Specification-Driven Development of an Executable Metamodel in Eiffel

Richard F. Paige
Department of Computer Science,
University of York, York, UK.
paige@cs.york.ac.uk

Phillip J. Brooke
School of Computing, Communications and Electronics
University of Plymouth, Plymouth, UK.
phil.brooke@plymouth.ac.uk

Jonathan S. Ostroff
Department of Computer Science,
York University, Toronto, Canada.
jonathan@cs.yorku.ca

Abstract

Metamodels precisely define the constructs and underlying well-formedness rules for modelling languages. They are vital for tool vendors, who aim to provide support so that concrete models can be checked formally and automatically against a metamodel for conformance. This paper describes how an executable metamodel – which supports fully automated conformance checking – was developed using a model-driven extension of test-driven development. The advantages and disadvantages of this approach to building metamodels are discussed.

1. Introduction

A *metamodel* is a set of well-formedness constraints for a graphical modelling language, such as UML [5] or BON [12]. The constraints express when a model, written using the language, is correctly constructed. A metamodel implicitly defines a notion of *conformance*. Given a metamodel *MM* for a language *L*, and a model *M* written in the notation of *L*, we can define the *model conformance* relation as follows.

$$conforms(M, MM) \quad \widehat{=} \quad \forall c \in MM \bullet M \textbf{ sat } c \qquad (1)$$

i.e., the model *M* conforms to the metamodel *MM* providing that it satisfies all constraints in the metamodel; the metamodel necessarily includes constraints defining the abstract syntax and semantics of *L*. Ideally, we would like to check the satisfaction relation in (1) automatically. To our knowledge, there are no tools available that, given a metamodel and model, check equation (1) completely and automatically.

Metamodels underpin developers' understanding of models and diagrams, and are the foundation of CASE and diagramming tools that are used to draw models; thus, they can be considered critical systems, and it is important that they are of high quality. Specific quality requirements for a metamodel should include: *reliability* (the metamodel must correctly capture the constraints that define well-formed models [9]), *understandability* (the metamodel must be designed to be read and understood by tool builders and language users), and *maintainability* (metamodels, particularly for large languages like UML, will undergo frequent revision [1]).

Test-driven development (TDD), due to Beck [3], is an increasingly popular approach for building systems with reliability, understandability, and maintainability requirements. TDD helps achieve high reliability by amortising testing throughout the development process, and by focusing on short code-and-test increments. TDD has three steps in its micro-process: (1) write a test (without worrying if it does not compile); (2) write enough code to make the test pass; (3) *refactor* the code to eliminate redundancies and other design flaws introduced by making the test pass – thus, tests act as specifications that drive the design process. Like all agile methods, TDD excels at dealing with requirements that change during development. It also emphasises the production of simple designs, and generally can improve the maintainability of systems, because of the emphasis on simplicity in design.

There are two main limitations with TDD: it is a code-based development method only; and, there are expressiveness limitations with using tests as specifications that drive design. The former is of particular concern when developing systems where modelling is useful, e.g., high-integrity systems, systems with substantial reliability requirements, and projects where models are essential for explaining the

system to others. The latter is a fundamental limitation.

TDD provides several essential capabilities for building metamodels: it has been proven useful in building systems with high reliability requirements; it deals well with changing requirements, and due to its refactoring capabilities it can be useful for building extensible systems; and because it emphasises simple designs, it can promote more understandable systems. However, the limitations of TDD imply that pure TDD is not sufficient for designing and implementing metamodels.

We propose instead that a new integration of TDD and model-driven techniques, called *specification-driven design* (SDD) [6], is well-suited for building metamodels. SDD has none of the limitations of TDD, and it works with models; models (with contracts) and tests can be used to drive the design process.

In order to evaluate this proposal, we have recently used SDD to develop a design and implementation of the metamodel for the BON modelling language [12] (roughly equivalent to a subset of UML, consisting of class and communication diagrams) in Eiffel: metamodels are expressed as Eiffel programs, where metamodel constraints are captured as Eiffel contracts (examples to follow). We report on the utility of SDD for building metamodels in this paper; full details of the metamodel itself can be found elsewhere [8].

