An Interactive Software Development Workbench based on Biomimetic Algorithms

Robert Feldt

Department of Computer Engineering
Chalmers University of Technology
Gothenburg, SWEDEN

November, 2002

Abstract

Based on a theory for software development that focus on the internal models of the developer this paper presents a design for an interactive workbench to support the iterative refinement of developers models. The goal for the workbench is to expose unknown features of the software being developed so that the developer can check if they correspond to his expectations. The workbench employs a biomimetic search system to find tests with novel features. The search system assembles test templates from small pieces of test code and data packaged into a cell. We describe a prototype of the workbench implemented in Ruby and focus on the module used for evolving tests. A case study show that the prototype supports development of tests that are both diverse, complete and have a meaning to the developer. Furthermore, the system can easily be extended by the developer when he comes up with new test strategies.

1. Introduction

Developing software without faults is an important task in our modern society. The effects of faults in software can range from annoying, over costly to fatal. Having tools that support software developers in avoiding and removing faults is thus important.

One of the most expensive phases of software development is testing. Since testing does not directly add any functionality to the software there is a risk that software developers does not prioritize it enough. This is unfortunate since testing is a crucial step in ensuring the dependability of a piece of software. An important goal for a software development workbench should therefore be to support developers in finding and writing good tests and to automate testing processes.

A candidate for automating testing would be evolutionary computation. Evolutionary algorithms (EA) ruthlessly exploit weaknesses in the scaffolding software needed to support the evolutionary process [5]. In an earlier experiment of ours [9, 10] the genetic programming (GP) algorithm revealed a fault in our simulator. By outputting NotANumber at a crucial step in the simulation the evolving programs could get a perfect score and quickly solve the task. In that study, we fixed the fault, re-reviewed the simulation software for similar or other faults and restarted the experiment. In effect, the GP algorithm had helped us debug our software. In this paper we investigate this ability further and design a system that capitalizes on the effect.

Evolutionary algorithms have previously been used to generate test data for software testing [19, 26, 30, 37]. The focus has been on finding test data for structural testing although one study investigated black-box testing [38] and another one searched for test cases for mutation testing [3].

In this paper we propose a system that supports an incremental learning process for writing code and executable properties of that code. The distinghuishing feature is a biomimetic algorithm that can search for new test cases that highlights previously unshown features of the code and the specification. The biomimetic algorithm employs an evolutionary multi-agent system for the search, an artificial chemistry for communication between entities in the system and a fitness evaluation distributed on multiple evaluators that focus on different aspects. The system is interactive in that the developers actions indirectly affect the search.

In section 2 we give a background to software development and testing and present a theory for software development. The theory has implications for tools to support software development and motivates the workbench for interactive software engineering[1] described in section 3. Section 3 also describes the prototype WiseR of this workbench that we have implemented in the programming language Ruby. The exeperiments we have conducted with WiseR are described in section 4. Sections 5, 6 and 7 then summarizes related work, discusses the results and draws conclusions.

2. Software development and testing

Software Engineering (SE) is the *’application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software’* [18]. Much effort in SE has gone into finding good development processes. The most traditional example is the waterfall model with strict separation between requirements analysis, design, implementation and testing. Other processes have abandoned the strict separation between phases since they are often impossible to withhold in practice; during design we realize we have overlooked or underspecified some requirements and during implementation we realize the design is not complete.

A development process that has received much attention lately takes this stance to

[1]The workbench was originally named after ’workbench for interactive software evolution’ but we changed evolution to engineering so as not muddle the different uses of evolution in this paper.

a new level. Extreme Programming (XP), highlights testing as a fundamental activity to ensure quality and puts it in the front seat [4]. Some XP proponents even use the term ’test-first design’. The tests should be one of the major driving forces and developers start each iteration by writing tests for the code that needs to be implemented. The tests thus constitutes executable examples of the requirements on the system.

A natural companion to the test-first ideas of XP is a unit testing framework that supports the writing of tests and automates the execution and result collection of running them. Kent Beck originally developed SUnit for unit testing of Smalltalk classes but a number of similar systems have now been developed for other languages and they are collectively called XUnit [8]. In practice the XUnit frameworks allow the developer to specify concrete inputs for the software under test and to state the expected outputs. Even though they focus on unit testing they are general enough to support integration and system testing.

Although the unit tests written in a XUnit framework constitutes executable examples they are different from formal specifications. Recently there has been efforts to overcome this by marrying the JUnit Java unit testing framework with the formal specification language JML (Java Modeling Language). The contracts written in JML in the form of pre- and post-conditions that must be valid when calling a method on a class are used as test oracles in the unit testing. This alleviates the developer from the task of writing the expected outputs and thus simplifies the task of writing tests. The developer only need to set up the testing context and specify the inputs and call sequence.

The type of testing supported by the XUnit frameworks is called behavioural or black-box testing. It focus on the behavior of the software under test (SUT) and aims to test the responses of the SUT regardless of its implementation. An example of a black-box testing technique is boundary-value testing which locates and probes points around extrema and discontinuities in the input data domain.

Another type of testing is called structural or white-box testing. It focus on the internals of the implementation, often the control flow. The goal is to find tests that give good coverage of the program, ie. executes all statements or paths in the program.

A special type of testing is mutation testing which creates mutants of the progam. The goal is then to devise tests that reveal the mutants. By choosing mutant operations that resemble faults that programmers frequently introduce the hypothesis is that a test set revealing mutants should also be good at revealing real faults.

With this background let us now summarize our view of the software development problem in a theory and state the implications it has for a development workbench. A detailed description of the theory can be found in [11].

Our theory for software development is built from the five elements fundamental to any development process: a *patron* which has a need for a program or program component, a *specification* which states how the program should and should not behave, a program, a *developer* writing it and a *library* containing all of humanities total knowledge relevant to the problem at hand. Based on these elements a software development process is defined as an incremental learning process in which there are two main ways to make progress: refining an internal model of the developer or refining an artefact based on an internal model. Since the former underlies the latter it is more fundamental.

This theory has implications for tools supporting software development. They should trigger the identification of discrepancies between the internal models of the developer and the ideal artefacts that would lead to acceptable behaviour. One way for them to do that would be to create novel test invocations for the developer to consider. If the tool is creative in creating these invocations it could help the developer ’think outside the box’ and realize his knowledge is incomplete. Furthermore the tool should support the sharing of recipes for creating novel test invocations. If a system for sharing such recipes was in wide-spread use it could lead to more general progress in the area of software development.

The theory also describes tests as half-baked requirements; they are requirements without a verdict on the actual behavior the program showed. It would be very powerful to have a tool that generated test sequences and then executed the program on them and presented the test and the program behavior to the developer. This would allow the developer to classify the test as valid, invalid or irrelevant and the behavior as correct or wrong. The tool should then generate the code for setting up and checking this behavioral requirement. The tool should also allow the developer to note that a test is important but that they do not know what the expected behavior should be.

It is also important that tests are clearly documented. If a test identifies a fault we should be able to demonstrate it to others. Also, as the software is completed and further evolved to meet additional user requirements we want to be able to re-run previous tests. This regression testing is important since the new additions may affect previous code so that it fails where it previously worked.

It is also important that the tests are in a form that facilitates automation. Tests are typically numerous and it would be too cumbersome to execute them by hand. If they are written in a form that is easily executed this should lower the barrier for the developers to continously run and monitor the progress on the tests. Such a tight ’feedback loop’ is what Extreme Programming and other similar, recent development methods prescribe [4, 2].

In all but trivial cases testing cannot be exhaustive; there are far more possible combinations of indata than we have the time to run. Thus, the tests we choose to run should represent different classes of inputs. If the software correctly handles the few examples from an input class it is likely that it will handle all inputs in the class. This partitioning of the inputs into different classes should be visible from the tests to justify why we have chosen this particular set of tests.

Many tests often have the same structure and only the test data differs between them. Our development tools should allow the programmer to express these recurring patterns in a form so that it can be reused in later projects and by others.

3. WISE - a Workbench for Interactive Software Engineering

WISE is a design for an interactive tool supporting software development based on the theory above. It is an integrated environment for developing an executable, behavioral specification and a program that implements it. It also highlights the importance of tests and their close relation to the behavioral specification. WISE searches for tests with properties different from the tests it already knows of. Interesting tests are presented to the developer which can review them and classify them. This interactivity between the tool and the developer is central to the design of WISE.

A prototype of WISE called WiseR (WISE for and in Ruby) has been implemented in the programming language Ruby. It currently focuses on searching for tests although a simple GUI has been implemented to interact with the system.

WISE draws upon biologically inspired ideas and a running WISE system uses several biomimetic algorithms. Before we describe the philosophy behind WISE, its architecture and the WiseR prototype we motivate why biological ideas are used and the biological processes they resemble.

3.1. Biomimetic ideas in WISE

WISE is based on several ideas inspired by biological systems and uses algorithms modeled after nature:

- It is continously active even if no developer is present. It searches for better and more interesting tests or learns how to use the knowledge in the library.
- Few parts of WISE are cast in stone. When there are alternative solutions WISE implements several of them and then dynamically learns which one works best.
- Templates for tests are built from building blocks resembling cells in biological organisms. They have a membrane with ports that can connect to ports on other cells. In this way cells grow into larger clusters showing more complex behavior.
- Test cells interact within a biochemical system where proteins can be released and sensed. Cells communicate both with other cells, other entities in the system and with the outside world via the biochemical system.

- The basic commodity for cells is energy. Cells compete for energy by producing data or test runs. Evaluators probe the chemical system for data or test runs that are novel and give energy to the entities that produced it.

The reasons for this use of biological ideas are manyfold. From a philosophical viewpoint the problems facing a developer have many similarities with the ones that biological systems are facing. They are ill-defined. If they were not there would be no real development task since it is by definition the formalization of a system from loose beginnings.

The problems facing a developer are also dynamic. As she defines some part of the system, her choices affects other, yet un-defined parts of the system. As she learns more the importance of some parts might decrease while other parts becomes more important. Even worse, the target might change as the patron gets a new idea or changes the requirements.

In any development process there is room for multiple different choices. The developer must identify important trade-offs and study how different decisions affect the behavior of the system. A workbench supporting the developer must support this playing with alternatives and exploring differing avenues.

Central to any development process is creativity and innovation. The developer needs to be innovative in finding solutions, refining the specification and writing tests that show conformance. Above all the developer needs to be creative, and ’think outside the box’ to identify the faults in her own internal models.

