routine without satisfying the called routine's precondition has a bug.
4. Check instruction violation.
5. Bad response, i.e., the slave is not able to respond correctly to the master. This can happen if a feature call damages the process of the interpreter. The interpreter cannot respond properly anymore. In this case the master waits for a timeout and declares the feature call a *bad response*.

A bad response results from one of the below cases:

- A CAT-call happened either directly or indirectly and as a consequence the interpreter crashed. If the CAT-call was directly caused by the test case no proper bug has been found. If the CAT-call was caused by a feature called from the test case a proper bug has been found.
- A feature caused a runtime panic. If the panic happened directly after or during the feature call, the feature contains a bug. If the panic happened some time after the feature call, a bug has been found but not in the feature that was currently tested.
- A feature like *EXCEPTION.die*, whose purpose is to terminate the process it was called from, terminated the interpreter. No bug has been found.

The decision whether a thrown exception signifies a failed or invalid test case cannot be made by only looking at the type and origin of the exception. A precondition violation only signifies a bug if it happens more than one step in the stack trace over the feature that executes the feature under test. Listing 4.4 shows a feature that when called with a positive integer, yields a precondition violation. The violation happens in the feature under test, but only in a recursive sub-call. Here the precondition violation signifies a bug. A call to feature *foo* with a negative integer would trigger a precondition violation directly in the feature under test. This precondition violation signifies an invalid feature call.

4.3.4 Modification of the Random Strategy

If there are many features that cause fatal errors, the interpreter is restarted often. The original random testing strategy from TestStudio first created and modified all objects that were needed for testing. Only then the actual testing started. As described in Section 4.2.2 this may lead to a fatal error even before the actual testing phase starts. To reduce the chances of a fatal error before and during a test case, the random strategy has been modified in the master/slave implementation: only the objects needed for the next feature under test are created. Then, one object is modified and the feature under test is called. This has an important advantage: assume that only a few features may cause a fatal error. Furthermore assume that only a few features depend on types that provide those malicious features. With the original strategy no feature could ever be tested because all creation and modification happens before the actual tests. The problematic features would always be called before the actual testing phase. With the new algorithm the creations and modifications happen directly before each test. All feature tests happening before a fatal error are not affected from the error anymore. When a feature which triggers a fatal error is called, the interpreter will terminate. After every termination the interpreter will be started anew. Eventually, the problematic test case will be skipped and all subsequent features that do not depend on a problematic feature call can be tested.

4.4 Results

This section provides the test results of the class *LINKED_LIST* from the EiffelBase library [Ise03] and the class *DS_LINKED_LIST* from the Gobo library [Bez03].

4.4.1 Explanation

TestStudio generates four files after testing completed:

Summary Table – A table with a line for each tested feature. An example summary table can be seen in Section A.2. The columns have the following meaning:

class – Name of class that was tested.

feature – Name of feature that was tested.

status – Test status: *pass*, *fail*, *no calls* (objects needed could not be created), or *no valid calls* (precondition of feature could not be satisfied)

#pass – Number of calls that passed.

#fail – Number of calls that failed.

#inv. – Number of times a feature could not be called because of a precondition violation.

#bad resp. – Number of times the interpreter sent back an invalid or no response at all.

Detailed Table – A table with a line for each feature call. An excerpt can be seen in Table 4.2.

The columns have the following meaning:

Type – Type name under test.

Feature – Feature under test.

Status – *passed* if no exception occured, *failed* if an exception occurred.

Exception – Exception type.

Exception Receiver – Name of the routine whose execution was interrupted by last exception.

Exception Class – Name of the class that includes the recipient of original form of last exception.

Exception Tag – Tag of last violated assertion clause.

The value corresponds to the queries from class *EXCEPTIONS*.

TestStudio Log – Log file written by the TestStudio process. It shows the communication between master and slave. An example excerpt is shown in Listing 4.6.

Interpreter Log – Log file written by the interpreter process. It contains information about every feature call. An example excerpt is shown in Listing 4.5.

4.4.2 Results for *LINKED_LIST*

The test result summary for class *LINKED_LIST* can be found in Section A.2. Additionally to *LINKED_LIST* also features of the classes *STRING*, *INTEGER_INTERVAL*, *CURSOR* and, *STD_FILES* have been tested automatically.

Figure 4.3 shows the test results of the features from class *LINKED_LIST* only. From *LINKED_LIST* 94 features have been tested, 10 features failed. Of those 10 features, 3 failed only because of a *bad response*. A bad response gets triggered by a bug in most cases, but not all. And finally, 35 features had no or no valid calls.

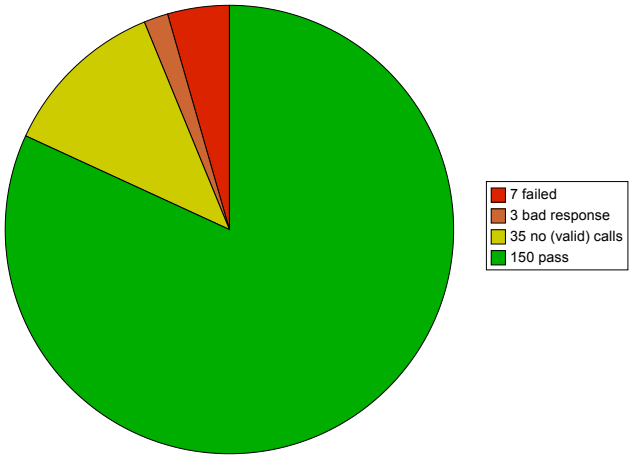


Figure 4.3: *LINKED_LIST* test results

If a feature failed because at least one test case failed, it is certain that a bug has been found. In most cases, using the detailed test results, the proxy log, and the interpreter log, the source of bug can be tracked down. For example, feature *LINKED_LIST.make* is marked as failed. The detailed test result contain the line shown in Table 4.2. The class invariant assertion with the tag *not_both* from class *LINKED_LIST* was violated. The assertion expression for that tag is *not (after and before)*, meaning that the internal cursor of a list must not be both before and after the end of the list. The corresponding log entry from the interpreter log is shown in Listing 4.5. It reveals the attribute state of the linked list right before the call that triggered the bug. From the object state we can infer that the list was empty and the cursor was moved one position past the end of the list (*after* = *True*). Based on these facts the counter example shown in Listing 4.7 can be constructed. Compiling and executing this example indeed successfully reproduces the bug. The source code for *LINKED_LIST.make* is shown in Listing 4.8. The feature *LINKED_LIST.make* does not set *after* to *False*. The author probably assumed that the feature would only ever be used as creation procedure and not in regular feature calls. The feature is however exported to *ANY* both as a creation procedure and as a regular feature. If the feature was only exported as creation procedure and not as a regular feature there would be no bug. A creation procedure is always called on a newly instantiated object. Such an object never has *after* set to *True*.

The bug in *make* has two effects:

- Lists can end up in an inconsistent state where both *before* and *after* are true. Features like *index* and *put_left*, which depend on the correct value of *before* and *after* will not perform as expected anymore.
- From the feature comment of *make*, the programmer is likely to infer that the command empties the list. The implementation however does not do this.

type	feature	status	exception	exception receiver	exception class	exception tag
LINKED_LIST [ANY]	make	failed	Class_invariant	make	LINKED_LIST	not_both

Table 4.2: Detailed Test Result Entry For *LINKED_LIST.make* bug

4.4.3 Results for *DS_LINKED_LIST*

In addition to *DS_LINKED_LIST*, features of the classes *STRING*, *INTEGER_INTERVAL*, *ARRAYED_LIST*, *DS_LINKED_LIST_CURSOR*, *OPERATING_ENVIRONMENT*, *ARRAY*, *SPECIAL*, and *STD_FILES* have been tested automatically. In total, 120 features from class *DS_LINKED_LIST* have been tested. As Figure 4.4 illustrates, 27 features had no or no valid calls. 4 Features failed properly and 4 features failed only because of a bad response.

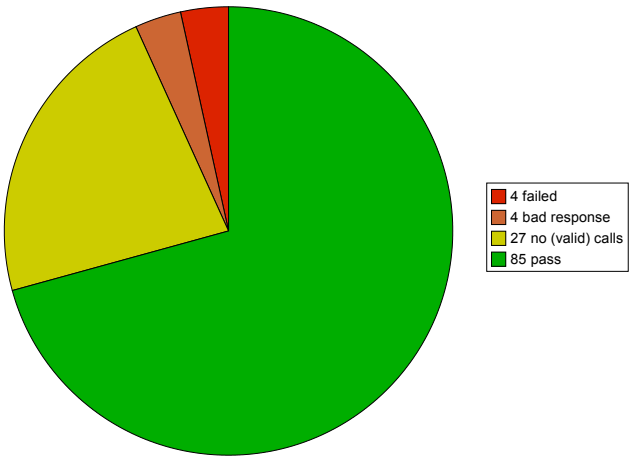


Figure 4.4: *DS_LINKED_LIST* test results

Listing 4.3: Log file of Example Interpreter Session

```
<session>
  <try_create type='LINKED_LIST [ANY]' feature='make'>
    <call_result type='success'/>
    <result static_type="LINKED_LIST [ANY]" dynamic_type="LINKED_LIST [ANY]">
      <slot>2</slot><instance><![CDATA[
      LINKED_LIST [0x40025278]
      active: Void
      first_element: Void
      object_comparison: BOOLEAN = False
      before: BOOLEAN = True
      after: BOOLEAN = False
      count: INTEGER = 0
      ]]>
    </instance>
  </result>
</try_create>
<try_call type='LINKED_LIST [ANY]' feature='force'>
  <operands>
    <operand><slot>2</slot><instance><![CDATA[
    LINKED_LIST [0x40025278]
    active: Void
    first_element: Void
    object_comparison: BOOLEAN = False
    before: BOOLEAN = True
    after: BOOLEAN = False
    count: INTEGER = 0
    ]]>
  </instance>
  </operand><operand><instance><![CDATA[
  INTEGER_REF [0x40029AE0]
  item: INTEGER = 55
  ]]>
</instance>
</operand></operands>
<call_result type='success'/>
</try_call>
<try_call type='LINKED_LIST [ANY]' feature='i_th'>
  <operands>
    <operand><slot>2</slot><instance><![CDATA[
    LINKED_LIST [0x40025278]
    active: LINKABLE [ANY] [0x4011D7C8]
    first_element: LINKABLE [ANY] [0x4011D7C8]
    object_comparison: BOOLEAN = False
    before: BOOLEAN = True
    after: BOOLEAN = False
    count: INTEGER = 1
    ]]>
  </instance>
  </operand><operand><instance><![CDATA[
  INTEGER_REF [0x40120BF0]
  item: INTEGER = 12
  ]]>
</instance>
</operand></operands>
<call_result type='exception'>
  <meaning value='Precondition violated.'/>
  <tag value='valid_key'/>
  <recipient value='i_th'/>
  <class value='CHAIN'>
    <exception_trace>
      <![CDATA[
      ------------------------------------------------------------------------------
      Class / Object      Routine                 Nature of exception          Effect
      ------------------------------------------------------------------------------
      LINKED_LIST         i_th @2                 valid_key:
      <0000000040025278>  (From CHAIN)            Precondition violated.      Fail
      ------------------------------------------------------------------------------
      GERL_TYPE_IMP_2     feature_i_th @1
      <000000004001B848>                          Routine failure.            Fail
      ------------------------------------------------------------------------------
      FUNCTION            rout_obj_call_function
      <000000004006BAB8>                          Routine failure.            Fail
      ------------------------------------------------------------------------------
      FUNCTION            apply @3
      <000000004006BAB8>                          Routine failure.            Fail
      ------------------------------------------------------------------------------
      ERL_FUNCTION_IMP    call @3
      <000000004006BA18>  (From ERL_ROUTINE)      Routine failure.            Fail
      ------------------------------------------------------------------------------
      ERL_FUNCTION_IMP    call @2
      <000000004006BA18>  (From ERL_FUNCTION)     Routine failure.            Fail
      ------------------------------------------------------------------------------
      TS_RT_INTERPRETER   call_routine @6
      <0000000040019030>                          Routine failure.            Rescue
      ------------------------------------------------------------------------------
      ]]>
    </exception_trace>
  </call_result>
</try_call>
</session>
```

Listing 4.4: Bug in recursive feature

```
    foo (i: INTEGER) is
      require
3       i > 0
      do
        foo (-1)
6     end
```

Listing 4.5: Interpreter Log Entry For *LINKED_LIST.make* bug

```
  <try_call type='LINKED_LIST [ANY]' feature='make'>
  <operands>
3 <operand><slot>145</slot><instance><![CDATA[
  LINKED_LIST [0x40706090]
    active: Void
6   first_element: Void
    object_comparison: BOOLEAN = False
    before: BOOLEAN = False
9   after: BOOLEAN = True
    count: INTEGER = 0
  ]]>
12 </instance>
  </operand></operands>
  <call_result type='exception'>
15 <meaning value='Class invariant violated.'/>
  <tag value='not_both'/>
  <recipient value='make'/>
18 <class value='LINKED_LIST'>
  <exception_trace>
  <![CDATA[
21 ----------------------------------------------------------------------------
   Class / Object       Routine                 Nature of exception          Effect
   ----------------------------------------------------------------------------
24 LINKED_LIST          make @2                 not_both:
   <0000000040706090>                           Class invariant violated.    Fail
   ----------------------------------------------------------------------------
27 LINKED_LIST          make @2
   <0000000040706090>                           Routine failure.             Fail
   ----------------------------------------------------------------------------
30 GERL_TYPE_IMP_1      feature_make @1
   <000000000408A3EC8>                          Routine failure.             Fail
   ----------------------------------------------------------------------------
33 PROCEDURE            rout_obj_call_procedure
   <000000000408A7D08>                          Routine failure.             Fail
   ----------------------------------------------------------------------------
36 PROCEDURE            apply @3
   <000000000408A7D08>                          Routine failure.             Fail
   ----------------------------------------------------------------------------
39 ERL_PROCEDURE_IMP    call @3
   <000000000408A7C70>  (From ERL_ROUTINE)      Routine failure.             Fail
   ----------------------------------------------------------------------------
42 TS_RT_TEST_CLIENT    call_routine @5
   <000000000408A34B8>                          Routine failure.             Rescue
   ----------------------------------------------------------------------------
45 ]]>
  </exception_trace>
  </call_result>
48 </try_call>
```

Listing 4.6: TestStudio Log Entry For *LINKED_LIST.make* bug

```
  driver->interpreter: [invoke:]
  driver->interpreter: ["]
3 driver->interpreter: [LINKED_LIST [ANY]]
  driver->interpreter: ["."]
  driver->interpreter: [make]
6 driver->interpreter: [" ()]
  driver->interpreter: [REFERENCE"145"]
  driver->interpreter: []
9 ]
  interpreter->driver [---multi-line-value-start---]
  interpreter->driver [---multi-line-value-end---]
12 interpreter->driver [status:"fail""6""make""LINKED_LIST""not_both"]
  interpreter->driver [---multi-line-value-start---]
  interpreter->driver
      [-------------------------------------------------------------------------------]
15 interpreter->driver [Class / Object      Routine                Nature of exception
      Effect]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [LINKED_LIST          make @2                not_both:          ]
18 interpreter->driver [<0000000040706090>                       Class invariant violated.
      Fail]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [LINKED_LIST          make @2                                   ]
21 interpreter->driver [<0000000040706090>                       Routine failure.
      Fail]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [GERL_TYPE_IMP_1      feature_make @1                           ]
24 interpreter->driver [<00000000408A3EC8>                       Routine failure.
      Fail]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [PROCEDURE            rout_obj_call_procedure]
27 interpreter->driver [<00000000408A7D08>                       Routine failure.
      Fail]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [PROCEDURE            apply @3                                  ]
30 interpreter->driver [<00000000408A7D08>                       Routine failure.
      Fail]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [ERL_PROCEDURE_IMP    call @3                                   ]
33 interpreter->driver [<00000000408A7C70>  (From ERL_ROUTINE)   Routine failure.
      Fail]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [TS_RT_TEST_CLIENT    call_routine @5                           ]
36 interpreter->driver [<00000000408A34B8>                       Routine failure.
      Rescue]
  interpreter->driver
      [-------------------------------------------------------------------------------]
  interpreter->driver [---multi-line-value-end---]
39 interpreter->driver [done:]
```

Listing 4.7: Example to reproduce *LINKED_LIST.make* bug

```
    local
        ll: LINKED_LIST [ANY]
3   do
      create ll.make
      ll.forth
6     ll.make
    end
```

Listing 4.8: Source code for feature *LINKED_LIST.make*

```
    make is
2       -- Create an empty list.
      do
        before := True
5     ensure
        is_before: before
      end
```

Chapter 5

Planning Strategy

This chapter introduces a guided strategy to automatically satisfy non-trivial preconditions. The case is made that the random strategy is not suitable to satisfy non-trivial preconditions. A strategy using a planner is presented as a solution. First the strategy is demonstrated on a simple example by going through the steps manually. An object model is introduced which makes a general conversion mechanism possible. Finally, the implemented conversion mechanism is tested using an example and the results are discussed.