With this approach, the tests that can be written when using SDD are in fact Eiffel *encodings of models*. Thus, while metamodel is an Eiffel program with contracts, a model is a unit test written in Eiffel, and by executing the test successfully against the metamodel (which is part of the SDD process), we obtain a *fully automatic* way of checking the relation in equation (1), i.e., that a model conforms to the metamodel. Not only does this let us check the well-formedness of models, but it gives us greater confidence that the metamodel we are writing is in fact correct. We discuss this further in the conclusions and [8]; our focus in this paper is describing the application of the SDD process by which the executable implementation of the metamodel was developed.

While we present the application of SDD in terms of BON, and its models and views, the approach can equally well be applied to metamodelling other languages, such as UML. Where relevant, we point out key differences between using SDD to construct a metamodel for BON, versus metamodels for other languages; however, we claim, based on experience, that most of the differences will be minor.

2. Eiffel, SDD, and Related Work

Eiffel is an object-oriented programming language and method [4] tailored for building reliable systems. It provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, generic (parameterised) types, polymorphism and dynamic binding, and automatic memory management. Eiffel also supports *contracts*, which have been posited as a substantial technique for improving software reliability [4]. These contracts take the form of pre- and postconditions of routines (which specify the constraints on callers and implementers of routines) and invariants of classes (which specify well-formedness properties that all instances of the classes must obey).

A short example of an Eiffel class is shown in Fig. 1. The class *CITIZEN* inherits from *PERSON* (defining a subtyping relationship). It provides several attributes, e.g., *spouse*, *children* which are of reference type (in other words, *spouse* refers to an object of type *CITIZEN*); these features are publicly accessible (i.e., are exported to *ANY* client). Attributes are by default of reference type; a reference attribute either points at an object on the heap, or is *Void*. The class provides one expanded attribute, *blood_type*. Expanded attributes are not references; memory is allocated for expanded attributes when memory is allocated for their enclosing object.

The remaining features of the class are routines, i.e., functions (like *single*, which returns *true* iff the citizen has no spouse) and procedures (like *divorce*, which changes the state of the object). These routines may have preconditions (**require** clauses) and postconditions (**ensure** clauses), but no implementations. Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, i.e., before any valid client call on the object. In writing the invariant of *CITIZEN* we have used Eiffel's *agent* notation. Agents are a way to encapsulate operations in objects; the operation can then be invoked on collections (e.g., a set or linked list) when necessary. Two built-in agents in Eiffel are `for_all` and `there_exists`, which can be used to implement quantifiers over finite data structures. In this example, one agent is used in the class invariant: `for_all` iterates over all elements of *children* and returns *true* if its body – applied to each element – returns *true*. The body is a boolean expression which returns *true* iff the current citizen is a child of one of its parents. In other words, the agent expression is true iff all children have links to their parents.

The key element of Eiffel that makes it a useful language for metamodelling is its agent construct. Agents allow declarative specifications of well-formedness constraints to be written and automatically executed. This feature thus supports two of the key requirements for metamodelling mentioned earlier: reliability (agents are formal specifications that can be checked automatically) and understandability (agents support declarative specification of well-formedness constraints).

Other elements of Eiffel that are useful for metamod-

```
class CITIZEN inherit PERSON
feature {ANY}

  spouse: CITIZEN
  children, parents: SET[CITIZEN]
  blood_type: expanded BLOOD_TYPE

  single: BOOLEAN is
    do Result := (spouse=Void)
    ensure Result = (spouse=Void)
    end

feature {BIG_GOVERNMENT}

  marry is ...
  have_child is ...
  divorce is
    require not single
    do ...
    ensure single and (old spouse).single
    end

invariant
  single or spouse.spouse = Current;
  parents.count <= 2;
  children.for_all((c:CITIZEN):BOOLEAN
   do Result := c.parents.has(Current) end);
end -- CITIZEN
```

Figure 1. Eiffel class interface

elling include its integrated debugging facilities, static typing, and generic types. The former is particularly useful when building systems, as the debugger can inform the developer as to which contract has failed. In metamodelling terms, this allows developers to quickly discover which well-formedness constraint is not satisfied by a model.