Even though biomimetic methods may not be the best optimizers, they have an excellent track record when it comes to ill-defined, dynamic, explorative and creative processes. So from a philosophical viewpoint they are natural candidates as building blocks in a development workbench.

An additional reason for the use of biomimetic ideas is that evolutionary algorithms in previous studies have revealed faults in scaffolding code used during evolution.

One example was in one of our earlier studies where a GP algorithm evolved aircraft brake controllers [9, 10]. In this experiment a simulator was used to evaluate the aircraft controllers. The simulator was faulty by not correctly handling exceptional conditions from the controllers. In particular the GP algorithm found that by returning the float value not-a-number (NaN) in a specific state of the simulation it could trick the simulator into achieving its goal in a non-realistic way without expending any energy.

In the experiment above the goal was not to test the simulator. But since the solutions produced by the evolutionary algorithm interacted with the simulator it was in essence tested. The fault in the simulator was not identified automatically but required human analysis. But it was clearly evident from running the evolutionary system that something was not right. Since the exploitation of the fault in the simulator was such an effective means for the EA to reach its goals all solutions in the population soon used the exploit. By tracing a simulation of one of the solutions the fault was easily spotted. This also points to the important interplay between the system and a human in finding and understanding the cause of a fault.

Other EC researchers have had similar experiences although few report on them in the final papers. In a recent paper [5] the EC researcher Peter Bentley says that when you work with evolution you

> *…get a few glimpses of the creativity of evolution through the bugs in your code: the little loopholes that are ruthlessly exploited by evolution to produce unwanted and invalid solutions…. Each result fascinating, and each prevented by the addition of another constraint by the developer. The bugs are never reported in any publication, and yet they point to the true capabilities of evolution.*

In this paper we extend this fault-revealing ability of EA to the testing of general software.

3.2. Goals and design philosophy

The goals for WISE are to

1. find new knowledge about the software under test (SUT), and
2. allow the developer to specify test building blocks, test strategies, and novelty criteria in a flexible way.

While goal 1 is obvious, goal 2 is explicitly stated since it is what makes 1 possible both for the current SUT but primarily for future development activity. The ’flexibility’ in goal 2 means that WISE should limit the form in which the developer can describe the system components as little as possible. It should also make as few assumptions about them as possible. This leads to the sub-goal that WISE must also find new knowledge about the system components since we cannot assume the developer has stated (or knows) all of it.

Central to WISE’s design is to focus on the interaction between the developer and the system. The developer is the ultimate source of knowledge so if the system is in trouble it should inform him. The system should also encourage feedback on its progress. Since testing can never be exhaustive for non-trivial systems we want to find tests that are meaningful.

Another design principle is to avoid making choices about the values of parameters to the components in the system. With choices we bias what can be expressed and limit creativity. Thus when there is a choice of different alternatives WISE implements several alternatives and let the system choose which ones are effective at run time.

3.3. Basic Architecture

The WISE architecture has four main parts: an interface to the developer (UI), a control module that formulates goals and initiates searches, a knowledge base acting as a central repository for information in and about the system, and compute daemons that perform searches.

The UI is centered around the two artefacts that should be the end results of the development process: the behavioral specification and the program. It can also display tests to the developer and allows him to classify them. If he classifies a test and the output from the program as valid the test is transformed to a behavioral requirement and becomes part of the specification. Central to making this work is the need to make things explicit. In as far as possible the artefacts are stated in a form that the workbench can actively use in later steps. Information in comments or ’outside’ the system is a lost oppportunity since the system has less information to base its decisions on[1].

The UI gives the developer access to the knowledge base. The knowledge base is a local version of the library that is part of the theory presented in [11]. In the future we envisage that the knowledge base could be an interface to central libraries on the Internet or directly linking the knowledge bases of for example the developers in a development team.

The Control module is the main initiator of actions in a WISE system. It can take commands from the developer and set up searches on a compute daemon. If the developer does not give any commands it can formulate goals and sub-goals and initiate actions based on them. As an example, if the user has not written or loaded any new code that needs to be tested the Controller can consult the knowledge base and initiate a search for test sequences that creates a certain type of data.

When the controller initiates a search it sends the search description and any information needed for the search off to a compute daemon. To decouple the WISE front-end from the compute daemons this communication is inter-process via a TupleSpace over TCP/IP. This decouples the UI and control module from the compute daemon and allow one WISE front-end to use multiple compute daemons. Even though high performance is not a goal of this study this separation was deemed necessary since many of the biomimetic algorithms are compute-intensive. Including this in the design from the beginning should make things easier later.

[1]Unless this information can be parsed and made useful…

The compute daemons are independent and may work on separate problems handed out by a WISE front-end. the TupleSpace model was chosen since it allows for very flexible communication between nodes [15, 39]. The daemons are not expected to cooperate to solve problems but the simplicity and power of the TupleSpace model does not disallow it. It decouples the WISE module from the number and type of daemons. The TupleSpace provides a noteboard where information can be published and seen by multiple or only some subset of the daemons. It also allows the daemons to publish information that can be seen both by the WISE front-end and by other drones.

The searches in the daemons is done in a dynamically evolving system that builds test templates that adds novel knowledge. The knowledge can be either about the piece of software under test (SUT) or about the search builing blocks and how to assemble them.

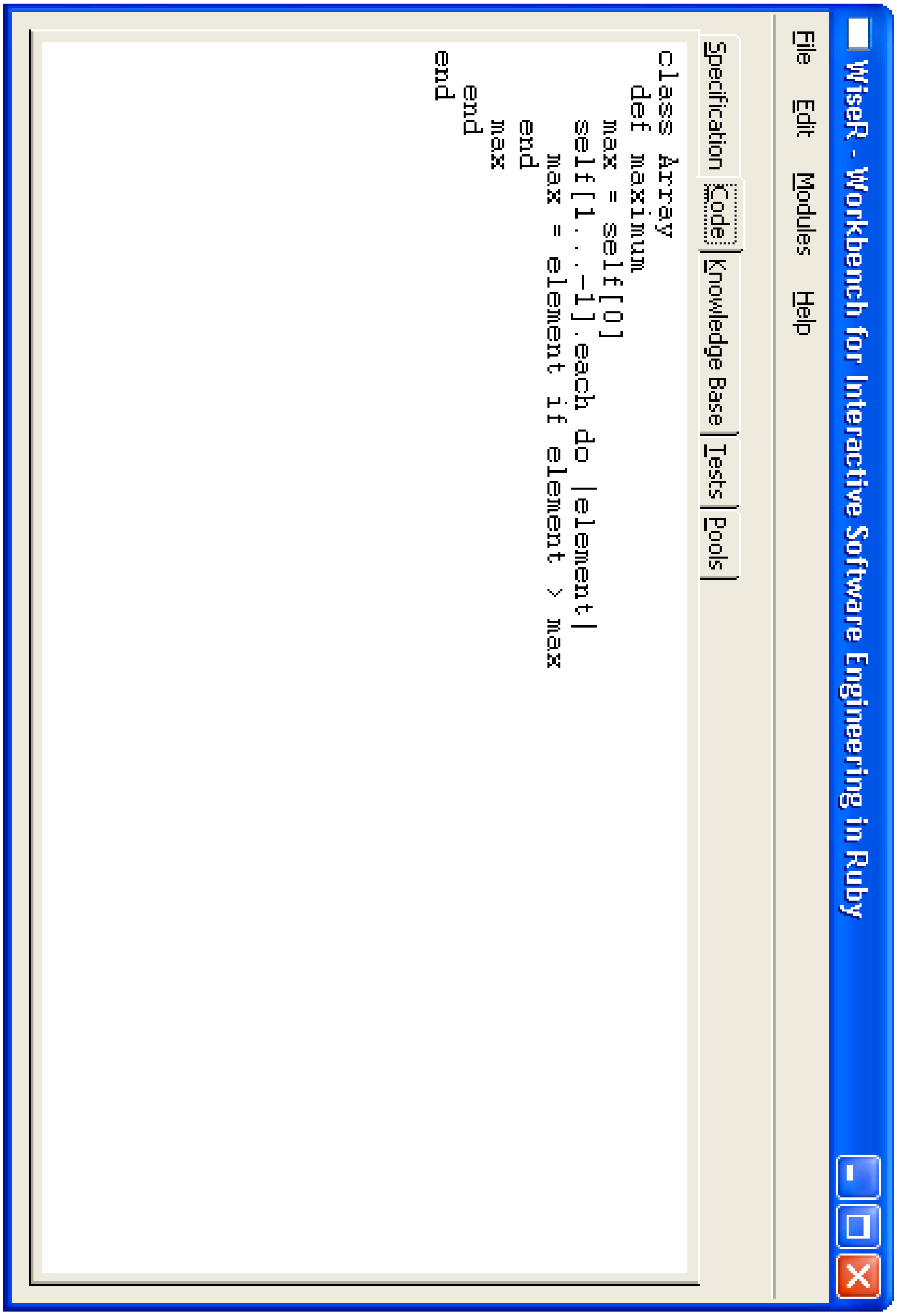
3.4. WiseR - the prototype

A prototype WISE implementation has been implemented in the object-oriented programming language Ruby [36, 29]. It is called WiseR (Wise for Ruby). Ruby was chosen since it is a high-level language with many features that support fast prototyping. It belongs to a new class languages that sports dynamic typing and easy access to all parts of the execution environment. This was deemed necessary in order to allow experimentation with different parts of the system.

Like the popular languages Java and C++ Ruby is object-oriented. Unlike them it is dynamically typed ie. there is no type checking at compile-time. In fact, there is no compile-time since Ruby is not compiled but interpreted[1]. Even though these aspects make Ruby non-typical compared to major languages in use today we do not think it confounds our results. If anything, the dynamic typing makes things harder for a tester. He cannot simply look at the code and see what types are allowed for parameters. In some sense, there is less information available in the code and thus more information needs to be re-discovered. The availability of a compiler would speed up the system, make the client more responsive and allow for more powerful computations. However, in research and for a prototype we think other factors are more important. More detalied information about Ruby can be found in appendix B and in the books [36, 29].

WiseR is both implemented in and supports development of Ruby code. All three artefacts are expressed as Ruby code. Even though this is not a requirement in WISE it makes things easier for our prototype. We can reuse common functionality used in analyzing the artefacts. It is also easier to exchange information between them.