5.1 Motivation

While the implementation presented in Chapter 4 was very effective in that many bugs have been found in the tested classes, it also uncovered a limitation: features with non-trivial preconditions are hardly ever tested. Table 5.1 shows the call statistics of all features of class *LINKED_LIST* (taken from the EiffelBase library) that could not be tested all. The meaning of the columns is as follows:

feature – Name of feature that was tested.

status – Test status.

#pass – Number of calls that passed. Precondition and postcondition was satisfied.

#fail – Number of calls that failed. Precondition was satisfied, postcondition was violated.

#inv. – Number of times a feature was called but its body was not executed because of a precondition violation.

#bad resp. – Number of times the interpreter sent back a bad or no response at all.

The features from Table 5.1 could not be called because of various reasons:

- Feature *default_pointer* has not been tested because it is of type *POINTER*. *POINTER* is a class representing C pointers such as *void**. Features dealing with such types are potentially much more harmful and thus excluded from testing. Since the introduction of the *master/slave* design this is less of an issue, because now a test case

feature	status	#pass	#fail	#inv.	#bad resp.
item	no valid calls	0	0	19	0
replace	no valid calls	0	0	26	0
remove	no valid calls	0	0	29	0
i_th	no valid calls	0	0	26	0
put_i_th	no valid calls	0	0	26	0
swap	no valid calls	0	0	39	0
duplicate	no valid calls	0	0	26	0
default_pointer	no calls	0	0	0	0
do_all	no valid calls	0	0	22	0
do_if	no valid calls	0	0	38	0
there_exists	no valid calls	0	0	32	0
for_all	no valid calls	0	0	27	0

Table 5.1: Features of *LINKED_LIST* that could not be tested

that causes its process to terminate can be recovered from. However, in order to call the features under test on "interesting" objects, a long life of the interpreter is still of benefit.

- The features *do_all*, *do_if*, *there_exists*, and *for_all* all take an agent as argument. Agents are objects representing partially or completely specified computations, a concept similar to *function pointers* and *closures*. TestStudio does not yet know how to create agents and is thus unable to call features which take agents as arguments.
- The other features *item*, *replace*, *remove*, *i_th*, *put_i_th*, *swap*, and *duplicate* have no property which would exclude them from testing, but they still were not called successfully. Listing 5.1 shows the interfaces of those features. The complete interface of class *LINKED_LIST* is shown in the Section A.3 of the appendix.

All untested features from the last group have preconditions stronger than *True*:

item*, *replace*, *remove demand that the list's internal cursor is on a valid position, which implies that the list is not empty.

i_th*, *put_i_th need an *INTEGER* called *i* which refers to a valid list index, which implies that the list is not empty.

swap demands that the list's internal cursor is on a valid position, which implies that the list is not empty and need an *INTEGER* called *i* which refers to a valid list index.

duplicate demands that the list's internal cursor is on a valid position, which implies that the list is not empty or that the cursor is positioned at the end of the list.

All but the last feature require a list that contains at least one element and a list-cursor positioned on one of the list's elements (i.e.: the query *off* returns *False*). Listing 5.2 shows how such a list can be constructed.

Listing 5.1: Interfaces of features from *LINKED_LIST* that have not been tested

```
item: G is
    -- Current item
  require
    not_off: not off
    readable: readable

replace (v: like item) is
    -- Replace current item by 'v'.
  require
    writable: writable
  ensure -- from ACTIVE
    item_replaced: item = v

remove is
    -- Remove current item.
    -- Move cursor to right neighbor
    -- (or 'after' if no right neighbor).
  require
    prunable: prunable
    writable: writable
  ensure
    after_when_empty: is_empty implies after

i_th (i: INTEGER): like item is
    -- Item at 'i'-th position
    -- Was declared in CHAIN as synonym of '@'.
    -- (from CHAIN)
  require
    valid_key: valid_index (i)

put_i_th (v: like item; i: INTEGER) is
    -- Put 'v' at 'i'-th position.
    -- (from CHAIN)
  require
    valid_key: valid_index (i)
  ensure
    insertion_done: i_th (i) = v

swap (i: INTEGER) is
    -- Exchange item at 'i'-th position with item
    -- at cursor position.
    -- (from CHAIN)
  require
    not_off: not off
    valid_index: valid_index (i)
  ensure
    swapped_to_item: item = old i_th (i)
    swapped_from_item: i_th (i) = old item

duplicate (n: INTEGER): like Current
    -- Copy of sub-chain beginning at current position
    -- and having min ('n', 'from_here') items,
    -- where 'from_here' is the number of items
    -- at or to the right of current position.
    -- (from DYNAMIC_CHAIN)
  require
    not_off_unless_after: off implies after
    valid_subchain: n >= 0
```

As trivial as it might seem for a human being, the random strategy was not able to satisfy the property *not off*. The contracts of a class help the reader to infer the semantics of that class. Unfortunately, in practice the contracts provided by the programmer are almost never *sufficiently complete*. To define *sufficient completeness* [Mey97] first defines what constitutes a *correct ADT expression*:

> **Definition: correct ADT expression**
>
> Let $f(x_1, \ldots, x_n)$ be a well-formed expression involving one or more functions on a certain ADT. This expression is correct if and only if all the x_i are (recursively) correct, and their values satisfy the precondition of f, if any.

Then [Mey97] defines *sufficient completeness* as:

Listing 5.2: Instructions to create a *LINKED_LIST* with the property *not off*

```
    local
2       ll: LINKED_LIST [ANY]
      do
        create ll.make   -- Create new object of type LINKED_LIST and
5                        -- attach it to entity 'll'.
        ll.force (Void)  -- Add the 'Void' reference at the end of 'll'.
        ll.finish        -- Move the internal cursor of 'll' to the last
8                        -- element in the list.
      end
```

Definition: sufficient completeness

An ADT specification for a type *T* is sufficiently complete if and only if the axioms of the theory make it possible to solve the following problems for any well-formed expression *e*:

S1 Determine whether *e* is correct

S2 If *e* is a query expression and has been shown to be correct under *S1*, express *e*'s value under a form not involving any value of type *T*.

One problem why contracts are almost never sufficiently complete is the frame problem, which [BMR95] defines as follows:

Definition: frame problem

The inability to express that a procedure changes only those things it has to, leaving everything else unmodified.

It is common practice in Eiffel to assume that things that are not stated in the post-condition do not change. For example, feature *force* from class *LINKED_LIST* (see Listing 5.4), does specify that after the insertion the list contains one more element and that it now contains the inserted element. It does not state that all the other elements, which already were in the list previous to the insertion, remained the same and unchanged. At a first glance it might seem reasonable to let the planner make the same assumption.

However, as can again be seen in the class *LINKED_LIST*, the above assumption does not necessarily hold at all times. For example feature *forth* (see Listing 5.4) states *index = old index + 1*, but also has the effect that after the execution either *not off* or *after* holds. The property *off* holds if and only if at the end of the execution of *forth* if *index > 0 and index <= count* (see invariant of *LINKED_LIST* in Listing A.2) holds. Since *forth* does not specify that *count* does not change, it could, with regards to the postcondition, reduce it, so that *off* would never hold. Since the name *forth* implies that the internal cursor moves forward one position, a human being implicitly assumes that the contained elements are not changed. An automatic reasoner however cannot infer this fact without an understanding of the English language. Thus a planner needs to assume that everything not mentioned in the postcondition can change. As discussed in Section 5.5.3, the planner does assume that an object can only change its direct attributes. While this is not true in general, it helps to reduce the amount of uncertainty.

Listing 5.3: Feature *LINKED_LIST.force*

```
    force (v: like item)
        -- Add 'v' to end.
3       -- (from SEQUENCE)
      require -- from SEQUENCE
        extendible: extendible
6     ensure then -- from SEQUENCE
        new_count: count = old count + 1
        item_inserted: has (v)
```

Listing 5.4: Feature *LINKED_LIST.forth*

```
1   forth
        -- Move cursor to next position.
      require -- from LINEAR
4       not_after: not after
      ensure then -- from LIST
        moved_forth: index = old index + 1
```

Because of the uncertainty in postconditions, it is in general not possible to automatically infer a sequence of instructions that will deterministically fulfill a given precondition, even though such a sequence exists. However, the contracts still bear hints as to what routes can not be successful at all. For example, feature *wipe_out* (see Listing 5.5), which removes all elements from a list, states *is_empty* in its postcondition, which implies (through the classes invariant) *count = 0*. Thus, it can be inferred from the contract that to increase the number of elements in the container *wipe_out* is counter-productive.

Listing 5.5: Feature *LINKED_LIST.wipe_out*

```
    wipe_out
        -- Remove all items.
3     require -- from COLLECTION
        prunable: prunable
      ensure -- from COLLECTION
6       wiped_out: is_empty
      ensure then -- from DYNAMIC_LIST
        is_before: before
```

5.2 Bifrost, A Planner

As described in Chapter 2, a universal planner extracts a universal plan from a planning problem. A problem consists of the planning domain, the initial states and the goal states. A planning domain consists of state variables and actions that change the variables. An action has a *precondition* clause which specifies from what states the action may be called, and an *effect* clause which specifies how the action changes the state. In the bifrost problem language (*extNADL*), an action, furthermore, has a *modifies* clause which states what variables can be changed by the action.

An extracted universal plan tells which action to call in what state in order to reach one of the goal states. The main concept behind the planning strategy is to create a planning problem from Eiffel source code. The planning domain is created from the classes involved, the goal state is the precondition to be reached, and the initial state is a state where no object has been created yet.

Bifrost, the planner used in the implementation, is based on Binary Decision Diagrams and thus limited to finite state domains. Since Eiffel programs are Turing complete, the planning problem is only an abstraction of the original. The amount of precision used for references and integers can be selected at runtime. Negative numbers are currently not supported. The ideas from [DHJ+01] could be used to improve to current abstraction mechanism. The experiments for this thesis have been run with a precision of 2 bits for both integers and references. The loss of precision is acceptable because a tester does not prove the correctness of software but it proves the presence of defects instead. Due to the loss of precision, even if a strong solution can be found, for completely deterministic source code, the resulting test case is not necessarily valid. A valid test case here means a test case that executes the feature under test (without violating its precondition). To remedy this an Eiffel interpreter is used to execute the test case. Since the interpreter works with the original source (without any precision loss) it can decide whether a test case is valid or not.

5.3 Manual Problem Creation

The planner needs a planning problem to extract a plan, so the Eiffel source code has to be converted to a planning problem. The planning problem is formulated in the extNADL language. extNADL is significantly different from Eiffel. Table 5.2 shows the main differences. To highlight the problems present in converting Eiffel to extNADL, a hand-written plan for the problem of satisfying *not off* (the precondition that could not be satisfied by the random strategy) is presented first. To make things even more simple, instead of the class *LINKED_LIST*, the class *SIMPLE_LIST* (see Listing 5.6) is used. A general translation method is described in Section 5.5.

Planning Domain	Eiffel
• Static memory (finite number of variables) • Actions with no arguments • Only natural numbers and booleans (no composite types) • Limited, side-effect free and non recursive expressions	• Dynamic memory management (dynamic object creation) • Routines with arbitrary many arguments • Integers, floating point values, composite objects, references, ... • Multiple inheritance (respectively multiple dispatch) • non-functional, recursive expressions

Table 5.2: extNADL versus Eiffel

The complete process from Eiffel source code to a universal plan is shown in Figure 5.1.

The Eiffel source code is converted into a planning problem, then the planner is asked to extract a plan from the problem. Note that for the *SIMPLE_LIST* example the conversion from Eiffel to extNADL was done by hand.

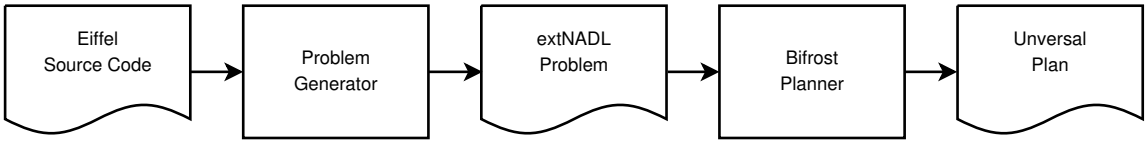


Figure 5.1: Plan Generation

5.3.1 Eiffel Source Code

Listing 5.6 shows a stripped down version of the class *LINKED_LIST* called *SIMPLE_LIST* where no manual frame information has been added. It is too simple to even handle to most important job of a list: to store objects. Nevertheless it is powerful enough to expose the obstacle that prevented the random strategy from testing several features of the class *LINKED_LIST*: it can represent the property *off* and features the two commands (*force* and *finish*) that are needed to create a list where *off* does not hold.

Listing 5.6: *SIMPLE_LIST*

```
1 class SIMPLE_LIST

  feature -- Queries
4
    index: INTEGER
      -- Index of item at current cursor position
7
    count: INTEGER
      -- Number of items in this list
10
    is_last: BOOLEAN
      -- Is internal cursor on the last item?
13
    off: BOOLEAN
      -- Is internal cursor not at a valid position?
16
  feature -- Commands

19   force
      -- Add an element.
    ensure
22     count = old count + 1

    finish
25     -- Move internal cursor to last item.
    ensure
      count > 0 implies is_last
28
  invariant

31   count >= 0
    index >= 0
    index <= count + 1
34   is_last = ((count > 0) and (index = count))
    off = ((index = 0) or (index = count + 1))

37 end
```

5.3.2 ExtNADL Problem

Listing 5.7 shows a hand written planning problem for *SIMPLE_LIST* with the goal state $\neg off$. See Chapter 2 for a brief description of the extNADL syntax. A problem consists of five clauses:

VARIABLES – The four Eiffel attributes of class *SIMPLE_LIST* are translated to extNADL variables. The Eiffel type *BOOLEAN* has a direct equivalent in extNADL, the type *bool*, while the type *INTEGER* doesn't. The extNADL type *nat(x)*, which represents natural numbers with the precision x, is used to represent *INTEGER*. A better solution would be to introduce a new type in extNADL representing signed numbers, but due to time constraints this has not been implemented.