Clearly, any language that provides an agent/closure construct, as well as standard object-oriented constructs like static typing, generics, and integrated debugging, would seem to be well-suited for metamodelling.

2.1. Specification- and Test-Driven Development

Test-driven development (TDD) is one of many agile development methodologies that have been proposed over the last few years. TDD is useful for producing reliable systems and for amortising the cost of testing across the development process. The TDD process described by Beck [3] is as follows.

1. Write a test which may not work initially (especially if code hasn't been written for a class). The test may verify the functionality of a method, the behaviour of a class, or a unit of functionality (i.e., it is an acceptance test).

2. Make the test work quickly, focusing on doing the simplest thing that allows the test to pass.

3. Refactor the design to eliminate duplication and improve the style and architecture in terms of simplicity, reusability and maintainability.

Reliability is emphasised in the process since changes to code are always made against specifications written in the form of tests, and the code can immediately checked against those specifications using unit testing. In TDD, unit tests are seen as (admittedly incomplete) formal specifications that drive the design [2, p38 and p51]. A key point of TDD is that testing is amortised across the development process, and a test suite is one of the deliverables of the process. There usually is no up-front design with TDD, though clearly some problem analysis and rough sketch design will be essential, e.g., for determining classes.

TDD can usefully be extended in a model-driven development setting, particularly one where correctness is a key concern. The extension is to use contracts (i.e., pre- and postconditions along with class invariants). The variant to TDD that we used in building the metamodel, which integrates elements of TDD, minimal model-driven development, and contracts, is called *specification-driven design* [6]. It is perhaps somewhat surprising that a model-driven mechanism like design-by-contract, and an agile process like test-driven development, are compatible, but the key point to note is that both tests and contracts are specifications: some specifications are easier to write as unit tests, others as contracts.

In this spirit, the statechart of Fig. 2 models the approach. SDD is not complete methodology, but it does represent the core practices of a balanced method. There are three states. The large state labelled TDD represents the traditional TDD process; the state on the right side of the diagram represents traditional design-by-contract. The left-most state involves writing specifications of complex multi-object behaviour using collaboration diagrams. These can in turn derive automatic test cases that can feed in to the TDD process.

As described in Fig. 2, SDD does not dictate where to start – it is the developer's choice based on the context, i.e., whether to start by writing unit tests, lightweight modelling, or writing contracts. System reliability can be improved by the most appropriate means at the time, e.g., via writing contracts on routines or classes, or by writing unit tests. However, whatever the starting point, the emphasis is on translating customer requirements to compilable and executable specifications. It might initially be possible to write a high level acceptance test, or perhaps the developer wants to sketch out some class diagrams and contractual specifications.