[1]Although Java was originally and is often used as an interpreted language there are numerous compilers available. In any case Java is typed so differs from Ruby in this regard.


WiseR - Workbench for Interactive Software Engineering in Ruby
File
Edit
Modules
Help
Specification
Code
Knowledge Base
Tests
Pools

```
class Array
  def maximum
    max = self[0]
    self[1...-1].each do |element|
      max = element if element > max
    end
    max
  end
end
```

Figure 1. WiseR GUI main window with the Code window active and loaded with the Array#maximum source code

The focus when developing WiseR has been on a module that searches for tests, WiseR-Tests. In addition to WiseR-Tests the system contains of a GUI and small implementations of the control module and a knowledge base. The main window of the WiseR GUI is shown in figure 1.

Before going deeper into WiseR-Tests we briefly describe the WiseR GUI and the support WiseR has for writing specifications.

3.4.1. The WiseR graphical user interface

WiseR's main window, shown in figure 1, has 5 tab windows for different types of information. The Specification and Code tabs are always present. They are two edit windows for writing the specification and the program, respectively. The 'Knowledge Base' tab is also always present. It gives an overview of the hierarchical data stored in the knowledge base and allows the developer to change parameters etc.

In the 'Modules' menu the developer can load a module. After loading the WiseR-Tests module it adds to additional tab windows. The 'Pools' tab shows all the pools that are currently active and can show some simple statistics for them. The 'Tests' window is the place where tests found by searches in the pools are reported to the developer.

Examples of how the different tabs look during a run can be found in the case study section below.

3.4.2. Writing specifications in WiseR

Even though the model for software development introduced in [12] and summarized in section 2 above considered requirements as atomic invocations of the program with a state, input stimuli, output and a classification of the invocation, WiseR gives rudimentary support for writing and checking requirements in a more general form.

A specification in WiseR is written as a set of properties in a class inheriting from the class Properties. The superclass Properties add some helper methods for defining the 4 different types of properties supported by WiseR:

- pre - A pre-condition that should hold before a method is called. Gets the arguments to the method as arguments.

- post - A post-condition that should hold after a method is called. Gets the output from the method, the object and the arguments for the call as arguments. Can NOT access the object before the call.

- invariant - A condition that should always hold for objects of the class, both before and after any call.

- raises_exception - Indicates that the method must raise an exception when called. Gets the arguments for the call as arguments.

We see that the first three are simply the ones dictated by Bertrand Meyer in his design by contract method [24, 25]. We added raises_exception since it seemed useful. Not much thought has gone in to this at this stage though and the way specifications are written may have to be updated in the future. For example, the post-conditions in WiseR cannot access attributes of the object before the call was made. This is a rather serious limitation for expressiveness of specifications and needs to be adressed in the future.

All of the three first methods above take a symbol that gives the name of the condition and a block of code that implements the condition. The block must evaluate to true if the condition holds or false if it doesn’t. The latter indicates that the specification is not fulfilled.

The raises_exception property takes a name, the class of the raised exception and a block implementing the condition under which the exception should be raised.

In addition to the four property creating methods above there are two more to indicate which class and method the properties should hold for. They are called for_class and for_method, respectively.

Additionally WiseR can add behavioral atomic requirements as described in section 2 and in paper [12]. They are added in a second class after the properties class at the bottom of the specification.

An example of a full specification is given in the case study section.

3.5. WiseR-Tests

WiseR-Tests is a WiseR module that searches for test templates with novel features. It starts a search by creating a *pool* and filling it with *cells* that are thought relevant to the search. The cells represent recipes for how to create (small) parts of a test template. Cells communicate with each other by sending *proteins* into a *biochemical fluid* in the pool. *Ports* on the cells have protein *sensors* that detect proteins flowing by. If the protein is a product that the port could have use for it saves some of the proteins. When the cell later ’runs’ it can use the protein to search for and connect to the originating cell. As cells connect to more cells and grow into cell clusters they can produce and emit more complex products.

Cells in the pool compete for energy. Without energy a cell hibernates. Cells with high energy levels can execute more actions and have a higher chance of growing into mature cell clusters that produce unique products.

The main source of energy is the *evaluators* on the surface of the pool. Like ports they have protein sensors attached to the biochemical fluid of the pool. Different evaluators sense different types of products. When the sensor of an evaluator fires the evaluator examines the product to assess its novelty. If the product is novel the evaluator assigns an energy score to it and boosts the originating cells energy level. An evaluator also communicates with the outside world by updating the knowledge base with the knowledge gained from the analyzed products.

With this high-level description of the main components of WiseR-Tests let us now go into the details on each one of them.

3.5.1. Cells

To construct a test we need an object of the class to be tested, a sequence of calls to the object, input data for each call and a description of what to do with the generated results. Essentially a test is a small program itself, invoking the class under test (CUT) for a specific purpose.

One approach we could take would be to evolve such test cases directly. For example, a genetic programming algorithm could be used to assemble test programs from the syntactical elements of the programming language. However, there are a number of problems with such an approach.

If we evolve source code directly there is no information about the higher level structure of the test. There is only the code. We will have a hard time writing whole test strategies with only the syntactic elements of the programing language. How should we specify what and how things can vary? Strategies are templates for a piece of source code; not for individual code elements.

When developing tests the goal is not only that they should efficiently test the implementation and show its conformance to the specification. They should also justify for humans (developers or ’customers’) that the system has been thouroughly tested. It is hard to see how such a justification could be built at the same time as evolving the code from atomic syntactic elements. There simply is not enough information to describe the semantics of the test in human-understandable terms. A possible solution would be to analyze the evolved test to produce a description of it. This seems very hard and a backward kind of way. Our representation for evolving tests should support descriptions at its core so that one and the same representation can be used both for running the test and generating a description of what it does.

A further problem with an evolutionary process based on low-level syntactical elements would be that it is unclear how it would scale. Evolutionary algorithms are used on increasingly larger problems and the evolved solutions are more complex but there is still a question of how well they will scale to really complex problems. By evolving the tests from higher-level building blocks we increase the likelihood that it will scale.

Finally, it is often the case that several tests have the same structure. Only some constants or input data differs between tests. The representation we choose must support the easy generation of large number of tests having the same structure.

These problems have led us to define a more powerful representation for test building blocks than what is traditionally used in for example Genetic Programming. In fact the design itself is inspired by cells which are the main building blocks of all biological systems.

The basic building block for our tests are cells. Each type of cell is implemented as one Ruby class. When writing a new cell there are two things the developer must do. He must specify the ports of the cell and he must give one method that implements the functionality of the Cell.

To capture the fact that a single cell can often generate a number of variants. Example of variants are a cell OrderingOfElements that can sort the elements in an array. It has two variants: one for sorting ascendingly and one for sorting descendingly. This allows one and the same cell to capture related variants together. By varying a variant specifier when executing the cell we can choose which variant will be chosen.

In addition to variants cells can often generate random examples of a variant. The OrderingOfElements above cannot be randomized[1] but for example a FixnumGen cell for generating Fixnums would have no variants but many different randomizations. Any combination of variants and randomization is possible.

As an example here is the definition of the ArrayGen cell for creating Array's filled with objects:

```
class ArrayGen < DataGenerator
  semantics "Array of size v(:size) filled with t(:element)"

  out_port :out, {:type => Array}

  in_port :size, {:semantics => "Size of generated arrays",
           :type => "Positive Fixnum or zero"}

  in_port :element, {:semantics => "Elements for array",
              :type => Object}

  def max_num_variants
    [port(:size).max_num_variants, port(:element).max_num_variants]
  end
```

[1]Well it actually can if it takes input from other cells that can be randomized.

```
  def run(token)
    Array.new(e(:size)).map {e(:element)}
  end
end
```

The ArrayGen cell is a data generator and thus inherits from the DataGenerator base cell class. The semantics line gives the semantics of the cell. Then comes the definitions of one out port and two in ports. The ports are named and assigned meta-data that further defines them. Then comes two methods. The max_num_variants method returns an array with the combined number of variants of the cells connected on the ports. The run method is the one that will be called when the cell produces a product. Here it executes the cell connected to port :size, creates an Array of that size and then fills it by repeatedly calling the cell connected to the element port.

One thing to note is how the string given as the semantics for the cell contains references to the cells connected to the respective ports. The ’v’ method call will insert the value received on port size while the ’t’ method returns the type of the objects returned on port element. This simple scheme allows descriptions of the tests to be built. Since its a simple scheme it will not work in more complex situations but it gives a reasonable hint.

In addition to DataGenerators there are also CodeGenerators. CodeGenerators are cells that implement a piece of Ruby code. They can range from simple cell that call a method to complex test strategies.

A major design goal was to allow flexibility for the developer writing cells. As few requirements as possible should restrict how cells can be written and how they work. A cell should be an isolated building block for building a piece of a test. How to assemble the building blocks to create tests should not be presepcified.

This flexibility is acheived by a flexible cell connection process. Cells connect through ports. A port has a type which specifies what other ports it can connect to. A cell can have zero or multiple ports and the number of ports can change dynamically based on what other cells it has already connected to.

Figure 2 shows an example of a cell cluster for testing the Array#maximum[1] method on arrays of Fixnum’s[2]. It is built from four cells. SizeOfDataStructure generates common sizes of datastructures such as array’s and hash’es. When executed it generates a size and sends it through its out port to the in port named size on the ArrayGen. The ArrayGen cell creates an array of the given size and fills it with elements

[1]In Ruby C#m indicates the instance method named m on an object of class C.