SYSTEM – The two Eiffel commands *force* and *finish* have been translated to extNADL actions. Notice that the effect clause is a conjunction of the postcondition and the invariant, since extNADL does not feature the invariant concept. The modifies clause is present only in extNADL and has been set according to the commands implementation and not their contracts. Because the modifies clause has been set by hand the frame problem has been solved.

ENVIRONMENT – Bifrost is able to create plans for systems with multiple agents that can act concurrently. Currently only single threaded test cases are considered and this feature is not needed.

INITIALLY – The initial state is taken from the list's implementation, not from the contract.

GOAL – The goal consists of every state in which $\neg off$ holds. This is directly derived from the condition the random strategy had problems with.

5.3.3 Universal Plan

Table 5.3 shows the strong solution that bifrost extracted from this planning problem. The table tells what action can be called in what state to reach one of the goal states. The values represent bit patterns; a star ("$\star$") represents both 0 and 1. Executing the solution leads to the following path:

1. The initial state is $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests to execute *force*.
2. The state now changed to $count = 1 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests *finish*.
3. A goal state was reached: $count = 1 \wedge index = 1 \wedge is\_last \wedge \neg off$.

Note that a strong solution can only be extracted because the problem was created by hand. It contains information that cannot be inferred form the source code automatically: Eiffel programmers usually only state, in the postcondition, those things that change and

Listing 5.7: Problem for *SIMPLE_LIST*

```
VARIABLES

    nat(2) count
    nat(2) index
    bool is_last
    bool off

SYSTEM

  agt: list
  force
    mod: count
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: index, is_last, off
    pre: true
    eff:   (count' > 0) => (is_last' = 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))


ENVIRONMENT

INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off
```

not those that don't. As discussed above, it needs to be assumed that a routine changes potentially anything, except if it states otherwise. An automatic problem generator would need to put all variables into the modifies clause. If the modifies clauses in the *SIMPLE_LIST* problem would list all variables, no strong solution could have been extracted. Extracting a weak solution leads to further problems, because it assumes that everything uncertain will act in favor of the agent. Features inherited from the default ancestor *ANY* often have no postcondition (which is equivalent with a *True* postcondition), for example the feature *print*. In practice those features need to be included for planning since they are part of the interface of every class. A weak solution extracted from a problem that includes actions that have the postcondition *True* (such as *print*) will always suggest the execution of *print* regardless of the goal. The *True* postcondition restricts nothing and per weak solution definition it is assumed to act in the agents favor (i.e.: reach the goal immediately).

Furthermore, the current problem only represents the very simple class *SIMPLE_LIST* and it cannot be easily extended to represent the much more complex class *LINKED_LIST*. There is no notion of *object*, *object creation*, *reference* or *polymorphism* . A list contains references to objects of many types like *CURSOR* to store the internal cursor position. *CURSOR* is only an abstract class, *LINKED_LIST* uses the *CURSOR* descendant *LINKED_LIST_CURSOR* internally. A general approach must be able to represent this and is described in Section 5.5.

count	index	is_last	off	
00	⋆0	0	1	force
01	⋆⋆	⋆	1	finish
1⋆	⋆⋆	⋆	1	finish

Table 5.3: Strong solution for *SIMPLE_LIST*

5.4 Dealing with Uncertainty

Before introducing a general object model, in the next section, uncertainty in postconditions is discussed. An object model (while necessary for a general conversion mechanism) is complex and would only distract from the problems stemming from uncertainty. As mentioned earlier, the problem shown in Listing 5.7 was manually tuned to contain a strong solution. Listing 5.8 shows the same problem, except that every action can modify every variable. This is the model obtained when frame information is not available. The extracted weak solution is shown in Listing 5.4. Note that a strong solution does not exist for this problem. Executing this solution is less straightforward:

1. The initial state is $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests to execute *add* or *finish*. The solution has no preference for either. Executing *finish* does not change the object state, only *force* is effective in this state. When the test is run repeatedly and in case of multiple suggested actions one is chosen randomly eventually *force* will be executed in this state.
2. (After executing *force*) the state now changed to $count = 1 \wedge index = 0 \wedge \neg is\_last \wedge off$. The solution suggests *finish*.
3. The goal state was reached: $count = 1 \wedge index = 1 \wedge is\_last \wedge off$

count	index	is_last	off	
0⋆	⋆⋆	⋆	1	force
10	⋆⋆	⋆	1	force
⋆⋆	⋆⋆	⋆	1	finish

Table 5.4: Strong solution for second *SIMPLE_LIST* problem

If the goal is strengthened to $\neg off \wedge count = 2$ and the precision is increased to 3 bits an even bigger problem shows. Listing 5.9 shows the new problem; Table 5.5 the new weak solution.

In the start state $count = 0 \wedge index = 0 \wedge \neg is\_last \wedge off$ only *finish* is suggested. Executing the solution the goal can never be reached. There are two problems here:

- The action *finish* is suggested in the start state. Executing it has no effect to the state of the list. Why does the solution suggest it then? The planner only knows the

Listing 5.8: Second *SIMPLE_LIST* Problem

```
VARIABLES

  nat(2) count
  nat(2) index
  bool is_last
  bool off

SYSTEM

  agt: list
  force
    mod: count, index, is_last, off
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: count, index, is_last, off
    pre: true
    eff:   (count' > 0) => (is_last' = 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

ENVIRONMENT

INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off
```

count	index	is_last	off	
001	⋆⋆⋆	⋆	⋆	force
00⋆	⋆⋆⋆	⋆	⋆	finish
010	⋆⋆⋆	⋆	1	finish
011	⋆⋆⋆	⋆	⋆	finish
1⋆⋆	⋆⋆⋆	⋆	⋆	finish

Table 5.5: Strong solution for third *SIMPLE_LIST* problem

planning problem and not the real list implementation. The postcondition of *finish* is too weak. An implementation that would add several elements to the list and then move the cursor to the last one would not violate the postcondition of *finish*. Per definition the weak solution assumes everything uncertain to be in favor of reaching the goal. Thus the planner suggests *finish*.

- Even though *force* is perfectly suited to be executed in the initial state, it is *not* suggested. The planner extracts the weak solution via a backwards breadth-first search. Starting with the goal states, it calculates pre-images and stops when either a fix point or one of the goal states has been reached. According to the domain, *finish* could be implemented in a way to reach the goal state in only one step. The planner only extracts the shortest optimistic path. Starting from the goal states, the planner find that *finish* can lead from a start state directly to a goal state in its first

Listing 5.9: Third *SIMPLE_LIST* Problem

```
VARIABLES

    nat(3) count
    nat(3) index
    bool is_last
    bool off

SYSTEM

  agt: list
  force
    mod: count, index, is_last, off
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: count, index, is_last, off
    pre: true
    eff:   (count' > 0) => (is_last' = 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))


ENVIRONMENT

INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off /\ count = 2
```

backward step. Thus the planner stops after one iteration. Why is *force* contained in the table at all? Because, according to its postcondition, it could be implemented not only to add one element, but also to move the cursor to the last position. Note however that the postcondition explicitly states that exactly one element is added. Thus *force* cannot be implemented to reach the goal in one step. The action *force* is in the table because it can reach the goal when already one element is inserted.

5.4.1 Planning with Learning

The weak solution extraction algorithm is too optimistic. Many routines from *ANY* have *True* as their postcondition. The weaker the postcondition, the worse the effect of the planner's optimism. Every class implicitly or explicitly inherits from *ANY*. These inherited routines alone make the weak solution extraction algorithm ineffective for all practical purposes. Since the testing process has to be completely automatic, strengthening the postconditions in the Eiffel source code by hand is not possible. They can, however, be strengthened in the problem automatically: the interpreter, while executing a plan, can record the state transitions. When a given solution could not reach the goal, the recorded transitions can be feed back to the problem. The following steps describe the new planning with learning algorithm:

1. A planning problem is created from the Eiffel source code.

2. A weak solution is extracted from the problem.

3. The solution is executed and the state transitions are recorded: The object state of every feature execution is recorded before (state) and after the execution (state′).

4. If, after a certain number of feature executions, the goal is reached, the process stops and the feature can now be tested. If the goal could not be reached the recorded state transitions are added to the problem: Let post_i be the postcondition of a feature of the current domain: the new postcondition is then defined as: $\text{post}_{i+1} := \text{post}_i \wedge (\text{state} \rightarrow \text{state}')$

5. Go to Step 2.

If, after a certain number of iterations, the goal is still not reached the process ends. Note that currently the domain only describes a single list object. When using the object model, which is described in Section 5.5, multiple objects can be represented. There, all objects of the same type and state are assumed to behave equal. Taking the list problem from Listing 5.9 and its solution from Table 5.5, only *force* would ever be suggested for execution. The recorded transition would be *(count = 0 ∧ index = 0 ∧ off ∧ ¬is_last) → (count' = 0 ∧ index' = 0 ∧ off' ∧ ¬is_last')*. Augmenting the problem with this recording results in the problem shown in Listing 5.10. The corresponding solution is shown in Table 5.6. Note that, in general, routines are not required to be deterministic. The postcondition augmentation assumes deterministic behavior. In practice only very few routines are non-deterministic. Thus the augmentation will work in most situations.

Listing 5.10: Fourth *SIMPLE_LIST* Problem

```
VARIABLES

    nat(3) count
    nat(3) index
    bool is_last
    bool off

SYSTEM

  agt: list
  force
    mod: count, index, is_last, off
    pre: true
    eff:  (count' = count + 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1)))

  finish
    mod: count, index, is_last, off
    pre: true
    eff:   (count' > 0) => (is_last' = 1) /\
      (count' + 1 > 0) /\
      (index' + 1 > 0) /\
      (index' < count' + 2) /\
      (is_last' <=> ((count' > 0) /\ (index' = count'))) /\
      (off' <=> ((index' = 0) \/ (index' = count' + 1))) /\
                    ((count = 0 /\ index = 0 /\ off /\ ~is_last) =>
                     (count' = 0 /\ index' = 0 /\ off' /\ ~is_last'))

ENVIRONMENT

INITIALLY
  count = 0 /\ index = 0 /\ off /\ ~is_last

GOAL
  ~off /\ count = 2
```

count	index	is_last	off	
000	000	0	1	force
001	⋆⋆⋆	⋆	⋆	force
00⋆	⋆⋆⋆	⋆	⋆	finish
010	⋆⋆⋆	⋆	1	finish
011	⋆⋆⋆	⋆	⋆	finish
1⋆⋆	⋆⋆⋆	⋆	⋆	finish

Table 5.6: Strong solution for fourth *SIMPLE_LIST* problem

The new solution now suggests *force* both with an empty list and a list with one element. Although it still suggests paths that do not lead to the goal state, the correct path is now included. Selecting a random action in states where multiple actions are suggested will eventually reach the goal.

5.5 A General Object Model in the Planning Domain

The generated planning problems must have a notion of object. extNADL doesn't have such a concept. To solve this, an object model is placed on top of extNADL. This is similar to what a compiler for an object oriented language does when compiling to native code. The language a computer directly understands does not have a notion for objects either. Since the planning domain must be finite state, the maximal number of objects possible is limited as well. Object state is stored in *slots*. With a (selectable) reference precision of n there are 2^n slots available. Their indexes range from 0 to $2^n - 1$. A slot with the index x has the following variables:

- A boolean variable (named *is_alive_x*) that tells if the slot contains an object.
- A natural number called *type_id_x* that contains the (dynamic) type of the object stored in this slot.
- A variable for each argument-less query of every included Eiffel type.

5.5.1 Example

Listing 5.11 shows two simple classes. Class *FOO* has two integer attributes that can be incremented and decremented. Class *BAR* has a reference to an object of type *FOO*. The corresponding object model can been seen in Figure 5.2. Note that Figure 5.2 shows only the storage aspect of the model, not the behavioral. A complete example is discussed in Section 5.6.

Four things are worth noting:

1. The model does not use memory in most ergonomic way, but instead focuses on simplicity and readability. With the current model every type has storage for every (argument-less) query for every type. At one time only one object of a given type

Listing 5.11: Example classes for object model

```
class interface FOO

feature
  i: INTEGER

  j: INTEGER

feature

  increase_i
    ensure
      i = old i + 1

  decrease_i
    ensure
      i = old i - 1

  increase_j
    ensure
      j = old j + 1

  decrease_j
    ensure
      j = old j - 1

end

class interface BAR

feature

  ref: FOO

end
```

can be stored in a slot. As soon as more than two types, each with more than 0 argument-less queries, are modeled there are unused variables.

2. The current model does not support queries with arguments. On first sight it seems easier to only store attributes in the model and retrieve the result of functions via actions. Functions are often used in Eiffel expressions. The planning language does not have a notion of functions and thus Eiffel expressions that contain calls to functions can not be translated. In this model, argument-less queries (i.e.: attributes and functions without arguments), are both treated as storage and can be used in expressions. Boolean expressions that contain functions with arguments are converted to *true* since they are not yet supported.

3. References are stored as natural numbers. The value of the number is the slot number of the object that is referenced.

4. The actual implementation does not use class and feature names to build an attribute variable. Instead, the identifier numbers assigned by the Eiffel parser are used. This is because type and feature names are not trivially convertible to extNADL identifiers. (For example a generic type contains square brackets, which are not allowed in extNADL identifiers.)

5.5.2 The Type *NONE*

In the extNADL object model, to represent that a reference is not attached to any object the variable is assigned the value 0. In Eiffel, unlike in many other object oriented

Slot No.	Data
0	bool is_alive_0
	nat(2) type_id_0
	nat(2) attr_FOO_i_0
	nat(2) attr_FOO_j_0
	nat(2) attr_BAR_ref_0
1	bool is_alive_1
	nat(2) type_id_1
	nat(2) attr_FOO_i_1
	nat(2) attr_FOO_j_1
	nat(2) attr_BAR_ref_1
2	bool is_alive_2
	nat(2) type_id_2
	nat(2) attr_FOO_i_2
	nat(2) attr_FOO_j_2
	nat(2) attr_BAR_ref_2
3	bool is_alive_3
	nat(2) type_id_3
	nat(2) attr_FOO_i_3
	nat(2) attr_FOO_j_3
	nat(2) attr_BAR_ref_3

Figure 5.2: Object Model

languages, the type hierarchy is a lattice. As illustrated in Figure 5.3, every type inherits from *ANY*. Class *NONE* inherits from every type. Class *NONE* is of course not a real class, but rather a concept. In every system there is exactly one object of type *NONE* which can be accessed via the keyword ***Void***. ***Void*** serves a similar purpose as *null* in Java, but it does not need a special rule in the type system. Since *NONE* conforms to every class in a given system, its sole instance referenced via ***Void*** can be assigned to any entity.

In the extNADL object model the object referenced by ***Void*** does not need special treatment either: The *INITIAL* clause simply states that no object is alive, except the one in slot 0 whose type is *NONE*. Assuming the id 0 for type *NONE* this can be written as $\text{alive\_0} \wedge (\text{type\_0} = 0) \wedge \neg\,\text{alive\_1} \wedge \neg\,\text{alive\_2} \wedge \neg\,\text{alive\_3}$.