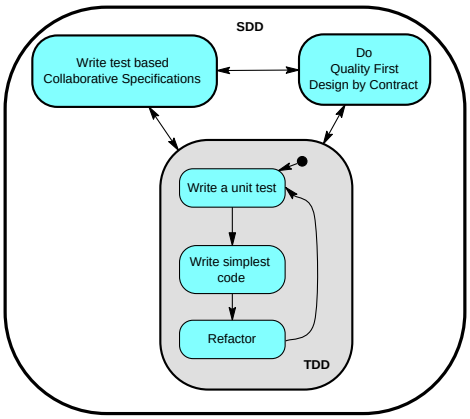

Figure 2. SDD – Specification Driven Design

2.2. Related Work

Metamodels have been constructed in a number of different languages and using a number of different development processes. Perhaps the most widely known metamodel is that for the Unified Modelling Language (UML). The UML metamodel is expressed in a combination of UML and its constraint language, OCL. The result captures the syntax and some of the semantic constraints on the language, and is an accepted standard. The metamodel for UML was developed over a number of years by a number of people using a *model-driven* process, making use of syntax-checking tools and peer review. The work of the 2UWorks partners has focused on improving the UML metamodel for extendability and understandability, via use of templates and patterns [1], again following a model-driven process. There has been limited work on ensuring that the UML metamodel is reliable; much of this has focussed on implementing small parts of the metamodel in a variety of tools (e.g., that allow syntax and type checking of OCL constraints) in order to carry out partial validation and verification. For example, substantial parts of the UML metamodel have been implemented in the XMF tool [13] from Xactium, which provides the means to generate tools that conform to the MOF-compatible metamodel.

Metamodels have been expressed in B [11], and in PVS [10]. The latter, part of our earlier work, is noteworthy in that it includes heavyweight semantic constraints that consider pre- and postconditions, something which is not considered in other metamodel specifications. This work provided the means to improve reliability via theorem proving technology, but this is in general difficult to apply and requires substantial formal methods expertise.

The main difference between the work we present here and previous work is that the metamodel in Eiffel supports fully automatic conformance checking of models against the metamodel (by running unit tests), as well as fully automatic view consistency checking. Other approaches invariably require a degree of user intervention.

3. Specification-Driven Development of the Metamodel

We now outline the specification-driven development of the BON metamodel, using Eiffel. We started with a brief modelling phase, where we determined the classes that were needed for representing the metamodel. Fortunately, a previous analysis of the metamodel had been carried out in [9], and we had several class diagrams on which to base our modelling. A rough sketch of parts of these class diagrams is shown in Fig. 3.

From these class diagrams, we were able to determine a preliminary set of Eiffel classes that we expected we would need to represent the metamodel in Eiffel, as well as a set of well-formedness rules, captured as class invariants in the class diagrams (see [9] for details of such invariants). However, we did not place too much trust in the correctness of class invariants captured in BON, as they had not been thoroughly validated due to lack of tool support. The validation that had been carried out had been done using the PVS theorem prover [9], and as such the transliteration from class invariants to PVS may have introduced errors. As well, we viewed the classes suggested from the BON version of the metamodel as a preliminary set of suggestions, and expected that parts of the design would need to be refactored, particularly for introducing additional system views (this is discussed further in Section 4).

If we were to develop a UML metamodel in Eiffel, we would start by automatically generating Eiffel class skeletons from EMOF; this would then be built upon in the same way as the BON metamodel constructed in Eiffel.

3.1. Initial design

From the class diagram previously written for the metamodel [9], we automatically generated Eiffel class skeletons. The skeletons consisted merely of class interface details (including names and signatures of routines). These formed the basis of the test-driven development. The contracts that were included in the BON diagrams were *not* automatically translated into Eiffel; in part this is because of the limited support for generating agent-based code in Eiffel, but it is also because we did not entirely trust the invariants included in the BON diagrams as they had not been completely checked with proof tools or by testing.

It is possible to envision extending the code generators for Eiffel to automatically produce the agent-based code; work is proceeding along these lines. This code generation process will have two steps: (a) map the BON contract syn-

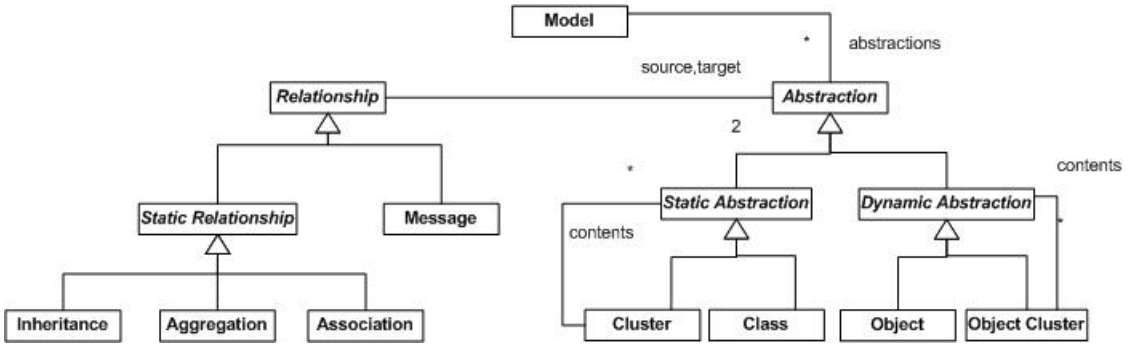

Figure 3. Partial metamodel for BON

tax into Eiffel agent code; and (b) extend the agent code with handlers for dealing with *Void* references.