[2]Fixnum is the Ruby class for 31-bit integers

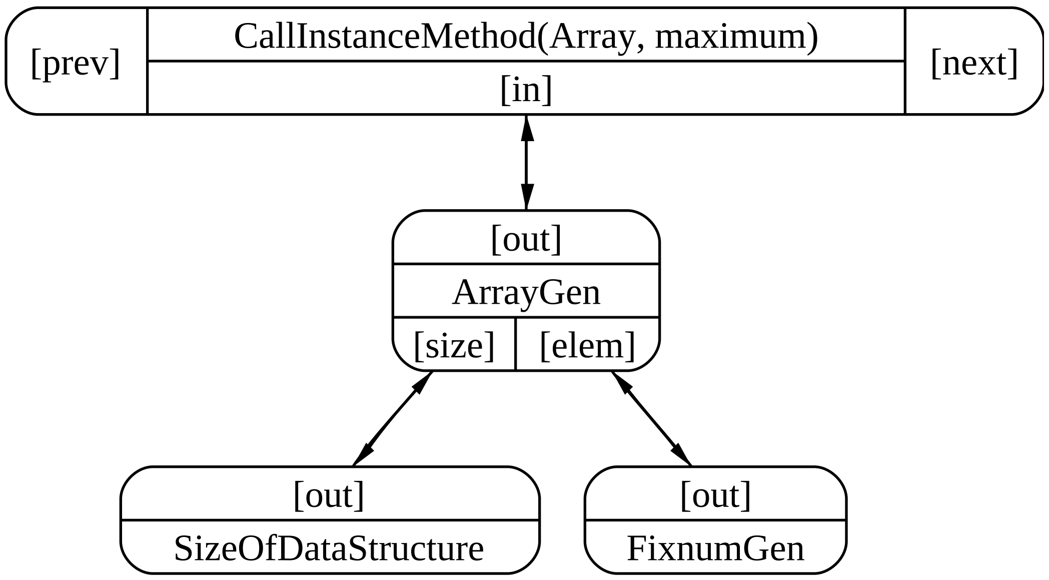


Figure 2. Test cell cluster with four cells for testing the Array#maximum method

generated by the FixnumGen cell. Finally the CallInstanceMethod cell calls the maximum method on the object received through its in port.

The CallInstanceMethod cell in the figure has two additional ports: a prev and a next port. They are used to build more complex tests by linking together several statements.

Cells are active agents in the system and are scheduled to be run depending on how much energy they have. When a cell runs it selects one of a set of cell actions and executes it. Every cell has the same set of cell actions but the parameters that determine the specifics of the action is saved in a genome in the cell. Different cells have different genomes and can thus evolve to perform different actions. In this way we need not explicitly tune parameter values in the system and cells with different functions can evolve different behaviour.

In the WiseR prototype there are currently only 4 different actions:

- Cloner - clones the cell
- Producer - produces a product if we are connected enough to be able to produce something
- Connector - connects to other cells if we have free ports and any of the port sensors have picked up something interesting
- Disconnector - disconnects one or more of the cells in the cluster by breaking up a connection

The actions are implemented as separate Ruby classes inheriting from one and the same base class. This model makes it very easy to experiment with alternative actions; you simply write a new action and hooks it up to the base Cell class. An example would be to implement a crossover action. We have not yet done that since crossover can be said to emerge from the combined actions of a disconnector and a connector.

All actions cost energy based on how long time they execute. A high-resolution timer is used to measure this and deduct the energy level of the cell accordingly.

The cloner cell action is typically activated when the energy of the cell is high. However its activation probability is governed by a constant that is part of the cell genome. Once activated the cloner simply clones the cell and any cells that is connected to it. When a cell is cloned its genome can undergo mutations. After a cell has cloned it must transfer a percentage of its energy to the clone. The percentage is also part of the genome and undergo evolution.

In contrast to the cloner, the producer cell action is activated pretty often given that the cell has matured enough so it can produce a product. A cell is mature if it has only one open port and its either an out port or a left sequence port. The former can produce a Ruby object while the latter can run a test and produce a TestRun product.

When a cells producer action is activated it will create a token and execute itself on the token. The token is passed among the cells during production and records intermediate results, calls to methods and results returned from statements. The token also selects for a certain variant of the variants the cell can produce and specifies the random seed to use for randomizations. Both the variant selector and the random seed are saved and together with a mature cell uniquely determines a test run.

Cell executions are protected in several ways so that invalid connections do not lead to infinite loops or uncaught exceptions. A time out value is used and terminates the cell execution after a constant number of seconds that depends on the speed of the CPU[1]. The cell execution is also invoked within a protected block that will catch any exception. If a test run raises an exception it is used as the return value from the test run. This allows evaluators to check for exceptions and punish the cell clusters that caused them.

When a cell has been executed it packs up the product in a protein, tags it with a unique id and ejects it into the fluid. The process of creating proteins cost energy and that energy is reflected in the concetration of the protein sent out. The amount of energy to spend on the product protein is determined by the cell genome. The amount of energy sent is a sort of gamble the cell makes. By sending out more proteins the potential gain can be higher if an evluator likes the product. However, if they do not the cell will gain no new energy and the energy spent on producing the protein is lost.

[1]WiseR tests the speed of the CPU on startup.

Sending out many proteins also increases the likelihood that other cells i need of the cells product gets the message.

The connector action is pretty straightforward. When activated it checks if any port sensors have detected interesting products that we could have use for. The method used to assess if a product is interesting is a parameter and governed by the genome. One method simply chooses products at random, another ranks the products according to their semantic match with the ports sementics.

The disconnector action is even simpler than the connector. It can be activated by chance but the probability that it is activated raises (by how much is governed by genome) when there are connections in the cluster that cause invalid executions. The disconnector will choose one of the problematic connections randomly and disconnect it.

3.5.2. Ports

Cells connect to each other via ports. A port is a placeholder for a piece of the test coded for by the cell that can vary. Ports are either *connected* to another port or are *free* for new connections. Free ports can be further divided into *open* and *closed*. An open port has to be connected for the cell to be mature. Closed ports does not have to be connected for the cell to be mature. A mature cell can open a closed port so that it can grow into a more complex test template. Cells can add or delete ports dynamically if needed.

There are four common types of ports: in port, out port, left sequence port and right sequence port. In ports are used to receive objects from other cells and out ports are used to send objects to other cells. In ports can only connect to out ports and out ports can only connect to in ports.

Sequence ports are used to chain pieces of code together into more complex templates. Every test template must start with a closed left sequence port; it indicates the first statement of the test. Right and left sequence ports can only connect to each other.

Ports are implemented as a separate class and can be easily extended. This way new type of ports with new semantics can be added.

Ports serve a dual purpose. In addition to being the connection points between cells they are the cells main medium for communication. Cells can send out information through ports and ports have sensors that sense information sent by other entities in the system.

Ports have a simple form of memory. They keep information about which ports on other cells they have been connected to and statistics on what was sent and received by the port during a connection. This information can be exploited by the cell when deciding which ports and cells to connect to. When a cell dies or when the pool is halted the information saved in the ports can be mined and entered into the knowledge base. ’In’ ports are informed by the cell when the information received on the port resulted in an invalid execution of the cell. The ports can thus keep track of which other ports it is fruitful or meaningless to connect to. This way we need not require that the developer specify the exact requirements that needs to be fulfilled for a port to accept a connection.

As an example lets consider the ArrayGen cell from figure 2. Its in port named ’size’ expects a Fixnum value that is larger than equal to 0 since it cannot create arrays with a negative number of elemets. However, since there is no Ruby class for positive Fixnums the type expected on the size port is simply stated to be Fixnum. When WiseR evolves arrays it may happen that NegativeFixnumGen cells connect to the size port. Since they will never result in valid arrays the size port on ArrayGen will be informed of this each time the ArrayGen tries to produce a product. The port saves the type of cell and port it is connected to and calculates an exception rate, ie. the probability that this connection will cause an exception when the cell produces a product. Connections with high exception rates are the primary candidates the Disconnector cell action. Also when the cell dies and the ports are mined for information the knowledge base will be updated. The next time the system runs the knowledge base now shows that it is not a good idea to connect NegativeFixnumGens to the size port on ArrayGen. This will decrease the probability that such a connection happens again.

Often the developer has detailed knowledge about the port and what kind of connections it can accept. He can specify such knowledge as meta-data when adding a port to a cell. It is encouraged that ports at least have meta data describing their semantics ie. their purpose. This allows the selection of candidates for innjection into a pool based on semantic similarity as described above. However semantics are not required. If so the cell will be injected into pools randomly so that the system can learn about the type of connections it can take part in.

3.5.3. Biochemical fluid

Cells communicate by sending proteins into a simulated biochemical fluid surrounding the cells in the pool. Proteins are constructed from a series of acids. An acid is either an ordinary Ruby object or a special acid used when matching proteins. The special acids are:

- DontCareAcid - matches any other acid, used when we don’t care what is in a certain position of the protein
- ClassAcid - wraps a Ruby class, matches any Ruby object that is a kind of the wrapped class, used to match a whole series of proteins that are similar in

structure but only differ by the actual object

- MultiClassAcid - same as ClassAcid but matches object that is kind of one of several classes
- NumAcid - encodes a numerical value in the protein, used to indicate the concentration of the protein

The concentration of the protein indicates how many copies of the protein is sent out into the fluid. The implementation is simply to release on protein and have the concentration indicate the number of copies. This saves down on the number of computations the biochemical fluid needs to perform when delivering proteins to sensors.

When a sensor senses a protein there is a reaction that may decrease the concetration of the protein in the fluid. However there are sensor that sense proteins without reducing their concentration. This is for example used to get a notion of time steps in the system. For each iteration of the main pool loop that schedules cells for execution the pool sends out a time chemical in the fluid. Elements in the pool can sense the concentration of the time protein without affecting its concentraion.

It is possible to register reactions with the fluid. This can be used to setup up decaying proteins whose concentration decrease over time. The prototype only supports RateDecayReactions that consume a percentage of the currently available concetration of the protein. They are used for the Frustration protein which is emitted by evaluators when they have seen no progress for a long time. The Frustration protein slowly decays unless more frustration is emitted by evaluators. The level of frustration can be a good indication of the overall progress the search is making. Sensors for frustration can also trigger global pool actions. The prototype does not currently use them but an example would be an Earthquake pool action that triggered on high frustration levels and randomly killed off cells in the pool. After such an earthquake new types of cells could be given more room since previously dominating cells have been killed off.

The fluid is implemented as a simple tuplespace. Sensors register with the fluid object and are organized in a hierarchy depending on the specificity of its matching protein. When proteins are injected into the pool the fluid object look ups the matching sensors and lets them react with the protein in a random order until no more sensors are left or the concetration is zero. So the fluid in the prototype does not model a spatial structure of protein diffusion. It is as if all sensors and emitters where at the same distance from each other and the protein randomly reacted with some of them.

3.5.4. Pool

A pool is a container for cells in different stages of development. It has a biochemical fluid that the cells use to communicate with each other.

Pools allow any entity that has a biochemical sensor or emitter to connect to the fluid. This allows the WiseR front-end to sense the status of the evolution in the pool and injectors to inject new cells into the pool when the proteins indicate that progress is slow. Novelty evaluators connects to the pool to sense products produced by mature cells and to send them energy based on the products novelty.