5.5.3 Routines

Figure 5.2 only shows the variable declarations for the domain. Eiffel routines are converted to actions, but instead of one action per Eiffel routine, $2^n - 1$ actions per routine are

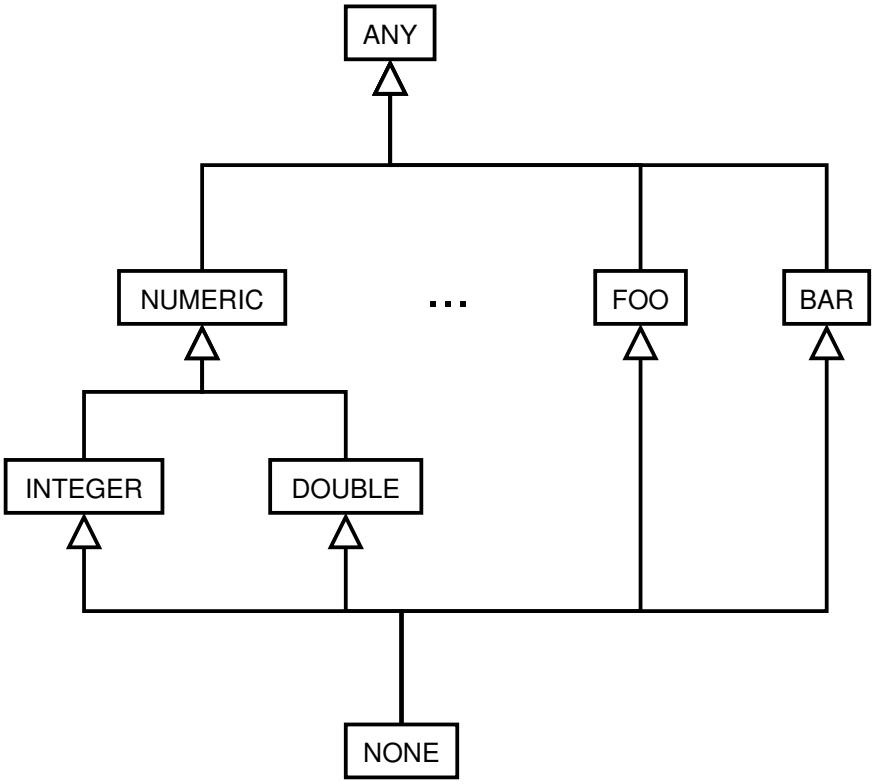


Figure 5.3: Eiffel Type Lattice

created (where n is precision for references). The subtraction by one is an optimization because the object referenced by ***Void*** has no public features. It might seem strange that every slot, except the first one, has its own set of actions. The frame problem forces such drastic measures. Every action needs a modifies clause which states what variables can be modified by its action. The more variables there are in a modifies clause, the more uncertain is the effect of its action. To keep the number of variables in the modifies clause limited, it is assumed that a routine can only modify the attributes from the object it has been called on (which is in general not true). This can only be expressed if there is an action per slot. Class *FOO* has four routines, with a reference type precision of two bits, hence 12 actions need to be generated.

Modifies Clause – lists all variables representing attributes. Note that this makes the planner assume that an object only modifies its own attributes. Of course in general an object can modify not only its direct attributes, but also the attributes of its attributes (recursively). However, we expect that in practice the above assumption will hold for many problems. The limitation was introduced to reduce uncertainty. Without this reduction of uncertainty, extracted plans get too optimistic. A routine would be assumed to possibly change all attributes of all instances not mentioned in the routines postcondition, even though those instances are completely decoupled.

Precondition Clause – an action representing a routine of object *target* must only be

called if the object in slot target is alive and its type includes the routine in question. Additionally the precondition of the routine must hold. The precondition of an action representing a routine is: alive_target ∧ type_target = type_id ∧ routine_precondition.

Effect Clause – an action representing a routine has the routine's postcondition as effect.

5.5.4 Creation Routines

Creation routines are handled similar to regular routines:

Modifies Clause – an action representing the creation procedure of object *target* lists all variables representing the objects attributes, *alive_target*, and *type_target*.

Precondition Clause – an action representing a routine of object target must only be called if the object in slot target is alive and its type includes the routine in question. Additionally the precondition of the routine must hold. The precondition of an action representing a routine is: alive_target ∧ type_target = type_id ∧ routine_precondition

Effect Clause – an action representing a routine has the routine's postcondition as effect and alive_target ∧ type_target = type_id.

5.5.5 Expressions

To generate a problem, expressions need to be converted from Eiffel to extNADL in various places:

- Precondition clauses
- Effect clauses
- Initial states clause
- Goal states clause

The lack of references also shows when converting expressions from Eiffel to extNADL. As a result, extNADL expressions are often much longer than their counterpart in Eiffel. Let's assume that the precondition of a routine from class *FOO* is $i = const$, where *const* is an arbitrary natural number. The first row in Table 5.7 shows its translation to extNADL. *target* is assumed to be the slot id of the action. In this conversion the extNADL expression is not any more complex than its Eiffel counterpart. As soon as references come into play the extNADL expressions start to grow. For example, assume a precondition of a routine in class *BAR* states $ref.i = const$. Again, *const* is assumed to be a constant, *target* the slot id of the action where the expression occurs. Since the i to compare depends on the value of *ref*, a manual quantification using a conjunction of implications is needed. Such quantifications significantly increases the size of the underlying BDD and affect the performance of the solution extraction.

Eiffel uses keyword ***old*** (in postcondition expressions) to identify the state of an entity before the execution of a routine. In extNADL primed variables ("$'$") are used to identify the state of a variables after the execution of a feature. Additionally Eiffel allows the

Eiffel	extNADL
$i = const$	$i\_target = const$
$ref.i = const$	$(ref\_target = 0 \rightarrow (i\_0 = const)) \wedge$ $(ref\_target = 1 \rightarrow (i\_1 = const)) \wedge$ $(ref\_target = 2 \rightarrow (i\_2 = const)) \wedge$ $(ref\_target = 3 \rightarrow (i\_3 = const))$

Table 5.7: Expression Conversion

old keyword as a prefix operation on an arbitrary expression, whereas in extNADL only variables (and not expressions) can be primed. An expression converter has been implemented that walks an Eiffel expression from its leaves to its root node and creates the corresponding extNADL expression. Because of time constraints not all Eiffel expressions are supported (e.g.: agent expressions, expressions containing primitive types other than *INTEGER* and *BOOLEAN*, expressions containing function calls that take arguments). Unsupported boolean expressions (in extNADL all top level expressions are boolean) are converted to *true*. Such a conversion makes the generated domain too optimistic. It adds infeasible paths to the solution, which have to be detected by the interpreter. Replacing unsupported expressions with *false* would yield in more pessimistic domains. Using *false*, many problems would not contain a solution at all, which is clearly the worse scenario.

5.6 The *COUNTER* Example

The implemented algorithm has been tested on the Eiffel class shown in Listing A.3. Class *COUNTER* has two integer attributes: *i* and *j*. Each integer attribute has two commands to increase and decrease the value. Note that the frame problem not only is present in the routines inherited from class *ANY* but also in the commands from class *COUNTER*. The feature to test is *foo*. Its precondition is $(i = j - 2) \wedge (i > 0)$.

The problem generated in the first iteration is shown in Listing A.4. To reach the goal, the initial problem has been augmented 5 times with transitions recorded during execution. The execution paths are shown in Section A.6.

Chapter 6

Conclusion and Outlook

Contract based testing promises to be an effective new testing method. Supplementing rather than replacing traditional unit testing, contract based testing provides a low entry barrier to testing. In this work we proposed two strategies for automatically testing Eiffel programs. Based on the work from Greber and Ciupa and the idea of contract based testing, we described a robust random strategy and a planning-based strategy. While the robust random strategy proves to be very effective, it falls short on routines with strong preconditions. Even though the planning-based strategy is not yet complete enough for practical usage, it promises to step in where the random strategy falls short.

6.1 Master/Slave Design

The master/slave design moved logic from the process that executes the tests, to the controlling GUI process. This change required the implementation of a special purpose interpreter and a reflection library generator. We showed that with the new design, the tester can recover from fatal errors. Our experiments prove that a slightly modified random strategy together with the master/slave design can reveal bugs in real world classes. Testing two classes and a total of 214 features the new implementation revealed bugs in 11 features. Both classes were taken unmodified from production quality libraries. The fact that we found several real bugs in this two classes is clear evidence of TestStudio's potential.

The interpreter and reflection library generator we implemented as part of this thesis and lack support for infix features, default creation procedures and agents. The reflection library generator is not dependent on other parts of the TestStudio code base. Further work on the generator would not only benefit TestStudio but might also interest other Eiffel projects that need a more complete reflection facility.

6.2 Planning Strategy

Features with strong preconditions present a problem to the random strategy. For such features a more guided approach is necessary. The proposed planning strategy borrows techniques from the planning and model checking domains. Because of the inherent complexity, automated reasoning about software system is infeasible without accepting a loss

of precession. The planning-based strategy abstracts the Eiffel source code to a (less precise) planning problem. The interpreter validates potential solutions by executing them. To overcome the uncertainty present in contracts, we combined planning with learning. The implementation is not yet complete enough for real world classes. Its main limitation is the lack of support for routines with arguments. However the strategy showed promising results on the *COUNTER* example. The contracts for the class *COUNTER* exposes the typical uncertainties present in Eiffel contracts. Conventional planning algorithms such as the weak and strong solution extraction algorithms failed at extracting a useful plan. However, the combination of the weak solution extraction algorithm in conjunction with learning solved the problem in only 6 planning/learning iterations.

6.3 Future Work

Using specification in the form of preconditions, postconditions and invariants to automatically generate test cases is an active research topic. Korat [BKM02] uses an interesting approach that emphasizes good input-data coverage, while the focus of TestStudio lies on good feature coverage. A combination of both approaches seems promising. Work by Christian Rilke at the ETH Zurich regarding coverage analysis is currently underway which can be seen as a first step in this direction.

Recently research has gone into program abstraction for the purpose of finite state verification. The abstraction algorithm used by the planner-based strategy could greatly benefit from the ideas proposed in [DHJ+01]. The transition recording, which is needed to augment plans could be generalized to automatically detect stronger contracts as described in [ECGN99].

To limit uncertainty the planning strategy currently assumes that an object can only assign to its own attributes. This assumption is true in many cases, but not all. A more sophisticated approach could weight transitions caused by actions. In practice routines change very little compared to the complete state. A possible weighting strategy is: the more attributes and the further away those attributes, the more a transition costs. A planner can then look for the cheapest instead for the shortest path to a goal state. Such an approach could remove the assumption that a routine only changes its own attributes, without increasing uncertainty too much. The planner we use currently, does not support actions with arguments. Eiffel routines with arguments could be supported by emulating arguments as global variables. Alternatively a different planner or model checker that more closely supports the Eiffel concepts could be used.

We expect contract based testing to supplement traditional unit testing. Good integration of contract based testing and unit testing would be very desirable. TestStudio could be extended to extracting minimal counter examples [Zel99] based on failed test cases. It could then ask the user if the counter example should be added to the existing unit test suite automatically.

We are certain that the idea of contract based testing will help programmers to detect bugs better and earlier. Even though TestStudio is relatively new and much remains to be done, the results already look very promising.

Appendix A

Listings

A.1 TestStudio Project File for Class *LINKED_LIST*

Listing A.1: TestStudio Project File For Class *LINKED_LIST*

```
<?xml version="1.0" encoding="ISO8859-1" ?>
<project_settings>
  <project_name> xxx </project_name>
  <project_dir> /tmp/ </project_dir>
  <ace_file> /home/aleitner/src/eclipse_workspace/test_studio/test_libraries/librarybase.ace </
      ace_file>
  <test_scope>
    <cluster_selection>
      <cluster> list </cluster>
    </cluster_selection>
    <class_selection>
      <cluster>
        <name> list </name>
        <classes>
          <class_name> LINKED_LIST </class_name>
        </classes>
      </cluster>
    </class_selection>
    <feature_selection>
      <class>
        <name> LINKED_LIST </name>
        <features>
          <feature_name> item </feature_name>
          <feature_name> first </feature_name>
          <feature_name> valid_cursor </feature_name>
          <feature_name> go_to </feature_name>
          <feature_name> replace </feature_name>
          <feature_name> remove </feature_name>
          <feature_name> make (creation) </feature_name>
          <feature_name> last </feature_name>
          <feature_name> index </feature_name>
          <feature_name> cursor </feature_name>
          <feature_name> count </feature_name>
          <feature_name> readable </feature_name>
          <feature_name> after </feature_name>
          <feature_name> before </feature_name>
          <feature_name> off </feature_name>
          <feature_name> isfirst </feature_name>
          <feature_name> islast </feature_name>
          <feature_name> full </feature_name>
          <feature_name> is_inserted </feature_name>
          <feature_name> start </feature_name>
          <feature_name> finish </feature_name>
          <feature_name> forth </feature_name>
          <feature_name> back </feature_name>
          <feature_name> move </feature_name>
          <feature_name> go_i_th </feature_name>
          <feature_name> put_front </feature_name>
          <feature_name> extend </feature_name>
          <feature_name> put_left </feature_name>
          <feature_name> put_right </feature_name>
          <feature_name> merge_left </feature_name>
          <feature_name> merge_right </feature_name>
          <feature_name> remove_left </feature_name>
          <feature_name> remove_right </feature_name>
```

```
            <feature_name> wipe_out </feature_name>
56          <feature_name> copy </feature_name>
            <feature_name> is_equal </feature_name>
            <feature_name> has </feature_name>
59          <feature_name> index_of </feature_name>
            <feature_name> i_th </feature_name>
            <feature_name> infix "@" </feature_name>
62          <feature_name> occurrences </feature_name>
            <feature_name> index_set </feature_name>
            <feature_name> valid_index </feature_name>
65          <feature_name> valid_cursor_index </feature_name>
            <feature_name> put </feature_name>
            <feature_name> put_i_th </feature_name>
68          <feature_name> swap </feature_name>
            <feature_name> duplicate </feature_name>
            <feature_name> writable </feature_name>
71          <feature_name> extendible </feature_name>
            <feature_name> prunable </feature_name>
            <feature_name> fill </feature_name>
74          <feature_name> prune </feature_name>
            <feature_name> prune_all </feature_name>
            <feature_name> is_empty </feature_name>
77          <feature_name> empty </feature_name>
            <feature_name> object_comparison </feature_name>
            <feature_name> changeable_comparison_criterion </feature_name>
80          <feature_name> compare_objects </feature_name>
            <feature_name> compare_references </feature_name>
            <feature_name> linear_representation </feature_name>
83          <feature_name> generator </feature_name>
            <feature_name> generating_type </feature_name>
            <feature_name> conforms_to </feature_name>
86          <feature_name> same_type </feature_name>
            <feature_name> consistent </feature_name>
            <feature_name> standard_is_equal </feature_name>
89          <feature_name> equal </feature_name>
            <feature_name> standard_equal </feature_name>
            <feature_name> deep_equal </feature_name>
92          <feature_name> standard_copy </feature_name>
            <feature_name> clone </feature_name>
            <feature_name> standard_clone </feature_name>
95          <feature_name> standard_twin </feature_name>
            <feature_name> deep_clone </feature_name>
            <feature_name> deep_copy </feature_name>
98          <feature_name> setup </feature_name>
            <feature_name> io </feature_name>
            <feature_name> out </feature_name>
101         <feature_name> tagged_out </feature_name>
            <feature_name> print </feature_name>
            <feature_name> Operating_environment </feature_name>
104         <feature_name> default_rescue </feature_name>
            <feature_name> default_create </feature_name>
            <feature_name> do_nothing </feature_name>
107         <feature_name> default </feature_name>
            <feature_name> default_pointer </feature_name>
            <feature_name> Void </feature_name>
110         <feature_name> force </feature_name>
            <feature_name> append </feature_name>
            <feature_name> search </feature_name>
113         <feature_name> exhausted </feature_name>
            <feature_name> do_all </feature_name>
            <feature_name> do_if </feature_name>
116         <feature_name> there_exists </feature_name>
            <feature_name> for_all </feature_name>
            <feature_name> sequential_occurrences </feature_name>
119       </features>
        </class>
      </feature_selection>
122   </test_scope>
      <test_data>
        <user_defined_values>
125       <basic_types>
            <integer>
              <value> -1 </value>
128           <value> 0 </value>
              <value> 1 </value>
              <value> 2 </value>
131           <value> -2 </value>
              <value> 10 </value>
              <value> -10 </value>
134           <value> 100 </value>
              <value> -100 </value>
              <value> 2147483647 </value>
137           <value> -2147483648 </value>
            </integer>
            <real>
140           <value> -1 </value>
              <value> 0 </value>
              <value> 1 </value>
```

```
143           <value> 2 </value>
              <value> -2 </value>
              <value> 10 </value>
146           <value> -10 </value>
              <value> 100 </value>
              <value> -100 </value>
149           <value> 3.40282e+38 </value>
              <value> 1.17549e-38 </value>
              <value> 1.19209e-07 </value>
152         </real>
            <double>
              <value> -1 </value>
155           <value> 0 </value>
              <value> 1 </value>
              <value> 2 </value>
158           <value> -2 </value>
              <value> 3.1415926535897931 </value>
              <value> -2.7182818284590451 </value>
161           <value> 1.7976931348623157e+308 </value>
              <value> 2.2250738585072014e-308 </value>
              <value> 2.2204460492503131e-16 </value>
164         </double>
            <boolean>
              <value> True </value>
167           <value> False </value>
            </boolean>
            <character> 0...64, 91...96, 123...255, 'a', 'Z' </character>
170         <string> "a...z", Void, "", "/01/.../255/", "x" </string>
          </basic_types>
          <other_values>
173       </other_values>
        </user_defined_values>
        <stress_level>
176       <global> 3 </global>
          <cluster_level>
          </cluster_level>
179       <class_level>
          </class_level>
          <feature_level>
182       </feature_level>
          <parameters>
            <stress_level>
185           <level> 1 </level>
              <number_of_method_calls> 10 </number_of_method_calls>
              <test_in_descendants> False </test_in_descendants>
188         </stress_level>
            <stress_level>
              <level> 2 </level>
191           <number_of_method_calls> 20 </number_of_method_calls>
              <test_in_descendants> False </test_in_descendants>
            </stress_level>
194         <stress_level>
              <level> 3 </level>
              <number_of_method_calls> 30 </number_of_method_calls>
197           <test_in_descendants> False </test_in_descendants>
            </stress_level>
            <stress_level>
200           <level> 4 </level>
              <number_of_method_calls> 50 </number_of_method_calls>
              <test_in_descendants> True </test_in_descendants>
203         </stress_level>
            <stress_level>
              <level> 5 </level>
206           <number_of_method_calls> 100 </number_of_method_calls>
              <test_in_descendants> True </test_in_descendants>
            </stress_level>
209       </parameters>
        </stress_level>
      </test_data>
212   <test_options>
        <assertions>
          <preconditions> True </preconditions>
215       <postconditions> True </postconditions>
          <class_invariants> True </class_invariants>
          <loop_variants_and_invariants> True </loop_variants_and_invariants>
218       <check_instructions> True </check_instructions>
        </assertions>
        <testing_order> 1 </testing_order>
221     <genericity_support_level> 1 </genericity_support_level>
        <output_directory> /tmp/ </output_directory>
      </test_options>
224 </project_settings>
```