Once the class skeletons had been produced, we annotated the routines with very simple preconditions, to ensure that arguments to routines were non-*Void*. These contracts would be strengthened, generalised, and refactored in later stages, as tests were written and implemented. We also intended to use strengthened contracts at a later stage for helping to generate unit and integration tests.

Unlike pure TDD, no refactoring of classes occurred during initial design as we had a preliminary, stable set of classes to work with. Refactoring occurred in later stages when the metamodel routines were implemented in Eiffel; these would also occur in adding new system views.

3.2. Writing acceptance tests

We continued the SDD process with writing *acceptance tests*, which would be used to validate individual metamodel constraints. Our aim was to write a set of tests to validate each constraint in the metamodel, e.g., that inheritance hierarchies could not have cycles, that BON clusters could not overlap, that there were no name clashes amongst features of classes. These tests took the form of simple Eiffel programs that *encoded BON models* which either satisfied or failed to satisfy the invariant constraints.

We found that some tests for certain metamodel constraints could not be easily expressed, because the class interfaces we had to work with did not yet possess the features that we need to capture the constraints. However, certain constraints could be tested. For example, given the class interfaces as automatically generated from BON, we could write the acceptance test shown in Fig. 4 to check that BON aggregation relationships do not define cycles. The test is easy to explain: entities in the BON model are declared and created (via Eiffel's **create** statements), the model is populated, and a call to the routine *prepare* initialises all data structures and checks that the metamodel constraints are satisfied. In this case, the test fails (since there is a cycle

```
class ACCEPTANCE_TEST inherit UNIT_TEST
  creation make
feature {ANY}
  no_aggregation_cycles: BOOLEAN is
  local a, b, c: E_CLASS;
        c_to_a, a_to_b, b_to_c: AGGREGATION;
        m: MODEL
  do
    create a.make("A"); create b.make("B");
    create c.make("C");
    create c_to_a; create a_to_b; create b_to_c;
    create m.make;
    ...
    c_to_a.set_source(c); c_to_a.set_target(a);
    b_to_c.set_source(b); b_to_c.set_target(c);
    a_to_b.set_source(a); a_to_b.set_target(b);
    ...
    m.prepare
    Result := true
  end

  make is do
    make_test;
    add_violation_test( agent no_aggregation_cycles );
    to_html("accept.html")
  end
end
```

Figure 4. A model encoding and acceptance test

in the metamodel from class *A* to *B* to *C*) and this is reported in the feedback from ETester. In TDD terms, we have not achieved a green bar.

The acceptance test was written using the ETester framework for Eiffel [7] which provides functionality similar to JUnit, but tailored for Eiffel; it provides specialised support for testing contracts, which JUnit does not. To use ETester, the unit tests must inherit from class *UNIT_TEST*. Each unit test – whether verifying procedures or functions – is then expressed as a routine. Eiffel's agent technology is used to add these unit tests to a test suite, which can then be executed and the results aggregated (via the call `to_html`, which generates an HTML summary of the test results).

We initially wrote six acceptance tests (which were implemented as routines in the unit testing class that inherited

from *UNIT_TEST*), and then commenced testing. Clearly, the tests all failed since no code had been written for the metamodel. The next stage was to start implementing the metamodel, and the routines of the Eiffel class skeletons.