The pool is the main actor in the system and drives everything by scheduling cells that can execute. The pool keeps a list of the cells ordered from highest to lowest energy. At each time step it randomly select one of the cells from the top 10% of cells having highest energy.

3.5.5. Novelty Evaluators

Novelty evaluators are the main energy sources in the system. They evaluate products produced by the cells and clusters and give them energy if the product is novel.

Novelty evaluators sense products in the fluid with a sensor. The sensor reacts with matching proteins and thus grabs a part of them. If the evaluator likes what they see they will give energy back to the producing cell in accordance with how many proteins they grabbed. Cells that send out much proteins thus have a greater chance of being spotted and getting more energy back.

The actual constants that govern how many proteins an evaluator grab and how much they give back is evolved with an evolutionary strategy algorithm local to the evaluator. This was added to the system since it was hard to set the constants manually and they affected the overall success of the system.

Novelty evaluators was added to the system so that many different criteria for test novelty could be added independently of each other. The approach is similar to [35] and allows multiple criteria and even contradicting criteria to coexist.

Evaluators that evaluate test runs report their findings to the WiseR GUI. The GUI uses the information from the evaluators to sort the tests that are presented to the developer. More novel tests get higher scores and end up higher on the list.

The novelty evaluators that have been implemented for the WiseR prototype are:

- ViolatesPostcondition - violates a post-condition in the specification (only active if there are any postconditions) (10)
- ViolatesInvariant - violates an invariant in the specification (only active if there are any pre-conditions) (10)
- UnseenException - a previously unseen exception was raised when calling a method (9)

- UniqueMethodCalls - the number and order of methods called on the test object differs from previous tests (8)
- UniqueAttributeType - the type of object returned from an attrbiute of the class differs from previous tests (8)
- UniqueReturnTypes - results from methods have different type than previously returned results (5)
- UniqueCellTypes - the number and type of cells in the cluster that produced the test (5)
- UniqueCellConnections - the number and type of connections in the cluster that produced the test (5)
- UniqueAttributeValue - the object returned from an attribute of the class differs from previous tests (4)
- UniqueReturnValues - results from methods have different values than previously returned results (3)
- UniqueParameterNumber - the number of parameters used when calling a method are different than what has been previously used (2)
- UniqueParameterValues - the values of the parameters to a method differs from what has been previously used (1)

The number in parenthesis is an initial weight that the evaluators have. The weight is updated based on which tests the developer investigates and classifies. The weight is used to calculate an aggregate novelty score by summing the individual scores from the evaluators multiplied by their weight.

Many of the evaluators above are of a binary nature and will not change much during a run. However, when they do apply and give high uniqueness scores they ensure that the cell cluster that created the condition gets lots of energy. This promotes the exploration of similar cell clusters since the cells get many changes to run actions that may clone or disconnect cells and connect to other cell combinations.

However the gist of evaluation are the evaluators that evaluate the uniqueness of Ruby objects used as parameters in method calls, returned from method calls and returned from attributes.

These evaluators use distance functions to compare the similarity of Ruby objects. The distance functions for simple objects like numbers is simply the absolute value of their difference. For complex objects the evaluator checks the values of attribute methods on the object. For example for an Array the evaluator would call the length method to compare the lengths of the Arrays. This need not be prespecified since Ruby supplies reflective methods so that the evaluator can dynamically find out which methods are attributes and call them. For objects that can be enumerated (aggrations such as Array's, Hashe's etc) the evaluator will recursively apply itself to compare each subobject.

3.5.6. Feedback from developer interaction

WiseR continously monitors the actions of the developer and uses this information to guide the search. When the developer classifies a test the cells that participated in producing the test gets an energy boost based on how unique the test is according to the evaluators and by counting how many tests with the same classification that are already in the specification.

The weight of evaluators that gave the test a high novelty value are also boosted somewhat.

3.5.7. Controller

The controller in WiseR is very simple. It receives goal statements from the UI, divides them into sub-goals and identifies building blocks that could be useful in searching for a test meeting the goal.

There are only two goal statements supported by the WiseR prototype:

- 'Test method X on class C'
- 'Co-test methods Y1, Y2, …, YN on class C'

The former tells the system to focus on testing one method while the latter indicates that a set of methods should be tested together. One example where the latter goal could be used would be when testing the push and pop methods of a PriorityQueue. With the second type of goal statement above we could tell the system that these methods 'go together' so its probably a good idea to call them both in a test.

Upon receiving a goal statement the controller checks whether the method(s) to be tested are instance or class methods. An instance method is a method on an instance (object) of a class. A class method is a method on the class itself. The canonical example of a class method is the method used to create instances[1]:

```
# Create an array object (instance of Array) of length 3
a = Array.new(3)
```

[1]In Ruby the '#' character indicates that the rest of the line is a comment

If the method under test (MUT) is an instance method the Controller formulates the sub-goal of first creating an instance of the class. So the goal of testing an instance method is broken down into first creating an instance of the class and then testing the method on the instances. After goal analysis the goals enter a goal queue where they are served in turn. The current status and history of the goal queue can be inspected by the developer in the UI.

The controller now takes the next goal from the queue. It first looks in the knowledge base if we already know how to create objects of this type. If not it searches the knowledge base for Cells with meta-data that is relevant to the goal. A cell is deemed relevant if it is known to generate objects of the right type or if its semantics matches the semantics of the goal. The semantic matching is done by a simple heuristic algorithm based on the edit distance between words. The algorithm used is further described in appendix A.

The search for Cells to include as building blocks in the search is repeated in several steps to ensure that the in-ports of previously chosen Cells has some chance to connect to other cells. For example if an ArrayGen Cell has been chosen the controller will in the next round search for Cells with meta-data that is relevant for connecting to the ArrayGen's in-ports.

4. Case Study

In this section we describe a case study that have been carried out on the WiseR system.

4.1. Array#maximum

This experiment is an interactive session with WiseR on a simple example, the Array#maximum method used throughout this paper. The example is somewhat contrived since the implementation of Array#maximum contain a bug on purpose in order to show the workings of the system.

4.1.1. Experimental set-up

For this experiment we start with WiseR in a clean state, ie. it has no knowledge except for the standard cells that come with the basic WiseR system. The cells in such a basic system contains data generators for the common Ruby classes, different ways to call methods with different number of parameters, many different cells for generating boundary cases of arrays since they are so common in Ruby programs and cells to multiplex between datagenerators.

4.1.2. The experiment

The developer needs a method on the Array class that gives the maximum object of all the objects in the Array. He starts WiseR and chooses the specification window. He writes a few properties that the method must obey:

```
class ArrayMaximumProperties < Properties
  in_class Array

  for_method :maximum

  # maximum must be element of array
  post :max_is_an_element do |out, ary|
    ary.include? out
  end

  # maximum must be larger than equal all the elements
  post :max_is_larger_than_equal do |out, ary|
    ary.all? {|element| out >= element}
  end
end
```

He goes on to the Code window to create an implementation and writes:

```
class Array
  def maximum
    max = self[0]
    self[1...-1].each do |element|
      max = element if element > max
    end
    max
  end
end
```

The algorithm simply loops over the elements and keeps the max element in a variable and then returns it. Note that there is an error in the code for the range specifying which elements to loop over (-1 refers to the last element in the array and a…b to the range from a up to but NOT including b). This was introduced for the sake of the experiment.

As soon as he has entered a valid Ruby program (ie. that parses ok), WiseR initiates a search by creating a pool and injecting cells into it. The cells are chosen based on their semantics as described earlier.

As test runs are found by the pool they show up in the Tests window. The developer goes there to check on the findings. Listed on the top is the entry ’Raises NameError: undefined method each for nil’. This indicates that the maximum method has raised an exception. By clicking on the entry it expands to show different test runs for which this happens. By double-clicking on one of them it can be viewed in the lower window. Figure 3 is a screen capture of the WiseR window at this point.

The developer views the source code for the test. It shows the test code and the output from the method in the comment. He realizes he has forgotten about the boundary case of an array of size 0. He decides that the method should return the nil object when called on an empty array since this is the standard way in Ruby to handle empty arrays. He updates the max_is_an_element to

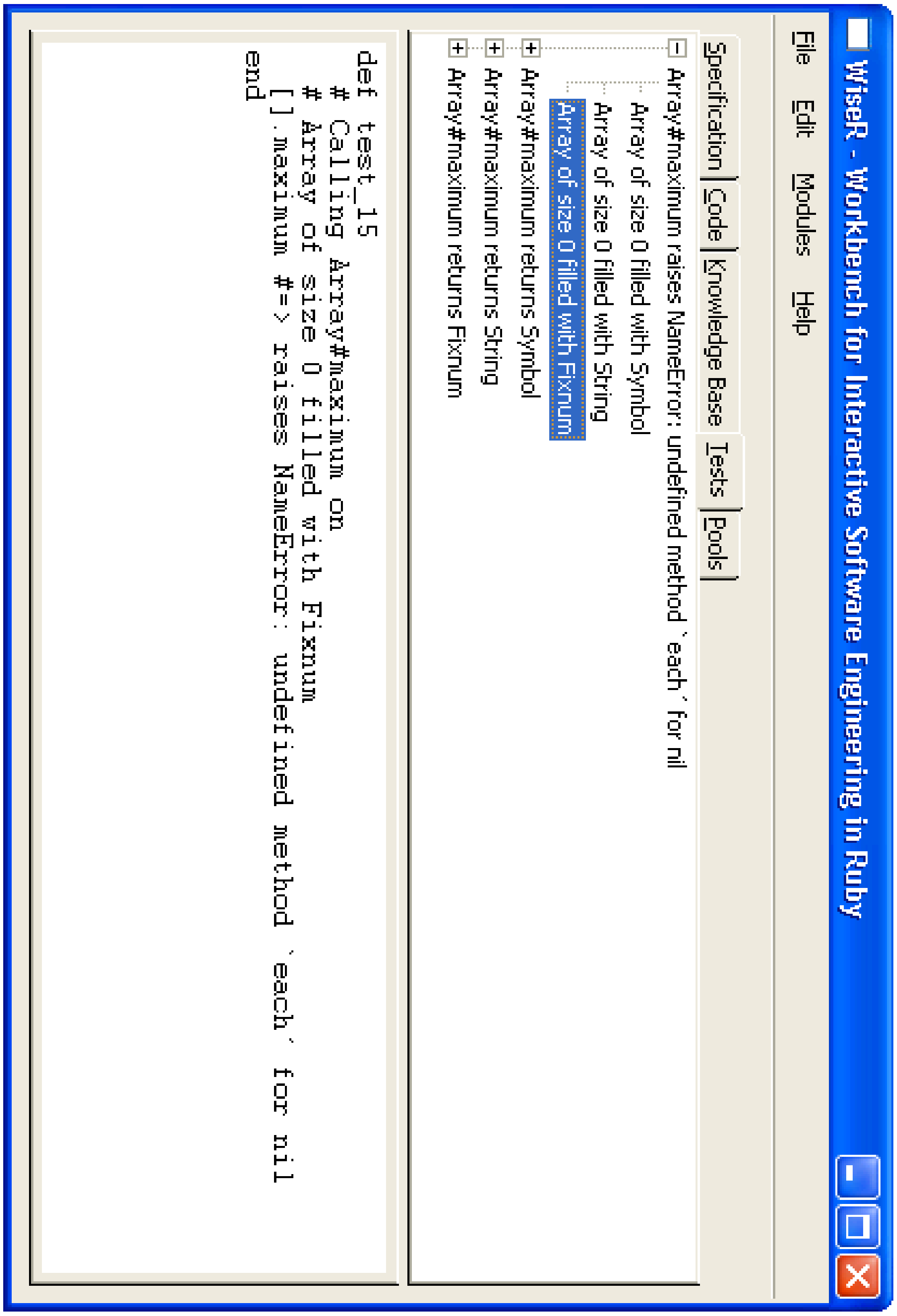
```
# maximum must be element of array or nil if array empty
  post :max_is_an_element_or_nil do |out, ary|
    if ary.length > 0
      ary.include? out
    else
      out == nil
    end
  end
```