A.2 Test Result Summary for Class *LINKED_LIST*

Table A.1: Test Result Summary for class *LINKED_LIST*

class	feature	status	#pass	#fail	#inv	#bad resp.
LINKED_LIST	make	failed	3696	1	0	0
LINKED_LIST	item	no valid calls	0	0	23	0
LINKED_LIST	first	passed	3	0	35	0
LINKED_LIST	last	no valid calls	0	0	29	0
LINKED_LIST	index	passed	51	0	0	0
LINKED_LIST	cursor	passed	57	0	0	0
LINKED_LIST	count	passed	48	0	0	0
LINKED_LIST	readable	passed	59	0	0	0
LINKED_LIST	after	passed	54	0	0	0
LINKED_LIST	before	passed	53	0	0	0
LINKED_LIST	off	passed	46	0	0	0
LINKED_LIST	isfirst	passed	49	0	0	0
LINKED_LIST	islast	passed	47	0	0	0
LINKED_LIST	valid_cursor	passed	50	0	0	0
LINKED_LIST	full	passed	56	0	0	0
LINKED_LIST	is_inserted	failed	0	39	0	0
LINKED_LIST	start	passed	50	0	0	0
LINKED_LIST	finish	passed	44	0	0	0
LINKED_LIST	forth	passed	42	0	11	0
LINKED_LIST	back	passed	2	0	29	0
LINKED_LIST	move	passed	40	0	0	0
LINKED_LIST	go_i_th	passed	7	0	29	0
LINKED_LIST	go_to	passed	27	0	19	0
LINKED_LIST	put_front	passed	61	0	0	0
LINKED_LIST	extend	passed	46	0	0	0
LINKED_LIST	put_left	passed	3	0	25	0
LINKED_LIST	put_right	passed	49	0	3	0
LINKED_LIST	replace	no valid calls	0	0	35	0
LINKED_LIST	merge_left	passed	3	0	28	0
LINKED_LIST	merge_right	passed	47	0	7	0
LINKED_LIST	remove	no valid calls	0	0	35	0
LINKED_LIST	remove_left	failed	0	2	28	0
LINKED_LIST	remove_right	no valid calls	0	0	34	0
LINKED_LIST	wipe_out	passed	47	0	0	0
LINKED_LIST	copy	passed	48	0	0	0
LINKED_LIST	is_equal	passed	40	0	0	0
LINKED_LIST	has	passed	47	0	0	0
LINKED_LIST	index_of	passed	29	0	31	0
LINKED_LIST	i_th	no valid calls	0	0	35	0
LINKED_LIST	infix @	no calls	0	0	0	0
LINKED_LIST	occurrences	passed	45	0	0	0
LINKED_LIST	index_set	passed	54	0	0	0
LINKED_LIST	valid_index	passed	47	0	0	0
LINKED_LIST	valid_cursor_index	passed	53	0	0	0
LINKED_LIST	put	failed	1	28	0	0
LINKED_LIST	put_i_th	no valid calls	0	0	29	0
LINKED_LIST	append	passed	57	0	1	0
LINKED_LIST	fill	failed	47	0	0	2
LINKED_LIST	swap	no valid calls	0	0	31	0
LINKED_LIST	duplicate	passed	2	0	31	0
LINKED_LIST	writable	passed	50	0	0	0

Table A.1: Test Result Summary for class *LINKED_LIST*

class	feature	status	#pass	#fail	#inv	#bad resp.
LINKED_LIST	extendible	passed	57	0	0	0
LINKED_LIST	prunable	passed	48	0	0	0
LINKED_LIST	prune	passed	53	0	0	0
LINKED_LIST	prune_all	passed	51	0	0	0
LINKED_LIST	is_empty	passed	47	0	0	0
LINKED_LIST	empty	passed	50	0	0	0
LINKED_LIST	object_comparison	passed	52	0	0	0
LINKED_LIST	changeable_comparison_criterion	passed	54	0	0	0
LINKED_LIST	compare_objects	passed	48	0	0	0
LINKED_LIST	compare_references	passed	57	0	0	0
LINKED_LIST	linear_representation	passed	50	0	0	0
LINKED_LIST	generator	passed	50	0	0	0
LINKED_LIST	generating_type	passed	40	0	0	0
LINKED_LIST	conforms_to	passed	55	0	0	0
LINKED_LIST	same_type	passed	49	0	2	0
LINKED_LIST	standard_is_equal	passed	41	0	0	0
LINKED_LIST	equal	failed	49	0	0	2
LINKED_LIST	standard_equal	passed	52	0	0	0
LINKED_LIST	deep_equal	passed	52	0	0	0
LINKED_LIST	standard_copy	passed	49	0	2	0
LINKED_LIST	clone	passed	46	0	0	0
LINKED_LIST	standard_clone	passed	50	0	0	0
LINKED_LIST	standard_twin	passed	49	0	0	0
LINKED_LIST	deep_clone	passed	45	0	0	0
LINKED_LIST	deep_copy	failed	48	2	0	0
LINKED_LIST	io	passed	57	0	0	0
LINKED_LIST	out	passed	47	0	0	0
LINKED_LIST	tagged_out	passed	50	0	0	0
LINKED_LIST	print	passed	45	0	0	0
LINKED_LIST	Operating_environment	passed	40	0	0	0
LINKED_LIST	default_rescue	passed	59	0	0	0
LINKED_LIST	do_nothing	passed	59	0	0	0
LINKED_LIST	default	passed	45	0	0	0
LINKED_LIST	default_pointer	no calls	0	0	0	0
LINKED_LIST	force	passed	50	0	0	0
LINKED_LIST	search	passed	49	0	0	0
LINKED_LIST	exhausted	passed	53	0	0	0
LINKED_LIST	do_all	no valid calls	0	0	25	0
LINKED_LIST	do_if	no valid calls	0	0	34	0
LINKED_LIST	there_exists	no valid calls	0	0	31	0
LINKED_LIST	for_all	no valid calls	0	0	35	0
LINKED_LIST	sequential_occurrences	passed	48	0	0	0
STRING	make_filled	failed	15	3	21	0
STRING	prune	passed	1	0	0	0
LINKED_LIST	twin	passed	22	0	0	0
STRING	make	failed	21	2	17	0
STRING	make_empty	passed	47	0	0	0
STRING	string	passed	2	0	0	0
LINKED_LIST	deep_twin	passed	19	0	0	0
CURSOR	is_equal	passed	1	0	0	0
CURSOR	standard_twin	passed	1	0	0	0
CURSOR	conforms_to	passed	2	0	0	0
CURSOR	generator	passed	1	0	0	0
CURSOR	deep_equal	passed	1	0	0	0

Table A.1: Test Result Summary for class *LINKED_LIST*

class	feature	status	#pass	#fail	#inv	#bad resp.
CURSOR	deep_copy	passed	1	0	0	0
CURSOR	twin	passed	1	0	0	0
CURSOR	deep_twin	passed	2	0	0	0
CURSOR	deep_clone	passed	2	0	0	0
STRING	make_from_string	passed	39	0	2	0
STRING	append_double	passed	1	0	0	0
STRING	resize	no valid calls	0	0	1	0
CURSOR	equal	failed	0	0	0	1
STRING	out	passed	1	0	0	0
STRING	has	passed	2	0	0	0
STRING	fill	no valid calls	0	0	2	0
STRING	is_equal	passed	1	0	1	0
STRING	fill_blank	passed	1	0	0	0
STRING	remove_tail	passed	1	0	1	0
CURSOR	standard_clone	passed	1	0	0	0
STRING	equal	passed	1	0	0	0
STRING	prune_all	passed	1	0	0	0
INTEGER_INTERVAL	make	passed	13	0	0	0
INTEGER_INTERVAL	fill	no valid calls	0	0	1	0
STRING	standard_equal	passed	1	0	0	0
STRING	replace_substring	no valid calls	0	0	2	0
STRING	extend	passed	1	0	0	0
STRING	right_adjust	passed	1	0	0	0
CURSOR	standard_equal	passed	1	0	0	0
STRING	is_inserted	passed	1	0	0	0
CURSOR	default_rescue	passed	1	0	0	0
INTEGER_INTERVAL	is_inserted	passed	1	0	0	0
STD_FILES	same_type	passed	1	0	0	0
INTEGER_INTERVAL	full	passed	1	0	0	0
INTEGER_INTERVAL	twin	passed	1	0	0	0
STRING	item	no valid calls	0	0	1	0
STRING	capacity	passed	1	0	0	0
CURSOR	default	passed	1	0	0	0
STRING	prepend_boolean	passed	1	0	0	0
STRING	default	passed	2	0	0	0
STRING	True_constant	passed	1	0	0	0
STRING	subcopy	no valid calls	0	0	3	0
INTEGER_INTERVAL	additional_space	passed	2	0	0	0
STRING	adapt_size	passed	1	0	0	0
CURSOR	standard_copy	passed	1	0	1	0
INTEGER_INTERVAL	exists1	passed	1	0	0	0
INTEGER_INTERVAL	lower_defined	passed	1	0	0	0
STRING	left_adjust	passed	1	0	0	0
STRING	remake	no valid calls	0	0	1	0
INTEGER_INTERVAL	object_comparison	passed	1	0	0	0
STRING	adapt	passed	2	0	0	0
INTEGER_INTERVAL	clone	passed	1	0	0	0
INTEGER_INTERVAL	deep_twin	passed	1	0	0	0
STRING	index_set	passed	3	0	0	0
STRING	valid_index	passed	1	0	0	0
INTEGER_INTERVAL	lower	passed	1	0	0	0
STRING	False_constant	passed	1	0	0	0
STRING	insert_string	passed	1	0	1	0
STRING	do_nothing	passed	1	0	0	0

Table A.1: Test Result Summary for class *LINKED_LIST*

class	feature	status	#pass	#fail	#inv	#bad resp.
STRING	deep_copy	passed	1	0	0	0
STRING	deep_equal	passed	2	0	0	0
STRING	is_empty	passed	1	0	0	0
STRING	linear_representation	passed	1	0	0	0
STRING	fuzzy_index	no valid calls	0	0	3	0
STRING	object_comparison	passed	3	0	0	0
STRING	mirror	passed	1	0	0	0
STRING	generating_type	passed	3	0	0	0
STRING	remove_substring	no valid calls	0	0	2	0
STRING	standard_twin	passed	2	0	0	0
STRING	append	passed	1	0	0	0
STD_FILES	lastchar	passed	1	0	0	0
STRING	default_rescue	passed	1	0	0	0
STRING	extendible	passed	1	0	0	0
STRING	Growth_percentage	passed	1	0	0	0
STRING	append_integer	passed	1	0	0	0
STRING	tail	no valid calls	0	0	1	0
STRING	compare_objects	no valid calls	0	0	2	0
STRING	prune_all_trailing	passed	2	0	0	0
STRING	to_boolean	no valid calls	0	0	1	0
STRING	standard_is_equal	passed	1	0	0	0
STRING	to_upper	passed	1	0	0	0
STRING	keep_head	passed	1	0	0	0
INTEGER_INTERVAL	adapt	passed	1	0	0	0
STRING	head	no valid calls	0	0	1	0
STRING	mirrored	passed	1	0	0	0
STRING	conforms_to	passed	1	0	0	0
STRING	to_lower	passed	1	0	0	0
STRING	twin	passed	1	0	0	0
STRING	grow	passed	1	0	0	0
STRING	to_integer_64	no valid calls	0	0	1	0
STRING	deep_twin	passed	1	0	0	0
STRING	clone	passed	1	0	0	0
STRING	replace_blank	passed	1	0	0	0
STD_FILES	error	passed	1	0	0	0
STRING	min	passed	1	0	0	0
STRING	fill_character	passed	1	0	0	0