3.3. Declarative coding in Eiffel

It would have been possible to implement the metamodel constraints algorithmically, using a standard graph library. However, our aim was to – as much as possible – write a *declarative specification* of the metamodel in Eiffel, so as to preserve readability, promote understanding, and make it more likely that a correct system has been produced. It is indeed possible to use Eiffel as a declarative specification language, using agents. Thus, the code that implemented the functionality to be checked by the unit tests was written, as much as possible, to make use of agents.

Fig. 5 contains an example of a metamodel constraint written using an Eiffel agent. The constraint establishes that there are no cycles in the inheritance graph in a class diagram; the transitive *closure* is calculated by the call to the routine *prepare*, mentioned earlier. This constraint is implemented as a boolean-valued function which is invoked in the class invariant of *Model*. We emphasise that this constraint is *fully executable*. More examples can be found in [8].

```
no_inheritance_cycles: BOOLEAN is do
  Result := closure.for_all((i1:INHERITANCE): BOOLEAN do
   Result := closure.for_all((i2: INHERITANCE): BOOLEAN do
     -- return true iff i1 and i2 do not form a cycle
     Result := not (i1.source=i2.target and
                    i1.target=i2.source)
   end) end) end
```

Figure 5. No cycles in the inheritance graph

Writing the Eiffel code for the metamodel consisted of three tasks:

1. Adding infrastructure, i.e., data structures to implement the graph structure that underlies a set of BON diagrams. This was carried out using a suite of set data structures, many of which were private attributes of the classes.

2. Adding routines to initialise and manage the data structures.

3. Adding agent-based implementations of the well-formedness constraints.

Each task generates code that must be tested, via a unit test that generally takes the form of a model. The first two tasks are straightforward, but are influenced by the third. Direct implementation of the well-formedness constraints as Eiffel class invariants is the obvious approach, and it is how we proceeded. The scheme for mapping well-formedness constraints to Eiffel agents is described in [8]. When we wrote the agents, we found that we had to refactor the data structures and corresponding management routines from tasks 1 and 2. This was primarily because of reasons of simplicity and understandability: with the infrastructure as originally written, the agent specifications were very complex and difficult to understand. Additional infrastructure was added and the agents refactored, e.g., by introducing intermediate functions, thus achieving a simpler, more understandable specification that in turn was easier to test.

Each metamodel constraint was implemented within the typical SDD process: we implemented the constraint as an agent, refactored the data structures and routines affected by the constraint so as to simplify the agents, and then added contracts to the routines that we had introduced during implementation of the constraint. Implementing each constraint was generally straightforward, because of the declarative nature of agents, but often involved adding additional routines to classes in order to simplify the code. Adding such helper routines was really a part of the refactoring/simplification phase of TDD.

Occasionally, the gap between the unit test and the code needed to implement it was substantial; a particular example was in implementing the constraint to check that routine calls were legal according to the information hiding properties of a class. For example, if, in a preconditions of a routine bound to a message there is a call $o.f(a)$, then the class containing the routine must have access permission to call the routine f of the class of object o. It is easy to write a unit test for this, but the code to implement the constraint in the metamodel is more complex. The complexity arose because messages in BON can be sent to both objects and object clusters (i.e., messages can be sent to more than one object simultaneously). Our original code for implementing the unit test considered only the single-object target case, which is straightforward to capture and test, but as our unit test also contained an object cluster, the testing process failed, and we had to add an additional case to our implementation. This was a good example of where the unit testing process provided feedback that led to an update of the code (though it did not directly lead to a refactoring).

3.4. Infrastructure tests

It quickly became clear that additional unit tests would be needed to verify the additional infrastructure that we needed in Eiffel in order to implement the metamodel. When specifying a metamodel in a language such as PVS or even in BON, well-formedness constraints are expressed on mathematical sets. However, in Eiffel, the constraints are implemented on *object* sets. As with any object-oriented

program, objects must be created and memory allocated on the heap; entities (variables) must be initialised to refer to this memory. Acceptance tests would not directly check such initialisation and memory allocation. This is clearly a downside of implementing a metamodel directly in a programming language. This should be contrasted with the model-driven development of, e.g., the UML metamodel in [1], for which no additional tests would be needed since only mathematical types are used.