and adds a new first statement in the implementation that returns nil if the length is zero. When the program or specification is updated all the evolved tests are reexecuted and the window with tests redrawn.

The previously selected test now returns nil which is the required behavior so the developer classifies it as being a valid test run. WiseR converts the test into a requirement and adds it to the specification after the ArrayMaximumProperties class in the specification window:

```
class ArrayMaximumSpec < Specification
    def req_1
      # Calling Array#maximum on
      # Array of size 0 filled with Fixnum
      # Array of size 0 filled with String
      # Array of size 0 filled with Symbol
      assert_equal(nil, [].maximum)
    end
  end
```

We see that WiseR not only added the test we choose that had Fixnum’s as elements but also the tests that give the same code but are generated in other ways. However, the algorithm WiseR uses for merging tests is simplistic and should be extended. Right


WiseR - Workbench for Interactive Software Engineering in Ruby
File
Edit
Modules
Help
Specification
Code
Knowledge Base
Tests
Tools

Array#maximum raises NameError: undefined method `each' for nil
Array of size 0 filled with Symbol
Array of size 0 filled with String
Array of size 0 filled with Fixnum
Array#maximum returns Symbol
Array#maximum returns String
Array#maximum returns Fixnum

```
def test_15
  # Calling Array#maximum on
  # Array of size 0 filled with Fixnum
  [].maximum #=> raises NameError: undefined method `each' for nil
end
```

Figure 3. The Tests window in WiseR showing the 'Array of size 0 filled with Fixnum's' test

now it has only very simple heuristics for how to merge test descriptions that do not work for more complex tests.

At this stage the developer also realizes that the description of this test is not optimal. The test is labeled 'Array of size 0 filled with Fixnum' but it is not interesting what type of objects are in an empty array. He decides to update the capabilities of the system by adding a specialized test cell for this boundary case. He goes into the knowledge base and clicks down in the hierarchy to WiserTests/Cells/DataGenerators/Array and chooses 'New cell based on…' from the context menu accessed by right-clicking the mouse. He gets an edit window with the code for ArrayGen except that the ArrayGen name has been removed so he can write a new cell name. He writes 'EmptyArrayGen' as the name, deletes the two in ports and updates the meta-data and code to execute when running the code. He ends up with (you can compare this to the code for ArrayGen given above):

```
class EmptyArrayGen < DataGenerator
  semantics "Empty Array without any elements (length is 0)"

  out_port :out, {:type => Array}

  def run(token)
    Array.new(0)
  end
end
```

He doesn't need to define the max_num_variants method since it returns 1 by default as defined in the DataGenerator class.

Going back to the Tests window there are now a new top-level entry 'Array#maximum violates max_is_larger_than_equal_all_elements'. The tests showing this behavior has two Fixnum or String elements. After ensuring himself that the max_is_larger_than_equal_all_elements is a valid property to require and that its implementation is flawless the developer examines the tests in close detail. The one with Fixnum's look like:

```
def test_64
  # Calling Array#maximum on
  # Array of size 2 filled with Fixnum
  [-196424314, 837355386].maximum #=> -196424314
end
```

Something is obviously not right with the implementation and he goes to review it. Well, the pool has now spotted the problem with the implementation that it never compares to the last element in the array. Before updating the implementation he turns the tests into requirements. He classifies them as valid state and valid input but invalid output so WiseR generates only a skeleton requirement where the developer can fill in the expected output. Here's the requirement above after he has filled in the blank:

```
def req_2
  # Calling Array#maximum on
  # Array of size 2 filled with Fixnum
  assert_equal(837355386, [-196424314, 837355386].maximum)
end
```

He updates the implementation to use the full range 1..-1. Tests are now rerun based on the new implementation and he goes back to the Tests window. On top is now new entries of the form 'Array#maximum raises NameError: undefined method > for nil'. The tests showing this behavior each have two or more elements that are nil or symbols. The developer realises that his implementation will only work for objects that have comparison operators that allow ordering of the elements. He considers adding a property to ensure this but instead decides that raising this exception is the correct behavior in such situations so he classifies the tests as valid and the output as expected. WiseR adds requirements of all the tests. As an example here is the one with 4 symbols:

```
def req_5
  # Calling Array#maximum on
  # Array of size 4 filled with Symbol.
  assert_raises(NameError) {[:vT, :Ho, :ye, :zZ].maximum}
end
```

He also adds the property at the top of the specification:

```
raises_exception :elements_must_be_comparable, NameError do |ary|
  ary.all? {|element| element.kind_of?(Comparable)}
end
```

Going back to the Tests he sees a new type of entry labeled 'Array#maximum raises TypeError: failed to convert Fixnum into String' and examining the test he sees an array with both Fixnum's and String's in it. Aha, not only need they be Comparable they need to be comparable to each other. In a similar way as before he write a new property and converts some tests into requirements.

Going back to the Tests window again he examines some of the 'normal' tests that do not raise an exception or violates any properties. He examines them and turns several of them into requirements by classifying them as valid.

He continues doing this until he feels satisfied there are no more errors.

4.1.3. Analysis

The interactive WiseR session reported above shows that WiseR was effective both in finding bugs in the specification and implementation of the Array#maximum method. The problem with max_is_an_element was an error in the developer internal model of what the method should do. There was a boundary case he hadn't considered. Seeing the actual test that shows the erroneous behavior gives insight of where the internal model needs to be refined.

5. Related work

5.1. In evolutionary algorithms for testing

There has been a number of studies that use genetic algorithms (GA's) for structural testing, ie. ensuring that all parts of the implementation are executed by the test set. Jones et al used a GA to generate test-data for branch coverage [19]. They use the control-flow graph (CFG) to guide the search. Loops are unrolled so that the CFG is acyclic. The fitness value is based on the branch value and the branching condition. They evaluated the approach, with good results, on a number of small programs.

Michael and McGraw at RST corporation have developed Gadget - a tool for generating test data that give good coverage of C/C++ code [26]. Gadget work for full C/C++ code and automatically instruments the code to measure the condition/decision coverage. This requires that each branch in the code should be taken and that every condition (atomic part of a control-flow affecting expression) in the code should be true at least once and false at least once. Four different algorithms can be used to search for test data in Gadget: simulated annealing, gradient descent and two different genetic algorithms. One of the GA's scored the best on a large (2046 LOC) program which is part of an autopilot system but on synthetic programs the GA had problems with programs of high complexity. Simulated annealing fared better here. In all of the experiments random testing fared the worst when the complexity increased.

Pargas et al use a GA to search for test data giving good coverage [30]. They use the control dependence graph instead of the control flow graph since it gives more information on how close to the goal node an execution was. Their system uses the original test suite developed for the SUT as the seed for the GA since it should cover the programs requirements. To reduce the execution time their system employs multiple processors. They compare their system to random testing on six small C programs. For the smallest programs there is no difference but for the three largest programs the GA-based method outperforms random testing.

Tracey et al presents a framework for test-data generation based on optimisation algortihms for structural testing [37]. It is similar to both Jones et al and Michael and

McGraw approaches and uses a CFG and branch condition distance functions. They use both simulated annealing and a genetic algorithm for the optimisation. Their tool is automated and works with ADA code.

Tracey have used a similar technique for functional (black-box) testing [38]. The formal specification is described with pre- and post-conditions that each function must obey. The goal is to find indata that will fullfill the pre-condition and the negated post-condition. These expressions are converted to disjunctive normal form. All pairs of single disjuncts from pre- and post-conditions are considered targets for the search since a fault is found when either of them is fulfilled.

Mueller and Wegener used an evolutionary algorithm to find bounds for the execution time of real-time programs and compared it to static analysis of the software [27]. Even though the evolutionary algorithm cannot give any safe timing garantuees it is universally applicable and only requires knowledge about the programs interface. Static analysis can give garantuees but only in a theoretical world. It needs extensive knowledge about the actual hardware if we are to trust the results. Such knowledge may not always be available.

Baudry et al have used genetic algorithms for evolving test sequences for mutation testing of Eiffel programs [3]. Their model is similar to ours in that they focus on specification, implementation and tests. Their specifications are written with pre- and post-conditions and invariant. A tool mutates the programs and a genetic algorithm searches for test sequences that kills the mutants. The GA is seeded with test sequences written by the developer. Mutants that are not killed by the GA are analyzed by hand to see if they are mutants that did not change the workings of the software.

Genetic algorithms have been used to generate test scripts for GUI testing [20]. Even though the tests generated were simple the authors concluded that the GA could test an application in an unexpected, but not purely random way.