A.3 Interface of *LINKED_LIST*

Listing A.2: Interface of class *LINKED_LIST*

```
indexing
    description: "Sequential, one-way linked lists"
    status: "See notice at end of class"
    names: linked_list, sequence
    representation: linked
    access: index, cursor, membership
    contents: generic
    date: "$Date: 2003/08/19 01:00:10 $"
    revision: "$Revision: 1.24 $"

class interface
  LINKED_LIST [G]

create

  make
      -- Create an empty list.
    ensure
      is_before: before

feature -- Initialization

    make
      -- Create an empty list.
    ensure
      is_before: before

feature -- Access

    cursor: CURSOR
      -- Current cursor position
    ensure -- from CURSOR_STRUCTURE
      cursor_not_void: Result /= Void

    first: like item
      -- Item at first position
    require -- from CHAIN
      not_empty: not is_empty

  generating_type: STRING
      -- Name of current object's generating type
      -- (type of which it is a direct instance)
      -- (from ANY)

    generator: STRING
      -- Name of current object's generating class
      -- (base class of the type of which it is a direct instance)
      -- (from ANY)

    has (v: like item): BOOLEAN
      -- Does chain include `v'?
      -- (Reference or object equality,
      -- based on `object_comparison'.)
      -- (from CHAIN)
    ensure -- from CONTAINER
      not_found_in_empty: Result implies not is_empty

  i_th (i: INTEGER): like item
      -- Item at `i'-th position
      -- Was declared in CHAIN as synonym of `@'.
      -- (from CHAIN)
    require -- from TABLE
      valid_key: valid_index (k)

    index: INTEGER
      -- Index of current position

    index_of (v: like item; i: INTEGER): INTEGER
      -- Index of `i'-th occurrence of item identical to `v'.
      -- (Reference or object equality,
      -- based on `object_comparison'.)
      -- 0 if none.
      -- (from CHAIN)
    require -- from LINEAR
      positive_occurrences: i > 0
    ensure -- from LINEAR
      non_negative_result: Result >= 0

  item: G
      -- Current item
    require -- from TRAVERSABLE
```

```
82          not_off: not off
        require -- from ACTIVE
          readable: readable
85
    last: like item
        -- Item at last position
88      require -- from CHAIN
          not_empty: not is_empty

91  sequential_occurrences (v: G): INTEGER
        -- Number of times 'v' appears.
        -- (Reference or object equality,
94      -- based on 'object_comparison'.)
        -- (from LINEAR)
      ensure -- from BAG
97      non_negative_occurrences: Result >= 0

    infix "@" (i: INTEGER): like item
100     -- Item at 'i'-th position
        -- Was declared in CHAIN as synonym of 'i_th'.
        -- (from CHAIN)
103   require -- from TABLE
        valid_key: valid_index (k)

106 feature -- Measurement

    count: INTEGER
109     -- Number of items

    index_set: INTEGER_INTERVAL
112     -- Range of acceptable indexes
        -- (from CHAIN)
      ensure -- from INDEXABLE
115     not_void: Result /= Void
      ensure then -- from CHAIN
        count_definition: Result.count = count
118
    occurrences (v: like item): INTEGER
        -- Number of times 'v' appears.
121     -- (Reference or object equality,
        -- based on 'object_comparison'.)
        -- (from CHAIN)
124   ensure -- from BAG
        non_negative_occurrences: Result >= 0

127 occurrences (v: like item): INTEGER
        -- Number of times 'v' appears.
        -- (Reference or object equality,
130     -- based on 'object_comparison'.)
        -- (from CHAIN)
      ensure -- from BAG
133     non_negative_occurrences: Result >= 0

  feature -- Comparison
136
    frozen deep_equal (some: ANY; other: like some): BOOLEAN
        -- Are 'some' and 'other' either both void
139     -- or attached to isomorphic object structures?
        -- (from ANY)
      ensure -- from ANY
142     shallow_implies_deep: standard_equal (some, other) implies Result
        both_or_none_void: (some = Void) implies (Result = (other = Void))
        same_type: (Result and (some /= Void)) implies some.same_type (other)
145     symmetric: Result implies deep_equal (other, some)

    frozen equal (some: ANY; other: like some): BOOLEAN
148     -- Are 'some' and 'other' either both void or attached
        -- to objects considered equal?
        -- (from ANY)
151   ensure -- from ANY
        definition: Result = (some = Void and other = Void) or else ((some /= Void and other /=
            Void) and then some.is_equal (other))

154 is_equal (other: like Current): BOOLEAN
        -- Does 'other' contain the same elements?
        -- (from LIST)
157   require -- from ANY
        other_not_void: other /= Void
      ensure -- from ANY
160     symmetric: Result implies other.is_equal (Current)
        consistent: standard_is_equal (other) implies Result
      ensure then -- from LIST
163     indices_unchanged: index = old index and other.index = old other.index
        true_implies_same_size: Result implies count = other.count

166 frozen standard_equal (some: ANY; other: like some): BOOLEAN
        -- Are 'some' and 'other' either both void or attached to
        -- field-by-field identical objects of the same type?
```

```
169         -- Always uses default object comparison criterion.
            -- (from ANY)
          ensure -- from ANY
172         definition: Result = (some = Void and other = Void) or else ((some /= Void and other /=
                Void) and then some.standard_is_equal (other))

      frozen standard_is_equal (other: like Current): BOOLEAN
175         -- Is 'other' attached to an object of the same type
            -- as current object, and field-by-field identical to it?
            -- (from ANY)
178       require -- from ANY
            other_not_void: other /= Void
          ensure -- from ANY
181         same_type: Result implies same_type (other)
            symmetric: Result implies other.standard_is_equal (Current)

184   feature -- Status report

      after: BOOLEAN
187         -- Is there no valid cursor position to the right of cursor?

      before: BOOLEAN
190         -- Is there no valid cursor position to the left of cursor?

      changeable_comparison_criterion: BOOLEAN
193         -- May 'object_comparison' be changed?
            -- (Answer: yes by default.)
            -- (from CONTAINER)
196
      conforms_to (other: ANY): BOOLEAN
            -- Does type of current object conform to type
199         -- of 'other' (as per Eiffel: The Language, chapter 13)?
            -- (from ANY)
          require -- from ANY
202         other_not_void: other /= Void

      exhausted: BOOLEAN
205         -- Has structure been completely explored?
            -- (from LINEAR)
          ensure -- from LINEAR
208         exhausted_when_off: off implies Result

      Extendible: BOOLEAN is True
211         -- May new items be added? (Answer: yes.)
            -- (from DYNAMIC_CHAIN)

214   Full: BOOLEAN is False
            -- Is structured filled to capacity? (Answer: no.)

217   is_empty: BOOLEAN
            -- Is structure empty?
            -- (from FINITE)
220
      is_inserted (v: G): BOOLEAN
            -- Has 'v' been inserted at the end by the most recent 'put' or
223         -- 'extend'?

      isfirst: BOOLEAN
226         -- Is cursor at first position?
          ensure -- from CHAIN
            valid_position: Result implies not is_empty
229
      islast: BOOLEAN
            -- Is cursor at last position?
232       ensure -- from CHAIN
            valid_position: Result implies not is_empty

235   object_comparison: BOOLEAN
            -- Must search operations use 'equal' rather than '='
            -- for comparing references? (Default: no, use '='.)
238         -- (from CONTAINER)

      off: BOOLEAN
241         -- Is there no current item?

      prunable: BOOLEAN
244         -- May items be removed? (Answer: yes.)
            -- (from DYNAMIC_CHAIN)

247   readable: BOOLEAN
            -- Is there a current item that may be read?

250   same_type (other: ANY): BOOLEAN
            -- Is type of current object identical to type of 'other'?
            -- (from ANY)
253       require -- from ANY
            other_not_void: other /= Void
          ensure -- from ANY
```

```
256         definition: Result = (conforms_to (other) and other.conforms_to (Current))

      valid_cursor (p: CURSOR): BOOLEAN
259         -- Can the cursor be moved to position 'p'?

      valid_cursor_index (i: INTEGER): BOOLEAN
262         -- Is 'i' correctly bounded for cursor movement?
            -- (from CHAIN)
          ensure -- from CHAIN
265         valid_cursor_index_definition: Result = ((i >= 0) and (i <= count + 1))

      valid_index (i: INTEGER): BOOLEAN
268         -- Is 'i' within allowable bounds?
            -- (from CHAIN)
          ensure then -- from INDEXABLE
271         only_if_in_index_set: Result implies ((i >= index_set.lower) and (i <= index_set.upper))
          ensure then -- from CHAIN
            valid_index_definition: Result = ((i >= 1) and (i <= count))
274
      writable: BOOLEAN
            -- Is there a current item that may be modified?
277         -- (from SEQUENCE)

    feature -- Status setting
280
      compare_objects
            -- Ensure that future search operations will use 'equal'
283         -- rather than '=' for comparing references.
            -- (from CONTAINER)
          require -- from CONTAINER
286         changeable_comparison_criterion: changeable_comparison_criterion
          ensure -- from CONTAINER
            object_comparison
289
      compare_references
            -- Ensure that future search operations will use '='
292         -- rather than 'equal' for comparing references.
            -- (from CONTAINER)
          require -- from CONTAINER
295         changeable_comparison_criterion: changeable_comparison_criterion
          ensure -- from CONTAINER
            reference_comparison: not object_comparison
298
    feature -- Cursor movement

301   back
            -- Move to previous item.
          require -- from BILINEAR
304         not_before: not before

      finish
307         -- Move cursor to last position.
            -- (Go before if empty)
          ensure then -- from CHAIN
310         at_last: not is_empty implies islast
          ensure then
            empty_convention: is_empty implies before
313
      forth
            -- Move cursor to next position.
316       require -- from LINEAR
            not_after: not after
          ensure then -- from LIST
319         moved_forth: index = old index + 1

      go_i_th (i: INTEGER)
322         -- Move cursor to 'i'-th position.
          require -- from CHAIN
            valid_cursor_index: valid_cursor_index (i)
325       ensure -- from CHAIN
            position_expected: index = i

328   go_to (p: CURSOR)
            -- Move cursor to position 'p'.
          require -- from CURSOR_STRUCTURE
331         cursor_position_valid: valid_cursor (p)

      move (i: INTEGER)
334         -- Move cursor 'i' positions. The cursor
            -- may end up 'off' if the offset is too big.
          ensure -- from CHAIN
337         too_far_right: (old index + i > count) implies exhausted
            too_far_left: (old index + i < 1) implies exhausted
            expected_index: (not exhausted) implies (index = old index + i)
340       ensure then
            moved_if_inbounds: ((old index + i) >= 0 and (old index + i) <= (count + 1)) implies index
                  = (old index + i)
            before_set: (old index + i) <= 0 implies before
```

```
343         after_set: (old index + i) >= (count + 1) implies after

      search (v: like item)
346         -- Move to first position (at or after current
            -- position) where 'item' and 'v' are equal.
            -- If structure does not include 'v' ensure that
349         -- 'exhausted' will be true.
            -- (Reference or object equality,
            -- based on 'object_comparison'.)
352         -- (from BILINEAR)
          ensure -- from LINEAR
            object_found: (not exhausted and object_comparison) implies equal (v, item)
355         item_found: (not exhausted and not object_comparison) implies v = item

      start
358         -- Move cursor to first position.
          ensure then -- from CHAIN
            at_first: not is_empty implies isfirst
361       ensure then
            empty_convention: is_empty implies after

364 feature -- Element change

      append (s: SEQUENCE [G])
367         -- Append a copy of 's'.
            -- (from CHAIN)
          require -- from SEQUENCE
370         argument_not_void: s /= Void
          ensure -- from SEQUENCE
            new_count: count >= old count
373
      extend (v: like item)
            -- Add 'v' to end.
376         -- Do not move cursor.
          require -- from COLLECTION
            extendible: extendible
379       ensure -- from COLLECTION
            item_inserted: is_inserted (v)
          ensure then -- from BAG
382         one_more_occurrence: occurrences (v) = old (occurrences (v)) + 1

      fill (other: CONTAINER [G])
385         -- Fill with as many items of 'other' as possible.
            -- The representations of 'other' and current structure
            -- need not be the same.
388         -- (from CHAIN)
          require -- from COLLECTION
            other_not_void: other /= Void