Our initial implementation of *Model* and *Class* in Eiffel contained many functions with side-effects; Eiffel guidelines recommend that functions be side-effect free so that they can be used in contracts and to improve overall system reliability. When attempting to write contracts on our infrastructure routines (i.e., accessors, mutators, and data structure initialisation) we encountered these side-effects and thereafter refactored each function with side-effects into an attribute and procedure. The interfaces of *Model* and *Class* were thereby made lengthier, but it then became possible to write simple contracts on these routines, and the routines themselves became simpler (and thus easier to guarantee reliability). This is an example of where the design qualities of the Eiffel language lended themselves to determining which practices to apply in SDD.

3.5. The remaining acceptance tests

With the infrastructure complete, it was possible to write the additional acceptance tests that were impossible to complete earlier. These included acceptance tests to verify that a BON model made use of covariant redefinition of inherited features, that the inheritance graph of a class diagram possessed no cycles, etc. The acceptance tests for these metamodel constraints were generally more complex than those written earlier. The reason for this was that the constraints made use of the metamodel infrastructure, which consisted of implementations of data structures, and this in turn made the agent-based specifications lengthier.

The metamodel was then executed, fully automatically, against the suite of acceptance and infrastructure tests that had been generated. All tests passed, thus providing evidence that the metamodel satisfied the key constraints of the BON language. There is no guarantee that the metamodel is free of bugs, but we have greater confidence in its correctness and robustness because of this testing and because of the inclusion of contracts in the implementation.

4. Observations, Conclusions, and Future Work

SDD was a useful approach to take in building the metamodel in Eiffel. We effectively oscillated between writing unit tests and contracts during development. The decision to switch from writing unit tests to contracts (or vice versa) was made when the unit tests that we were writing became too complex: our view was that it was necessary to keep the unit tests and contracts as simple as possible in order to best achieve high reliability. For example, our initial set of contracts and agents for capturing covariant redefinition were very complex (because we were directly translating the PVS specification of covariance from [9]), so we instead wrote a unit test that included a simple example of covariance. This led us to refactor the contracts and agents for covariance until we obtained a reasonably simple specification that executed correctly against the unit test, and which we thought was more understandable than the PVS version. The improvement in simplicity and understandability in part came because we were able to use sequential composition directly in Eiffel; sequencing must be encoded in PVS. Our conclusion from this refactoring is that, particularly for metamodelling, it is useful to be able to use sequential composition in specifying the metamodel.

There were several key situations where contracts were used to indicate correctness conditions on the metamodel. Most of these contracts were used to capture implementation-oriented correctness conditions, i.e., that non-*Void* references were passed as arguments.

We noted the following observations as we followed the SDD process.

- Minimal refactoring at the class level was carried out; in part, this was because the class model was well-defined and understood before implementation commenced. Refactoring did take place at the method/routine level; this took the form of splitting a single routine into two or three separate routines to achieve a simpler design. The guidance in carrying out this refactoring came from the design of Eiffel itself: the language encourages separating functional methods from procedural methods. A refactoring during the SDD process simplified the agent-based code substantially. Class-level refactoring would be carried out if we were to add a new view (e.g., statecharts) to the metamodel. This is discussed further below.

- The SDD process produced, as a deliverable, an automated test suite that could be used for validating the metamodel, validating any future changes to the metamodel, and as application evidence for arguing the correctness and robustness of the metamodel. Minimal application evidence exists for other published metamodels. Such evidence is essential for reliable critical systems, even if their validity is to be checked via peer review.

- Our use of contracts primarily focused on conditions to check valid arguments to routines. Thus, the design process was more like pure TDD than SDD as

described earlier. We were not particularly surprised by the minimal use of contracts as the metamodel was primarily declarative, and most of the routines in the classes were boolean-valued functions used in class invariants.

- Acceptance tests for the metamodel corresponded to unit tests, since the metamodel was declarative and we were primarily testing class invariants.