5.2. In evolutionary multi-agent systems

Like WiseR the system developed by Krzysztof Socha and Marek Kisiel-Dorohinicki uses multiple entities, called agents, that together explore a multi-objective search landscape [34]. Agents exchange a non-renewable resource called life energy in transactions based on comparing their behavior against a fitness function. The traditional evolutionary processes of selection and inheritance are not governed by some central authority but happen locally in each agent. When agents have high energy they will reproduce and when energy goes low they die. Agents have a physical location in the world they are in and all actions happen locally. The system has been applied to optimization of some numeric test functions with promising results.

Our system differs from Socha’s and Kisiel-Dorohinicki’s by not using locality, having heterogenous agents, a renewable ’energy’ and a dynamically changing fitness landscape. The agents in our system are cells of diverse constitution. They exist in a pool without a physical location. Communication thus go to many more other cells. This is needed since there may be fewer receivers and too strong locality might hinder progress. More importantly our system does not try to optimize a static fitness function and the fitness function is not smooth. In these conditions we don’t think it would be possible to have a non-renewable ’energy’ resource.

5.3. In cell-based programming models

Some research groups are studying how programming models inspired by biological systems can be used to build more robust systems. George et al recently introduced such a model where cell programs are automatons containing discrete states and transitions between the states [16]. Cells can sense there immediate neighbourhood and send out chemicals. They can also divide. The authors beleive they will be able to build self-healing software with the model.

The ’Amorphous Computing’ group at MIT studies organizational principles and programming languages for coherent behavior from the cooperation of myriads of unreliable parts [1].

5.4. In evolutionary design systems

The Agency GP system is used to let designers explore the design space of 3D objects [35]. It has a very flexible approach to fitness evaluation where agents evaluating one aspect of fitness can be released into the system and affect fitness evaluation. New agents can be added as needed. The authors claim that this model is well suited for fitness evaluation based on conflicting, non-linear and multi-level requirements. Our model with evaluators is very similar to this agent-based fitness model.

Ian Parmee and colleagues have investigated the use of genetic algorithms for conceptual engineering design [31]. Their research has focused on different ways to allow the designer to guide the multi-objective optimization carried out by the genetic algorithm. They have applied their systems to ’traditional’ engineering disciplines such as aerospace and civil engineering.

5.5. In biochemically inspired system

Lones and Tyrell have proposed a new representation for genetic programming inspired by gene expression and enzymes in the metabolic pathways of cells [22, 23]. The building blocks for the GP algorithm are enzymes containing an activity and a set of specificities. The activity is the function the enzyme encodes and the specificities are templates that determine which other components the enzyme can connect to. Genotypes are sets of enzymes and develop into a program by starting the build process from an output enzyme. The system has been evaluated on the evolution of simple, non-recurrent digital circuits. The WiseR-Tests system shares many similarities with Enzyme GP (EGP). Our cells corresponds to EGP’s activities and our ports to EGP’s specificities.

Many other researchers also build computational models based on modeling cell communication via chemicals. An overview of different approaches is given in [17] which also presents an aggregated model taking different parts from the earlier models. Their system is very similar to ours in that a blackboard is used for communication between autonomous agents. However, their ultimate goal is to model cells for medical research.

5.6. In software testing

The QuickCheck system by Claessen and Hughes is a tool for automatic specification-based testing of programs written in the functional programming language Haskell [6]. The programmer provides a specification of the program by writing properties that the functions in the program must satisfy. The programmer can also combine simple test data generators into more complex ones. The data generators are then used to generate random data for testing the properties of the specification.

The Ballista system can be used for robustness testing of commercial-off-the-shelf (COTS) components [21]. They use the very simple criterion ’Crash or not?’ to determine if the response was valid and thus do not require a behavioral specification. The reason is that specifications are often not available for COTS software. The test sets generated by Ballista are exhaustive based on the data types of parameters to each function. There are generators available for each data type and they return extreme or boundary values. Our approach with data generators is very similar to Ballista’s with the exception that we allow multiple generators for each type and generators can be combined by connecting to each other.

5.7. In methods for semi-automated software development

The Programmer’s Apprentice (PA) was an attempt to build intelligent assistants to support in requirements analysis, design and implementation of a program [32]. They sought to automate the programming process by applying techniques from Artificial Intelligence. As a step towards that long-term goal they built assistants that could help the developer make intelligent decisions. For example, the implementation assistant allowed a programmer to construct programs by combining algorithmic fragments stored in a library.

PA is similar to WISE in that it allows the developer to bypass the system and directly enter code (or tests in WISE). But PA’s way to represent knowledge is different.

It is based on finding and encoding knowledge in pre-specified formats. Restrictions are thus put on how people must enter knowledge about the domain. The knowledge is also represented in a form that is different from the implementation language. In contrast the format used to represent knowledge in WISE is the same as the programming language itself. This makes things easier for developer since they do not need to learn another language. Another difference is that the Programmer's Apprentice system does not concern itself with testing.

The approach taken by PA can be called rule-based. A similar approach is the case-based reasoning approach to automated SE taken by some systems [7]. Like WISE they use some fuzzy measure of similarity to find components from a library that are relevant to a task. Like WISE they also learn by allowing the developer to add knowledge to the system. However, none of them have focused on testing and few of them produces artefacts that can easily be read by humans [7]. Since WISE focuses on knowledge about tests and test sequences are often simpler than the software they test the tasks facing WISE is simpler. However, a possible future work can be to investigate if and how more complex meta-data and matching schemes, as used in case-based SE systems, can be used in WISE.

Scheetz et al used an AI planner to generate test cases from an UML class diagram [33]. The UML diagram needs to be augmented with test-specific information.

6. Discussion and future work

6.1. Discussion

On why we did not compare the biomimetic search algorithm to random search. The interactive aspects of WiseR makes it hard to compare the system to a blind random search. If we compare the system without any developer interaction we the system is crippled and it is hard to draw conclusions about the full power of the system. Such an experiment could shed light on the importance and effect of the developer interaction. However, it is not even clear what should be considered a random search to which we could compare. What amount of information should we allow the random search to have? Should it have access to the port memories showing which ports are valid or invalid to connect to? Should it be allowed to use the semantic matching algorithm to select a set of cells that are promising? It is not clear-cut what the answers should be and the limited time available for this study did not allow us to investigate this any further. It is an important point for future work though.

On why it is fast enough even though Ruby is interpreted. Even though Ruby is interpreted and between 5-100 times slower than compiled C (depending on the type of task) the WiseR system can find tests in reassonable time. One reason is that the system can test very many cell clusters while the developer investigates a single test. Another reason is that much knowledge about potentially good tests is captured in the cells. They are high-level building blocks that have shown to be useful in previous testing efforts.

On how tied the system is to Ruby. Even though our goal has been for a generally useful system the WiseR prototype uses many special features of Ruby that may not be available in other languages. Reflection is used in the evaluators to find way to compare unknown objects. The fact that Ruby is interpreted also helps since we can easily reload tests. But there are not only downsides with going for other languages. A statically typed language would simplify things since the types of data would be known.

On WiseR's complexity. The design of WiseR might appear complex on the surface. Even if there is no absolute way to measure and compare the complexity of software systems we think a major cause for WiseR's apparent complexity is that it utilizes concepts not commonly used in software designs. The fact is that the complete WiseR system, including rudimentary graphical interfaces and the code for the test cells, is about 3800 lines of Ruby code[1]. We do not consider that a major software system.

On the risc of using automated methods to search for tests. There is a clear risc with using automated methods such as the one employed by WiseR for finding tests: we get a false sense of security by seeing the mass of tests that can be fairly easily added. However, if the system does not have the right information to base the search on it only searches a small subset of the space of possible input sequences. We note that this is a potential problem and that further development of WiseR should try to find methods to at least partyl overcome this. As an example the QuickCheck system by Claessen and Hughes can summarize the different input data sequences used in the test to show their disitrbution [6]. Something similar would be of value to WiseR, possibly combined with some way of visualizing these distributions.

6.2. Future work

6.2.1. Further experiments on WiseR-Tests

The case study described in section 4 are limited and on a very small development task. To really gauge the power of the developed system we need to perform more experiments on larger development tasks. Since the developer is such an important element in the WISE philosophy experiments should be carried out with several developers on one and the same development task. This could reveal differences in how developers experience and make use of WiseR's capabilities.

[1]Including comments and blank lines

One way to acheive developer feedback about the system could be to release it as open-source. An intriguing possibility would be to develop WiseR-Tests into a plugin for the new FreeRide Ruby integrated development environment [13].

6.2.2. Extending WiseR-Tests

There are a multitude of things that can be changed within the current WiseR prototype.

The connection between what cells are available in the CellPool and the tournaments in the Arena could be tighter. This might add important feedback that more quickly would steer the evolution to interesting areas of the TestSpace. It would also have the potential of trapping the process in local minima, ie. parts of the TestSpace where not much new knowledge is to be found. Exploring this trade-off might be worthwhile.

Improve the descriptions and process of generating test descriptions from a builder. Even though it currently give valuable information to the developer its a bit awkward and might be improved upon.

The controller is not currently part of the evolutionary process in WiseR. We consider this a drawback. This decision was made because we wanted the goal of the current searches to be visible to the developer. It was not clear how a goal could be formulated from an ongoing evolutionary process. An interesting area for future work would be to have a co- or meta-evolutionary search for CellSource's that could attach to the running search and add new cell material. Much of the scaffolding needed for this is already present with the system of triggers that monitor the knowledge base for when to inject new cell material into the pool.

6.2.3. Additional modules and extending WISE

CodeFaultAnalyser. The WISE system knows when you are correcting the source code and can thus save information about error corrections that you do. By saving the faulty and corrected syntax trees these trees can be analyzed. Over time the system can build a knowledge of your common faults and how to correct them. This information can be coupled with the Tester module to allow strengthening the tests based on mutation analysis. By basing the mutations on the actual faults of the user we can assure they are representative. This would be an excellent basis for mutation-based testing in the spirit of [Baudry et al] but with faults that are relevant for the current developer.

Explicitly representing faults also makes it possible to exchange fault sets between different developers. Thus over time this could lead to a common database of faults and how to solve them. Faults and corrections could also be exchanged over the internet etc.

A basic CodeFaultAnalyser module has been implemented in WiseR. However, it has not yet been integrated with Tester so that they can cooperate to strengthen the tests. Future work should strive to integrate these two modules so that they can use each others knowledge. For example, knowledge about the developers common faults could be used to create mutant code that tests would have to identify as faulty. A new novelty evaluator could thus reward tests that killed (new) mutants.