391         extendible: extendible

      force (v: like item)
394         -- Add 'v' to end.
            -- (from SEQUENCE)
          require -- from SEQUENCE
397         extendible: extendible
          ensure then -- from SEQUENCE
            new_count: count = old count + 1
400         item_inserted: has (v)

      merge_left (other: like Current)
403         -- Merge 'other' into current structure before cursor
            -- position. Do not move cursor. Empty 'other'.
          require -- from DYNAMIC_CHAIN
406         extendible: extendible
            not_before: not before
            other_exists: other /= Void
409         not_current: other /= Current
          ensure -- from DYNAMIC_CHAIN
            new_count: count = old count + old other.count
412         new_index: index = old index + old other.count
            other_is_empty: other.is_empty

415   merge_right (other: like Current)
            -- Merge 'other' into current structure after cursor
            -- position. Do not move cursor. Empty 'other'.
418       require -- from DYNAMIC_CHAIN
            extendible: extendible
            not_after: not after
421         other_exists: other /= Void
            not_current: other /= Current
          ensure -- from DYNAMIC_CHAIN
424         new_count: count = old count + old other.count
            same_index: index = old index
            other_is_empty: other.is_empty
427
      put (v: like item)
            -- Replace current item by 'v'.
430         -- (Synonym for 'replace')
```

```
        -- (from CHAIN)
      require -- from COLLECTION
        extendible: extendible
      ensure -- from COLLECTION
        item_inserted: is_inserted (v)
      ensure then -- from CHAIN
        same_count: count = old count

  put_front (v: like item)
      -- Add 'v' to beginning.
      -- Do not move cursor.
    ensure -- from DYNAMIC_CHAIN
      new_count: count = old count + 1
      item_inserted: first = v

  put_i_th (v: like item; i: INTEGER)
      -- Put 'v' at 'i'-th position.
      -- (from CHAIN)
    require -- from TABLE
      valid_key: valid_index (k)
    ensure then -- from INDEXABLE
      insertion_done: i_th (k) = v

  put_left (v: like item)
      -- Add 'v' to the left of cursor position.
      -- Do not move cursor.
    require -- from DYNAMIC_CHAIN
      extendible: extendible
      not_before: not before
    ensure -- from DYNAMIC_CHAIN
      new_count: count = old count + 1
      new_index: index = old index + 1
    ensure then
      previous_exists: previous /= Void
      item_inserted: previous.item = v

  put_right (v: like item)
      -- Add 'v' to the right of cursor position.
      -- Do not move cursor.
    require -- from DYNAMIC_CHAIN
      extendible: extendible
      not_after: not after
    ensure -- from DYNAMIC_CHAIN
      new_count: count = old count + 1
      same_index: index = old index
    ensure then
      next_exists: next /= Void
      item_inserted: not old before implies next.item = v
      item_inserted_before: old before implies active.item = v

  replace (v: like item)
      -- Replace current item by 'v'.
    require -- from ACTIVE
      writable: writable
    ensure -- from ACTIVE
      item_replaced: item = v

feature -- Removal

  prune (v: like item)
      -- Remove first occurrence of 'v', if any,
      -- after cursor position.
      -- If found, move cursor to right neighbor;
      -- if not, make structure 'exhausted'.
      -- (from DYNAMIC_CHAIN)
    require -- from COLLECTION
      prunable: prunable

  prune_all (v: like item)
      -- Remove all occurrences of 'v'.
      -- (Reference or object equality,
      -- based on 'object_comparison'.)
      -- Leave structure 'exhausted'.
      -- (from DYNAMIC_CHAIN)
    require -- from COLLECTION
      prunable: prunable
    ensure -- from COLLECTION
      no_more_occurrences: not has (v)
    ensure then -- from DYNAMIC_CHAIN
      is_exhausted: exhausted

  remove
      -- Remove current item.
      -- Move cursor to right neighbor
      -- (or 'after' if no right neighbor).
    require -- from ACTIVE
      prunable: prunable
      writable: writable
```

```
      ensure then -- from DYNAMIC_LIST
        after_when_empty: is_empty implies after

    remove_left
        -- Remove item to the left of cursor position.
        -- Do not move cursor.
      require -- from DYNAMIC_CHAIN
        left_exists: index > 1
      require else -- from DYNAMIC_LIST
        not_before: not before
      ensure -- from DYNAMIC_CHAIN
        new_count: count = old count - 1
        new_index: index = old index - 1

    remove_right
        -- Remove item to the right of cursor position.
        -- Do not move cursor.
      require -- from DYNAMIC_CHAIN
        right_exists: index < count
      ensure -- from DYNAMIC_CHAIN
        new_count: count = old count - 1
        same_index: index = old index

    wipe_out
        -- Remove all items.
      require -- from COLLECTION
        prunable: prunable
      ensure -- from COLLECTION
        wiped_out: is_empty
      ensure then -- from DYNAMIC_LIST
        is_before: before

  feature -- Transformation

  swap (i: INTEGER)
        -- Exchange item at 'i'-th position with item
        -- at cursor position.
        -- (from CHAIN)
      require -- from CHAIN
        not_off: not off
        valid_index: valid_index (i)
      ensure -- from CHAIN
        swapped_to_item: item = old i_th (i)
        swapped_from_item: i_th (i) = old item

  feature -- Conversion

    linear_representation: LINEAR [G]
        -- Representation as a linear structure
        -- (from LINEAR)

  feature -- Duplication

    copy (other: like Current)
        -- Update current object using fields of object attached
        -- to 'other', so as to yield equal objects.
      require -- from ANY
        other_not_void: other /= Void
        type_identity: same_type (other)
      ensure -- from ANY
        is_equal: is_equal (other)

    frozen deep_copy (other: like Current)
        -- Effect equivalent to that of:
        --     'copy' ('other' . 'deep_twin')
        -- (from ANY)
      require -- from ANY
        other_not_void: other /= Void
      ensure -- from ANY
        deep_equal: deep_equal (Current, other)

    frozen deep_twin: like Current
        -- New object structure recursively duplicated from Current.
        -- (from ANY)
      ensure -- from ANY
        deep_equal: deep_equal (Current, Result)

    duplicate (n: INTEGER): like Current
        -- Copy of sub-chain beginning at current position
        -- and having min ('n', 'from_here') items,
        -- where 'from_here' is the number of items
        -- at or to the right of current position.
        -- (from DYNAMIC_CHAIN)
      require -- from CHAIN
        not_off_unless_after: off implies after
        valid_subchain: n >= 0

    frozen standard_copy (other: like Current)
```

```
607         -- Copy every field of 'other' onto corresponding field
            -- of current object.
            -- (from ANY)
610       require -- from ANY
            other_not_void: other /= Void
            type_identity: same_type (other)
613       ensure -- from ANY
            is_standard_equal: standard_is_equal (other)

616     frozen standard_twin: like Current
            -- New object field-by-field identical to 'other'.
            -- Always uses default copying semantics.
619         -- (from ANY)
          ensure -- from ANY
            standard_twin_not_void: Result /= Void
622         equal: standard_equal (Result, Current)

      frozen twin: like Current
625         -- New object equal to 'Current'
            -- 'twin' calls 'copy'; to change copying/twining semantics, redefine 'copy'.
            -- (from ANY)
628       ensure -- from ANY
            twin_not_void: Result /= Void
            is_equal: Result.is_equal (Current)
631
    feature -- Basic operations

634     frozen default: like Current
            -- Default value of object's type
            -- (from ANY)
637
      frozen default_pointer: POINTER
            -- Default value of type 'POINTER'
640         -- (Avoid the need to write 'p'.'default' for
            -- some 'p' of type 'POINTER'.)
            -- (from ANY)
643
      default_rescue
            -- Process exception for routines with no Rescue clause.
646         -- (Default: do nothing.)
            -- (from ANY)

649     frozen do_nothing
            -- Execute a null action.
            -- (from ANY)
652
    feature -- Iteration

655     do_all (action: PROCEDURE [ANY, TUPLE [G]])
            -- Apply 'action' to every item.
            -- Semantics not guaranteed if 'action' changes the structure;
658         -- in such a case, apply iterator to clone of structure instead.
            -- (from LINEAR)
          require -- from TRAVERSABLE
661         action_exists: action /= Void

      do_if (action: PROCEDURE [ANY, TUPLE [G]]; test: FUNCTION [ANY, TUPLE [G], BOOLEAN])
664         -- Apply 'action' to every item that satisfies 'test'.
            -- Semantics not guaranteed if 'action' or 'test' changes the structure;
            -- in such a case, apply iterator to clone of structure instead.
667         -- (from LINEAR)
          require -- from TRAVERSABLE
            action_exists: action /= Void
670         test_exits: test /= Void

      for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOLEAN
673         -- Is 'test' true for all items?
            -- (from LINEAR)
          require -- from TRAVERSABLE
676         test_exits: test /= Void
          ensure then -- from LINEAR
            empty: is_empty implies Result
679
      there_exists (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOLEAN
            -- Is 'test' true for at least one item?
682         -- (from LINEAR)
          require -- from TRAVERSABLE
            test_exits: test /= Void
685
    feature -- Output

688     io: STD_FILES
            -- Handle to standard file setup
            -- (from ANY)
691
      out: STRING
            -- New string containing terse printable representation
694         -- of current object
```

```
        -- Was declared in ANY as synonym of 'tagged_out'.
        -- (from ANY)

    print (some: ANY)
        -- Write terse external representation of 'some'
        -- on standard output.
        -- (from ANY)

    frozen tagged_out: STRING
        -- New string containing terse printable representation
        -- of current object
        -- Was declared in ANY as synonym of 'out'.
        -- (from ANY)

feature -- Platform

    operating_environment: OPERATING_ENVIRONMENT
        -- Objects available from the operating system
        -- (from ANY)

invariant

    prunable: prunable
    empty_constraint: is_empty implies ((first_element = Void) and (active = Void))
    not_void_unless_empty: (active = Void) implies is_empty
    before_constraint: before implies (active = first_element)
    after_constraint: after implies (active = last_element)
      -- from ANY
    reflexive_equality: standard_is_equal (Current)
    reflexive_conformance: conforms_to (Current)
      -- from ANY
    reflexive_equality: standard_is_equal (Current)
    reflexive_conformance: conforms_to (Current)
      -- from LIST
    before_definition: before = (index = 0)
    after_definition: after = (index = count + 1)
      -- from CHAIN
    non_negative_index: index >= 0
    index_small_enough: index <= count + 1
    off_definition: off = ((index = 0) or (index = count + 1))
    isfirst_definition: isfirst = ((not is_empty) and (index = 1))
    islast_definition: islast = ((not is_empty) and (index = count))
    item_corresponds_to_index: (not off) implies (item = i_th (index))
    index_set_has_same_count: index_set.count = count
      -- from ACTIVE
    writable_constraint: writable implies readable
    empty_constraint: is_empty implies (not readable) and (not writable)
      -- from INDEXABLE
    index_set_not_void: index_set /= Void
      -- from BILINEAR
    not_both: not (after and before)
    before_constraint: before implies off
      -- from LINEAR
    after_constraint: after implies off
      -- from TRAVERSABLE
    empty_constraint: is_empty implies off
      -- from FINITE
    empty_definition: is_empty = (count = 0)
    non_negative_count: count >= 0
      -- from DYNAMIC_CHAIN
    extendible: extendible

indexing
    library: "[
        EiffelBase: Library of reusable components for Eiffel.
    ]"
    status: "[
        Copyright 1986-2001 Interactive Software Engineering (ISE).
        For ISE customers the original versions are an ISE product
        covered by the ISE Eiffel license and support agreements.
    ]"
    license: "[
        EiffelBase may now be used by anyone as FREE SOFTWARE to
        develop any product, public-domain or commercial, without
        payment to ISE, under the terms of the ISE Free Eiffel Library
        License (IFELL) at http://eiffel.com/products/base/license.html.
    ]"
    source: "[
        Interactive Software Engineering Inc.
        ISE Building
        360 Storke Road, Goleta, CA 93117 USA
        Telephone 805-685-1006, Fax 805-685-6869
        Electronic mail <info@eiffel.com>
        Customer support http://support.eiffel.com
    ]"
    info: "[
        For latest info see award-winning pages: http://eiffel.com
    ]"
```

```
784 end -- class LINKED_LIST
```