- The ability to test frequently led to bugs being caught quickly - particularly in the infrastructure where we initially ran in to problems with down-casts that arose from the use of Eiffel's generic/parameterised classes.

- A key difficulty with using Eiffel to write metamodels is the need to deal with memory explicitly, e.g., *Void* references, cloning of structures. This would be hidden from the user in a true specification language.

In the end, we found that SDD was an ideal way to build a metamodel quickly in Eiffel. The process was fast and feedback was provided quickly and efficiently from the compiler; the entire metamodel was implemented (in a distributed fashion) by two people within the span of two days, and in the process a suite of unit tests for the metamodel was constructed. This suite can be packaged with the metamodel so that other developers can build atop it in extending it to include additional views.

We mentioned earlier that the tests that were developed during the SDD process were *encodings* of BON models, and that running the tests had the side-effect of checking that the models conformed to the BON metamodel. In principle, this provides a fully automatic way of carrying out metamodel conformance checking including contracts, something that, to our knowledge, no CASE tool currently supports. This can be accomplished in practice by (a) implementing a code generator that transforms BON models into unit tests, and (b) providing a unit testing framework for running the unit tests and documenting the results. Tools, particularly ETester, support (b). Implementing the code generator in (a) is not difficult and is underway.

Our current work on the metamodel itself is proceeding along two lines. We have recently extended the metamodel to include *contracts* on routines, so that we can fully check the consistency of class and dynamic diagrams that include preconditions and postconditions. It is essential to support this level of consistency checking to make the metamodel fully suitable for building reliable systems. However, the extension is not easy to use from the perspective of the user. We are thus exploring an alternative solution, via a mapping to a theorem prover, and we expect to make use of .NET's reflection capabilities for this. A second line of work is focusing on refactoring the metamodel at the class level in order to introduce additional views, particularly statecharts. The introduction of statecharts will require view consistency constraints between objects (dynamic abstractions) and states, as well as constraints between states and classes. Since such constraints will span two branches of the class hierarchy in the metamodel, a refactoring will be necessary. This refactoring will make use of Eiffel's event-handling libraries and mechanisms, in order to guarantee – by construction – consistency between the statechart view of a system, and the class view. Finally, we have recently implemented parts of EMOF in Eiffel, and will continue this work to further explore Eiffel's suitability as a general metamodelling language.

References

[1] 2UWorks-Consortium. Unambiguous UML revised submission to UML 2 superstructure RFP, 2003. www.2uworks.org.

[2] S. Ambler. Extreme testing. *Software Development*, 11(5), June 2003.

[3] K. Beck. *Test-Driven Development*. Addison-Wesley, 2000.

[4] B. Meyer. *Object Oriented Software Construction, Second Edition*. Prentice Hall, 1997.

[5] Object Modelling Group. UML Standard Guide 1.5, 2003.

[6] J. Ostroff, D. Makalsky, and R. Paige. Agile specification-driven design. In *Proc. Extreme Programming 2004*. LNCS, Springer-Verlag, June 2004.

[7] J. Ostroff, R. Paige, and D. Makalsky. ETester: a contract-aware and agent-based testing framework for Eiffel, submitted. June 2004.

[8] R. Paige, P. Brooke, and J. Ostroff. Executable metamodelling and consistency checking with Eiffel. July 2004. Submitted.

[9] R. Paige and J. Ostroff. Metamodelling and conformance checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*. LNCS, Springer-Verlag, 2001.

[10] R. Paige, J. Ostroff, and P. Brooke. Theorem proving support for view consistency checking. *L'Objet*, 9(4), 2003.

[11] J. Sourrouille and G. Caplat. A pragmatic view about consistency checking of UML models. In *Workshop on Consistency Problems in UML-Based Software Development*, 2002.

[12] K. Walden and J.-M. Nerson. *Seamless Object Oriented Software Architecture*. Prentice Hall, 1995.

[13] Xactium. XMF user guide prerelease version 0.1, 2004. www.xactium.com.