CellExtractor. There are many possibilities for automating the extraction of test cells, ie. test strategies, test code patterns and test data generators. By analysing existing test suites the system could find recurring patterns that can be extracted into new test cells and used for future test evolution.

Extractors could also insert specific data generators tailored to the implementation at hand. A static analysis of the code to be tested could reveal values that are boundary cases for this particular implementation and thus are likely to reveal new information about the system.

7. Conclusions

Based on the theory of software development proposed in [11] we identified opportunities for a workbench to support the development process. Our design for an integrated software development workbench, WISE, tries to follow the ideas indicated by the theory. It explicitly represents both the artefacts to be produced during development and encourage the encoding of meta-information about testing that can be used to derive meaningful tests.

WISE uses biomimetic algorithms to support the development processes. In particular, WiseR, our first prototype of WISE implemented in the programming language Ruby, evolves test templates that generate tests that add interesting information to the system. A common design theme is flexibility. The developer can continously interact with the automatic evolutionary process to guide it and turn it to interesting areas of the design space.

We have performed an initial case study on WiseR. It shows that WiseR can successfully evolve test sets that are both powerful and meaningful.

Appendix A. Algorithm for calculating semantic similarity

The WiseR prototype uses the following heuristic to compare the semantic similarity of two strings:

1. Divide both strings into it constituent words while dropping any non-alphanumeric characters.

2. Delete the short, ’trivial’ words that tend not to carry much information: a, an, the, in, on, of, and, or.
3. Calculate the edit distance (also called the Levenstein distance [14]) for all pairs of words in the two strings that share a prefix at least of length MIN_PREFIX_LENGTH. Inverse this to get a similarity score.
4. Sort all the similarity scores and sum them with a weight that is the inverse of their rank.
5. For each word that do not share a prefix subtract a MISSING_PENALTY from the similarity score

The MIN_PREFIX_LENGTH and MISSING_PENALTY constants was optimized (off-line) with an evolutionary strategy so that the heuristic above gives values that correspond with common sense on a set of strings. However, no evolution is done online on these parameters.

No doubt there are better algorithms for doing the semantic matching and investigating them could be an important future work. However, the above heuristic is simple and gives an indication of semantic similarity. Since the measure is only used to select building blocks it is not fundamental to the success of the system.

Appendix B. Short introduction to Ruby and its syntax

Since Ruby is not very well known we here gives a brief introduction to it and its syntax. This introduction is heavily based on a paper by Michael Neumann [28].

Ruby is an interpreted, object-oriented programming language. It is similar to both Smalltalk, Perl and Python but the syntax is more like Eiffel, Modula or Ada. Like Smalltalk everything is an object[1], there is a garbage collector, variables don’t have type, there is only single-inheritance and code can be packaged into objects. Ruby’s Perl heritage manifests itself in strong support for text-manipulation using regular expressions and substitution but also iterators. In many regards Ruby is very similar to Python although many consider the object-orientedness to be somewhat purer in Ruby than in Python.

In Ruby you declare a class and a method like:

```
class MyClass
  def my_method
    1
```

[1]There are some exceptions to this but they are not important here

```
  end
end
```

and can now get an instance (object) of the class and call the method with

```
o = MyClass.new
o.my_method         # Returns 1!
```

where everything after the # is a comment.

Ruby is dynamic. All classes are open and at any time you can add new methods to classes[1].

Ruby has an eval method so that Ruby code in strings can be evaluated.

```
eval "1"            # Returns 1!
```

References

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald J. Sussman and Ron Weiss. Amorphous Computing. *Communications of the ACM* **43** (5), 74-82 (2000).

[2] Kent Beck, Dave Thomas, Andy Hunt et al. Agilent Development Manifesto. Tech. Rep. (2001). URL *http://citeseer.nj.nec.com/justice93objectoriented.html*.

[3] Benoit Baudry, Vu Le Hanh, Yves Le Traon. Testing-for-Trust: The Genetic Selection Model Applied to Component Qualification. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, 2000.

[4] Kent Beck. *Extreme Programming Explained*, 1997.

[5] Peter J. Bentley. Fractal Proteins. Tech. Rep. (2002), Dept. of Computer Science, University College London.

[6] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, 2000.

[7] H. Dayani-Fard and J.I. Glasgow and D.A. Lamb. A Study of Semi-Automated

[1]Unless you explicitly freeze them in which case they are not allowed to change at all

Program Construction. Tech. Rep. AI memo 933A (1998), MIT's Artificial Intelligence Laboratory.

[8] Stephane Ducasse. SUnit Explained. Tech. Rep. (2000). URL *http://www.iam.unibe.ch/~ducasse/WebPages/Programmez/OnTheWeb/Eng-Art8-SUnit-V1.pdf*.

[9] Robert Feldt. Generating Diverse Software Versions with Genetic Programming: an Experimental Study. *IEE Proceedings - Software Engineering* **145** (6), 228–236 (December 1998). Special issue on Dependable Computing Systems

[10] Robert Feldt. Genetic Programming as an Explorative Tool in Early Software Development Phases. In Conor Ryan and Jim Buckley, *Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering*, pages 11–20, 1999.

[11] Robert Feldt. A Theory of Software Development. Tech. Rep. (2002), Department of Computer Engineering, Chalmers University of Technology, Gothneburg, Sweden.

[12] Robert Feldt. A Theory of Software Development. Tech. Rep. (November 2002), Department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden.

[13] Curt Hibbs, Rich Kilmer et al. *The FreeRide Ruby IDE home page*, 2002. URL *http://www.rubyide.org/cgi-bin/wiki.pl?HomePage.*

[14] Hal Fulton. *The Ruby Way*, 2001.

[15] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems* **7** (1) (1985).

[16] Selvin George, David Evans and Lance Davidson. A Biologically Inspired Programming Model for Self-Healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.

[17] P.P. González Pérez, M.C. Garcia, C.G. Garcia and J. Lagunez-Otero. Integration of Computational Techniques for the Modelling of Signal Transduction. Tech. Rep. (2001), Instituto de Quimica, Universidad Nacional Autonoma de Mexico. URL *http://www.cogs.susx.ac.uk/users/carlos/doc/GonzalezEtAl-integration-of-computational-techniques.pdf*.

[18] IEEE Standards Team. IEEE Standard Glossary of Software Engineering Terminology. Tech. Rep. (1990).

[19] B. Jones, H. Sthamer, D. Eyres.. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal* **11** (5), 299–306 (September 1996).

[20] David Kasik and Harry George. Toward Automatic Generation of User Test Scripts. In *Proceedings of the Conf. on Human Factors in Computing Systems: Common Ground*, pages 244-251, 1996.

[21] Nathan P. Kropp and Philip J. Koopman Jr. and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the Fault-Tolerant Computing Symposium*, pages 230-239, 1998.

[22] M.A. Lones and A.M. Tyrrell. Biomimetic Representation in Genetic Programming. In *Proceedings of the Workshop on Computation in Gene Expression at the Genetic and Evolutionary Computation Conference 2001 (GECCO2001)*, 2001.

[23] M.A. Lones and A.M. Tyrrell. Crossover and Bloat in the Functionality Model of Enzyme Genetic Programming. In *Proc. 2002 World Congress on Computational Intelligence.*, 2002.

[24] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[25] Bertrand Meyer. Applying 'Design by Contract'. *IEEE Computer* **25** (10), 40-51 (October 1992).

[26] Christoph C. Michael and Gary McGraw. Automated Software Test Data Generation for Complex Programs. In *Proceedings 13th IEEE Conference in Automated Software Engineering*, pages 136–146. IEEE Computer Society, October 1998. URL *citeseer.nj.nec.com/66954.html*.

[27] F. Mueller and J. Wegener. A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. In *IEEE Real Time Technology and Applications Symposium*, 1998.

[28] Michael Neumann. Comparing and Introducing Ruby. Tech. Rep. (February 2002). URL *http://www.s-direktnet.de/homepages/neumann/rb/download_ruby.html*.

[29] Michael Neumann, Robert Feldt, Lyle Johnson, Jonothon Ortiz. *Ruby Developer's Guide.* Syngress, 2002.

[30] Roy P. Pargas and Mary Jean Harrold and Robert Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability* **9** (4), 263–282 (July 1999). URL *citeseer.nj.nec.com/pargas99testdata.html*.

[31] Ian Parmee. *Evolutionary and Adaptive Computing in Engineering Design: The Integration of Adaptive Search Exploration and Optimization with Engineering Design Processes.* Springer Verlag UK, 2000.

[32] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Program Design Scenario. Tech. Rep. AI memo 933A (1987), MIT's Artificial Intelligence Laboratory.

[33] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman and A. Howe. Generating Test Cases from an OO Model with an AI Planning System. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE '99)*, 1999.

[34] Krzysztof Socha and Marek Kisiel-Dorohinicki. Agent-based Evolutionary Multiobjective Optimisation. In *Proceedings of Congress on Evolutionary Computation (CEC'02), Honolulu, HI, USA*, pages 109-114, May 12-17 2002.

[35] Peter Testa, Una-May O'Reilly and Simon Greenwold. AGENCY GP: Agent-Based Genetic Programming for Spatial Exploration. In *Proceedings of the ACSA*, 2002. URL *http://www.ai.mit.edu/projects/emergentDesign/agency-gp/ACSA.html*.

[36] Dave Thomas and Andy Hunt. *Programming Ruby: A Pragmatic Programmer's Guide.* Addison-Wesley, 2000.

[37] N J Tracey and J A Clark and K C Mander and J A McDermid. An Automated Framework for Structural Test-Data Generation. In *Proceedings 13th IEEE Conference in Automated Software Engineering.* IEEE Computer Society, October 1998. URL *http://www.cs.ukc.ac.uk/pubs/1998/974*.

[38] Nigel Tracey and John Clark and Keith Mander. Automated Program Flaw Finding using Simulated Annealing. In *Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis*, pages 73–81. ACM SIGSOFT, March 1998. URL *http://www.cs.york.ac.uk/testsig/publications/njt-mar98b.html*.

[39] P. Wyckoff. T Spaces. *IBM Systems Journal* **37** (3) (1998). URL *http://www.research.ibm.com/journal/sj/373/wyckoff.html*.