A.4 Source Code of Class *COUNTER*

Listing A.3: Source code of class *COUNTER*

```
class
  COUNTER

create

  make

feature

  make is
    do
      i := 0
      j := 0
    ensure
      i = 0
      j = 0
    end

feature -- Attributes

  i: INTEGER
  j: INTEGER

feature -- Commands

  increase_i is
    do
      i := i + 1
    ensure
      i = old i + 1
    end

  decrease_i is
    do
      i := i - 1
    ensure
      i = old i - 1
    end

  increase_j is
    do
      j := j + 1
    ensure
      j = old j + 1
    end

  decrease_j is
    do
      j := j - 1
    ensure
      j = old j - 1
    end

feature -- Feature under test

  foo is
    require
      i = j - 2
      i > 0
    do
      print ("Hoorayy!!!%N")
    end

end
```

A.5 Initial Problem for *COUNTER.foo*

Listing A.4: Initial problem for *COUNTER.foo*

```
VARIABLES

    nat(2) arg_todo
    bool alive_0
    bool alive_1
    bool alive_2
    bool alive_3
    nat(2) type_0
    nat(2) type_1
    nat(2) type_2
    nat(2) type_3
    % variables for attribute or function without arguments: COUNTER.i
    nat(2) attr_1_2927_0
    nat(2) attr_1_2927_1
    nat(2) attr_1_2927_2
    nat(2) attr_1_2927_3
    % variables for attribute or function without arguments: COUNTER.j
    nat(2) attr_1_2928_0
    nat(2) attr_1_2928_1
    nat(2) attr_1_2928_2
    nat(2) attr_1_2928_3
    % variables for attribute or function without arguments: COUNTER.generator
    nat(2) attr_1_1_0
    nat(2) attr_1_1_1
    nat(2) attr_1_1_2
    nat(2) attr_1_1_3
    % variables for attribute or function without arguments: COUNTER.generating_type
    nat(2) attr_1_2_0
    nat(2) attr_1_2_1
    nat(2) attr_1_2_2
    nat(2) attr_1_2_3
    % variables for attribute or function without arguments: COUNTER.twin
    nat(2) attr_1_10_0
    nat(2) attr_1_10_1
    nat(2) attr_1_10_2
    nat(2) attr_1_10_3
    % variables for attribute or function without arguments: COUNTER.standard_twin
    nat(2) attr_1_15_0
    nat(2) attr_1_15_1
    nat(2) attr_1_15_2
    nat(2) attr_1_15_3
    % variables for attribute or function without arguments: COUNTER.deep_twin
    nat(2) attr_1_16_0
    nat(2) attr_1_16_1
    nat(2) attr_1_16_2
    nat(2) attr_1_16_3
    % variables for attribute or function without arguments: COUNTER.io
    nat(2) attr_1_20_0
    nat(2) attr_1_20_1
    nat(2) attr_1_20_2
    nat(2) attr_1_20_3
    % variables for attribute or function without arguments: COUNTER.out
    nat(2) attr_1_22_0
    nat(2) attr_1_22_1
    nat(2) attr_1_22_2
    nat(2) attr_1_22_3
    % variables for attribute or function without arguments: COUNTER.tagged_out
    nat(2) attr_1_21_0
    nat(2) attr_1_21_1
    nat(2) attr_1_21_2
    nat(2) attr_1_21_3
    % variables for attribute or function without arguments: COUNTER.Operating_environment
    nat(2) attr_1_24_0
    nat(2) attr_1_24_1
    nat(2) attr_1_24_2
    nat(2) attr_1_24_3
    % variables for attribute or function without arguments: COUNTER.default
    nat(2) attr_1_28_0
    nat(2) attr_1_28_1
    nat(2) attr_1_28_2
    nat(2) attr_1_28_3

  SYSTEM

  agt: interpreter
    % actions for creation procedure: COUNTER.make
    creat_1_2926_1
      mod:
        alive_1, type_1, attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1,
            attr_1_15_1, attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1,
            attr_1_28_1
```

```
80      pre:
          ~alive_1 /\ alive_0
        eff:
83        alive_1' /\ (type_1' = 1)  /\
  % postcondition
            (attr_1_2927_1' = 0) /\
86          (attr_1_2928_1' = 0) /\
          true

89
      creat_1_2926_2
        mod:
92        alive_2, type_2, attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2,
              attr_1_15_2, attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2,
              attr_1_28_2
        pre:
          ~alive_2 /\ alive_1
95      eff:
          alive_2' /\ (type_2' = 1)  /\
  % postcondition
98          (attr_1_2927_2' = 0) /\
            (attr_1_2928_2' = 0) /\
          true
101

      creat_1_2926_3
104     mod:
          alive_3, type_3, attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3,
              attr_1_15_3, attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3,
              attr_1_28_3
        pre:
107       ~alive_3 /\ alive_2
        eff:
          alive_3' /\ (type_3' = 1)  /\
110% postcondition
            (attr_1_2927_3' = 0) /\
            (attr_1_2928_3' = 0) /\
113       true


116   % actions for routine: COUNTER.make
      rout_1_2926_1
        mod:
119       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
        pre:
          alive_1 /\ (type_1 = 1)
122     eff:
          % postcondition
            (attr_1_2927_1' = 0) /\
125         (attr_1_2928_1' = 0) /\
          true
    /\
128% transitions
          true

131
      % actions for routine: COUNTER.make
      rout_1_2926_2
134     mod:
          attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
              attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
        pre:
137       alive_2 /\ (type_2 = 1)
        eff:
          % postcondition
140         (attr_1_2927_2' = 0) /\
            (attr_1_2928_2' = 0) /\
          true
143 /\
  % transitions
          true
146

      % actions for routine: COUNTER.make
149   rout_1_2926_3
        mod:
          attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
              attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
152     pre:
          alive_3 /\ (type_3 = 1)
        eff:
155       % postcondition
            (attr_1_2927_3' = 0) /\
            (attr_1_2928_3' = 0) /\
158       true
    /\
  % transitions
```

```
161       true


164   % actions for routine: COUNTER.increase_i
      rout_1_2929_1
        mod:
167       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
          pre:
            alive_1 /\ (type_1 = 1)
170       eff:
            % postcondition
              (attr_1_2927_1' = attr_1_2927_1 + 1) /\
173         true
    /\
    % transitions
176         true


179   % actions for routine: COUNTER.increase_i
      rout_1_2929_2
        mod:
182       attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
              attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
          pre:
            alive_2 /\ (type_2 = 1)
185       eff:
            % postcondition
              (attr_1_2927_2' = attr_1_2927_2 + 1) /\
188         true
    /\
    % transitions
191         true


194   % actions for routine: COUNTER.increase_i
      rout_1_2929_3
        mod:
197       attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
              attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
          pre:
            alive_3 /\ (type_3 = 1)
200       eff:
            % postcondition
              (attr_1_2927_3' = attr_1_2927_3 + 1) /\
203         true
    /\
    % transitions
206         true


209   % actions for routine: COUNTER.decrease_i
      rout_1_2930_1
        mod:
212       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
          pre:
            alive_1 /\ (type_1 = 1)
215       eff:
            % postcondition
              (attr_1_2927_1' = attr_1_2927_1 - 1) /\
218         true
    /\
    % transitions
221         true


224   % actions for routine: COUNTER.decrease_i
      rout_1_2930_2
        mod:
227       attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
              attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
          pre:
            alive_2 /\ (type_2 = 1)
230       eff:
            % postcondition
              (attr_1_2927_2' = attr_1_2927_2 - 1) /\
233         true
    /\
    % transitions
236         true


239   % actions for routine: COUNTER.decrease_i
      rout_1_2930_3
        mod:
242       attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
              attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
```

```
      pre:
        alive_3 /\ (type_3 = 1)
245   eff:
        % postcondition
          (attr_1_2927_3' = attr_1_2927_3 - 1) /\
248       true
    /\
  % transitions
251       true


254 % actions for routine: COUNTER.increase_j
    rout_1_2931_1
      mod:
257     attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
            attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
      pre:
        alive_1 /\ (type_1 = 1)
260   eff:
        % postcondition
          (attr_1_2928_1' = attr_1_2928_1 + 1) /\
263       true
    /\
  % transitions
266       true


269 % actions for routine: COUNTER.increase_j
    rout_1_2931_2
      mod:
272     attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
            attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
      pre:
        alive_2 /\ (type_2 = 1)
275   eff:
        % postcondition
          (attr_1_2928_2' = attr_1_2928_2 + 1) /\
278       true
    /\
  % transitions
281       true


284 % actions for routine: COUNTER.increase_j
    rout_1_2931_3
      mod:
287     attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
            attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
      pre:
        alive_3 /\ (type_3 = 1)
290   eff:
        % postcondition
          (attr_1_2928_3' = attr_1_2928_3 + 1) /\
293       true
    /\
  % transitions
296       true


299 % actions for routine: COUNTER.decrease_j
    rout_1_2932_1
      mod:
302     attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
            attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
      pre:
        alive_1 /\ (type_1 = 1)
305   eff:
        % postcondition
          (attr_1_2928_1' = attr_1_2928_1 - 1) /\
308       true
    /\
  % transitions
311       true


314 % actions for routine: COUNTER.decrease_j
    rout_1_2932_2
      mod:
317     attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
            attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
      pre:
        alive_2 /\ (type_2 = 1)
320   eff:
        % postcondition
          (attr_1_2928_2' = attr_1_2928_2 - 1) /\
323       true
    /\
  % transitions
```

```
326       true


329   % actions for routine: COUNTER.decrease_j
      rout_1_2932_3
        mod:
332       attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
              attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
          pre:
            alive_3 /\ (type_3 = 1)
335       eff:
            % postcondition
              (attr_1_2928_3' = attr_1_2928_3 - 1) /\
338         true
    /\
    % transitions
341         true


344   % actions for routine: COUNTER.foo
      rout_1_2933_1
        mod:
347       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
          pre:
            alive_1 /\ (type_1 = 1) /\
350           (attr_1_2927_1 = attr_1_2928_1 - 2) /\
              (attr_1_2927_1 > 0) /\
            true
353
          eff:
                 true
356

      % actions for routine: COUNTER.foo
359   rout_1_2933_2
        mod:
          attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
              attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
362     pre:
          alive_2 /\ (type_2 = 1) /\
            (attr_1_2927_2 = attr_1_2928_2 - 2) /\
365         (attr_1_2927_2 > 0) /\
          true

368     eff:
                 true

371
      % actions for routine: COUNTER.foo
      rout_1_2933_3
374     mod:
          attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
              attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
        pre:
377       alive_3 /\ (type_3 = 1) /\
            (attr_1_2927_3 = attr_1_2928_3 - 2) /\
            (attr_1_2927_3 > 0) /\
380       true

        eff:
383             true


386   % actions for routine: COUNTER.copy
      rout_1_11_1
        mod:
389       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
          pre:
            alive_1 /\ (type_1 = 1) /\
392           (arg_todo <> 0) /\
            true /\% unsupported expression
            true
395
          eff:
            % postcondition
398         true /\% unsupported expression
            true
    /\
401 % transitions
          true


404
      % actions for routine: COUNTER.copy
      rout_1_11_2
407     mod:
```

```
        attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
            attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
      pre:
        alive_2 /\ (type_2 = 1) /\
          (arg_todo <> 0) /\
        true /\% unsupported expression
        true

      eff:
        % postcondition
        true /\% unsupported expression
        true
 /\
% transitions
        true

    % actions for routine: COUNTER.copy
    rout_1_11_3
      mod:
        attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
            attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
    pre:
        alive_3 /\ (type_3 = 1) /\
          (arg_todo <> 0) /\
        true /\% unsupported expression
        true

    eff:
        % postcondition
        true /\% unsupported expression
        true
  /\
% transitions
        true


% actions for routine: COUNTER.standard_copy
    rout_1_12_1
      mod:
        attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
            attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
      pre:
        alive_1 /\ (type_1 = 1) /\
          (arg_todo <> 0) /\
        true /\% unsupported expression
        true

      eff:
        % postcondition
        true /\% unsupported expression
        true
  /\
% transitions
        true

    % actions for routine: COUNTER.standard_copy
    rout_1_12_2
      mod:
        attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
            attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
      pre:
        alive_2 /\ (type_2 = 1) /\
          (arg_todo <> 0) /\
        true /\% unsupported expression
        true

      eff:
        % postcondition
        true /\% unsupported expression
        true
 /\
% transitions
        true

    % actions for routine: COUNTER.standard_copy
    rout_1_12_3
      mod:
        attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
            attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
    pre:
        alive_3 /\ (type_3 = 1) /\
          (arg_todo <> 0) /\
        true /\% unsupported expression
        true
```

```
491     eff:
          % postcondition
          true /\% unsupported expression
494       true
    /\
   % transitions
497       true


500   % actions for routine: COUNTER.deep_copy
      rout_1_18_1
        mod:
503       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
          pre:
            alive_1 /\ (type_1 = 1) /\
506           (arg_todo <> 0) /\
            true

509     eff:
          % postcondition
          true /\% unsupported expression
512       true
    /\
   % transitions
515       true


518   % actions for routine: COUNTER.deep_copy
      rout_1_18_2
        mod:
521       attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
              attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
          pre:
            alive_2 /\ (type_2 = 1) /\
524           (arg_todo <> 0) /\
            true

527     eff:
          % postcondition
          true /\% unsupported expression
530       true
    /\
   % transitions
533       true


536   % actions for routine: COUNTER.deep_copy
      rout_1_18_3
        mod:
539       attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
              attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
          pre:
            alive_3 /\ (type_3 = 1) /\
542           (arg_todo <> 0) /\
            true

545     eff:
          % postcondition
          true /\% unsupported expression
548       true
    /\
   % transitions
551       true


554   % actions for routine: COUNTER.print
      rout_1_23_1
        mod:
557       attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
              attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
          pre:
            alive_1 /\ (type_1 = 1)
560       eff:
                true


563   % actions for routine: COUNTER.print
      rout_1_23_2
566     mod:
          attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
              attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
          pre:
569       alive_2 /\ (type_2 = 1)
          eff:
                true
572
```

```
    % actions for routine: COUNTER.print
575 rout_1_23_3
      mod:
        attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
            attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
578   pre:
        alive_3 /\ (type_3 = 1)
      eff:
581           true


584 % actions for routine: COUNTER.default_rescue
    rout_1_26_1
      mod:
587     attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
            attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
      pre:
        alive_1 /\ (type_1 = 1)
590   eff:
            true

593
    % actions for routine: COUNTER.default_rescue
    rout_1_26_2
596   mod:
        attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
            attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
      pre:
599     alive_2 /\ (type_2 = 1)
      eff:
            true
602

    % actions for routine: COUNTER.default_rescue
605 rout_1_26_3
      mod:
        attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
            attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
608   pre:
        alive_3 /\ (type_3 = 1)
      eff:
611           true


614 % actions for routine: COUNTER.do_nothing
    rout_1_27_1
      mod:
617     attr_1_2927_1, attr_1_2928_1, attr_1_1_1, attr_1_2_1, attr_1_10_1, attr_1_15_1,
            attr_1_16_1, attr_1_20_1, attr_1_22_1, attr_1_21_1, attr_1_24_1, attr_1_28_1
      pre:
        alive_1 /\ (type_1 = 1)
620   eff:
            true

623
    % actions for routine: COUNTER.do_nothing
    rout_1_27_2
626   mod:
        attr_1_2927_2, attr_1_2928_2, attr_1_1_2, attr_1_2_2, attr_1_10_2, attr_1_15_2,
            attr_1_16_2, attr_1_20_2, attr_1_22_2, attr_1_21_2, attr_1_24_2, attr_1_28_2
      pre:
629     alive_2 /\ (type_2 = 1)
      eff:
            true
632

    % actions for routine: COUNTER.do_nothing
635 rout_1_27_3
      mod:
        attr_1_2927_3, attr_1_2928_3, attr_1_1_3, attr_1_2_3, attr_1_10_3, attr_1_15_3,
            attr_1_16_3, attr_1_20_3, attr_1_22_3, attr_1_21_3, attr_1_24_3, attr_1_28_3
638   pre:
        alive_3 /\ (type_3 = 1)
      eff:
641           true


644
  ENVIRONMENT

647 INITIALLY
    alive_0 /\ (type_0 =  0) /\ ~alive_1 /\ ~alive_2 /\ ~alive_3
  GOAL
650 (alive_1 /\ (type_1 = 1) /\          ((true /\ (attr_1_2927_1 = attr_1_2928_1 - 2)) /\ (
        attr_1_2927_1 > 0))) \/
    (alive_2 /\ (type_2 = 1) /\          ((true /\ (attr_1_2927_2 = attr_1_2928_2 - 2)) /\ (
        attr_1_2927_2 > 0))) \/
```

```
    (alive_3 /\ (type_3 = 1) /\          ((true /\ (attr_1_2927_3 = attr_1_2928_3 - 2)) /\ (
        attr_1_2927_3 > 0)))
```

A.6 Execution Paths of *COUNTER.foo*

- Problem 1
  - Step 1: create COUNTER.make (1)
  - Step 2: COUNTER.increase_i (1)
  - Step 3: COUNTER.print (1)
  - Step 4: COUNTER.default_rescue (1)
  - Step 5: COUNTER.print (1)
  - Step 6: COUNTER.print (1)
  - Step 7: COUNTER.default_rescue (1)
  - Step 8: COUNTER.print (1)
  - Step 9: COUNTER.default_rescue (1)
  - Step 10: COUNTER.do_nothing (1)
  - Step 11: COUNTER.print (1)
- Problem 2
  - Step 1: create COUNTER.make (1)
  - Step 2: COUNTER.print (1)
  - Step 3: COUNTER.default_rescue (1)
  - Step 4: COUNTER.do_nothing (1)
  - Step 5: COUNTER.default_rescue (1)
  - Step 6: COUNTER.print (1)
  - Step 7: COUNTER.do_nothing (1)
  - Step 8: COUNTER.do_nothing (1)
  - Step 9: COUNTER.increase_i (1)
  - Step 10: COUNTER.make (1)
  - Step 11: COUNTER.do_nothing (1)
- Problem 3
  - Step 1: create COUNTER.make (1)
  - Step 2: COUNTER.do_nothing (1)
  - Step 3: COUNTER.increase_j (1)
  - Step 4: COUNTER.default_rescue (1)
  - Step 5: COUNTER.increase_i (1)
  - Step 6: COUNTER.print (1)
  - Step 7: COUNTER.default_rescue (1)
  - Step 8: COUNTER.print (1)

  - Step 9: COUNTER.do_nothing (1)
  - Step 10: COUNTER.do_nothing (1)
  - Step 11: COUNTER.default_rescue (1)

- Problem 4
  - Step 1: create COUNTER.make (1)
  - Step 2: COUNTER.default_rescue (1)
  - Step 3: COUNTER.increase_j (1)
  - Step 4: COUNTER.print (1)
  - Step 5: COUNTER.do_nothing (1)
  - Step 6: COUNTER.do_nothing (1)
  - Step 7: COUNTER.print (1)
  - Step 8: COUNTER.do_nothing (1)
  - Step 9: COUNTER.print (1)
  - Step 10: COUNTER.print (1)
  - Step 11: COUNTER.do_nothing (1)

- Problem 5
  - Step 1: create COUNTER.make (1)
  - Step 2: COUNTER.make (1)
  - Step 3: COUNTER.make (1)
  - Step 4: COUNTER.make (1)
  - Step 5: COUNTER.make (1)
  - Step 6: COUNTER.make (1)
  - Step 7: COUNTER.make (1)
  - Step 8: COUNTER.make (1)
  - Step 9: COUNTER.make (1)
  - Step 10: COUNTER.make (1)
  - Step 11: COUNTER.make (1)

- Problem 6
  - Step 1: create COUNTER.make (1)
  - Step 2: COUNTER.increase_j (1)
  - Step 3: COUNTER.increase_j (1)
  - Step 4: COUNTER.do_nothing (1)
  - Step 5: COUNTER.print (1)
  - Step 6: COUNTER.do_nothing (1)
  - Step 7: COUNTER.increase_i (1)
  - Step 8: COUNTER.increase_j (1)

Bibliography

[Aic01] B. Aichernig. *Systematic Black-Box Testing of Computer-Based Systems through Formal Abstraction Techniques.* PhD thesis, Institute for Software Technology, TU Graz, Austria, January 2001.

[ARM03] K. Arnout, X. Rousselot, and B. Meyer. Test wizard: Automatic test case generation based on *Design by Contract*™, draft report. Retrieved July 2003 from `http://se.inf.ethz.ch/people/arnout/arnout_rousselot_meyer_test_wizard.pdf`, 2003.

[Bec00] K. Beck. *Extreme Programming, Das Manifest.* Addison-Wesley, 2000.

[Bez03] E. Bezault. Gobo Eiffel project. `http://www.gobosoft.com/`, 2003.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, 2002.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *Software Engineering*, 21(10):785–798, 1995.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CGGT97] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for AR. In *ECP*, pages 130–142, 1997.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, MA, 1999.

[Ciu04] I. Ciupa. Test Studio: An environment for automatic test generation based on *Design by Contract*™. Master's thesis, Chair of Software Engineering, Eidgenössische Technische Hochschule Zürich, 2004.

[Cla04] M. Clark. *Pragmatic Project Automation, How to Build, Deploy, and Monitor Java Applications.* The Pragmatic Programmers, 2004.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[DDH74] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Number 8 in A.P.I.C. Studies in Data Processing. Academic Press, London, New York, sixth edition, 1974.

[DHJ+01] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[ECGN99] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[Flu04] B. Fluri. Reflection for Eiffel. Master's thesis, Chair of Software Engineering, Eidgenössische Technische Hochschule Zürich, 2004.

[FN71] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proc. of the 2nd IJCAI*, pages 608–620, London, UK, 1971.

[Gre04] N. Greber. Test Wizard: Automatic test generation based on *Design by Contract*™. Master's thesis, Chair of Software Engineering, Eidgenössische Technische Hochschule Zürich, 2004.

[Hol92] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992.

[HT03] A. Hunt and D. Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.

[Ikr03] I. M. Ikram. Genetic Algorithm classes. `http://eiffelzone.com/esd/gac/index.html`, 2003.

[Ise03] ISE Eiffel. `http://www.eiffel.com`, 2003.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[Jen03] R. M. Jensen. *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains*. PhD thesis, Carnegie Mellon University, June 2003.

[Jen04] Eiffel to C++ terminology mapping. `http://www.berenddeboer.net/eiffel/archive/eiffel-cpp-map.html`, 2004.

[LBR00] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, 2000.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall, New York, London, Toronto, Sydney, Tokyo, Singapore, 1992.

[Mey94] B. Meyer. *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.

[MK01] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 22–34, 2001.

[Zel99] A. Zeller. Yesterday, my program worked. Today, It does not. Why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.