A Test-Based Agile Approach to Checking the Consistency of Class and Collaboration Diagrams

Richard F. Paige
Department of Computer Science, University of York,
Heslington, York, YO10 5DD, U.K.
paige@cs.york.ac.uk

Jonathan S. Ostroff
Department of Computer Science, York University,
Toronto, Ontario, Canada, M3J 1P3.
jonathan@cs.yorku.ca

Phillip J. Brooke
School of Computing, University of Plymouth,
Drake Circus, Plymouth, Devon, PL4 8AA, UK.
philb@soc.plym.ac.uk

June 30, 2003

Abstract

The problem of checking the consistency of different views of a system is presented, and a test-based approach – which is being implemented in the context of an object-oriented CASE tool – is described using the Eiffel language. The approach is novel in that it supports design-by-contract mechanisms, including preconditions, postconditions, and class invariants, that may be embedded within views. The agile development process in which the approach is intended to be used is outlined.

1 Introduction

Modelling is increasingly being used in the development of systems. The Unified Modelling Language (UML) [3], a *de facto* standard language for describing systems, presents a suite of modelling notations and techniques that can be used for describing systems from a number of different perspectives. Once models are constructed, they can be used for code generation, for static analysis and simulation, and for documenting both the design process and the management process. Modelling in particular provides a unifying suite of techniques for system development. There is increased interest in automating *model-based testing*, particularly in the high-integrity systems community,

from the perspective of reducing the costs associated with incremental certification of systems. Model-based testing can be applied in a number of ways: via simulation, where an execution semantics is provided to models, and a simulation engine is used to step through traces of behaviour; and, by automatically or semi-automatically generating test cases, expected results, and test drivers from models. These can thereafter be used to verify and validate the code generated from the models.

A particular challenge in model-based testing using UML is *view consistency* (discussed in further detail in Section 2). In short, when modelling a system using UML, several different models are constructed, representing different aspects or views of the system, e.g., the system's static (compile-time) structure, its run-time behaviour, the behaviour of individual objects in the system, its deployment over one or more hardware units, etc. The information contained in these models may overlap; as well, the models may be independently conceived and constructed by separate teams of developers. This situation can easily lead to *model inconsistency*, which may only be detected when code has been automatically generated and the testing process commences. Model inconsistency may also arise due to misunderstandings of requirements, mistakes in constructing designs, and syntactical or semantic errors in writing the models themselves. It is desirable to be able to detect model inconsistency at an early stage, for a number of reasons.

- In general, the earlier we are able to detect problems in models, the better, since it is less likely that the problems will propagate to code or customer deliverables such as documentation.
- Modifying models is often easier than modifying code, given that models usually omit much detail.
- When product families and product line models are being constructed, we may be implementing a number of different product instances from a single set of models, and thus errors or flaws in models will affect many concrete products. Dually, the cost of tracking down a flaw or error in a product line model can be amortised over the set of product instances that are expected to be generated.

It is therefore advantageous, from a cost perspective, to try to detect inconsistencies during modelling, and to provide a process for doing so.

In this paper, we present a scheme for test-based view consistency checking, intended to be used during lightweight or *agile* development processes where modelling is applied. Mechanisms for executing tests are to be generated from the models that are constructed during agile development.

Agile processes – such as Extreme Programming, SCRUM, and feature-driven development – are defined in terms of a number of principles and practices, focusing on simplicity, customer involvement, early and constant delivery of working code, and the ability to handle changing requirements. When modelling is involved in agile development, it is applied minimally, with the knowledge that the models themselves are not deliverables, while the code that is derived from the models is the critical deliverable. Agile processes can be contrasted with heavyweight approaches, e.g., the Rational Unified Process [7], wherein a set of models and documentation is specified for delivery,

and a rigorous sequence of steps is to be followed, ideally to enable traceability and audits.

In this paper, we focus on check the consistency of two widely used views in UML: class diagrams (for modelling the static structure of systems) and collaboration diagrams (for modelling message passing behaviour). There is nothing specific in our discussions that constrains our approach to UML; in fact, we are currently implementing the technique described in the sequel in a CASE tool that supports a different suite of modelling languages, those of BON [17]. A novelty with our approach is that we consider design-by-contract additions to modelling. Specifically, we treat models that include pre- and postconditions of operations, and invariants of classes, and include tests that check the relationship between contracts of clients and suppliers in models. Much of the view consistency work related to UML does not treat contracts; some exceptions are discussed in the next section.

We start by describing some of the key problems with view consistency in UML, and also define some basic UML notation. We then discuss related work, including a *heavyweight* yet theoretically complete approach to view consistency checking that requires theorem proving support, in order to deal with contracts. Next, we explain how a collaboration diagram in UML can be extended, in a trivial way, to support a test-based approach to consistency checking. In doing so, we outline a development process – that is both test-driven *and* agile – in which the test-based approach to consistency checking is intended to be used. We describe an algorithm for consistency checking, and thereafter touch on how the algorithm is currently being implemented in a CASE tool.

2 View Consistency in UML, and Related Work

Consistency checking of the documents that are produced during software development is a difficult task. Consistency checking has been discussed in the context of work on multi-viewpoint specification [11] and combining specifications [18] written in different formal and semiformal languages. It is especially challenging and relevant when using the modelling language UML [3], where five different and potentially conflicting views of a software system of interest can be independently constructed. The intent of using multiple, disparate views is to describe different aspects of a system in the most appropriate way. The different descriptions must at some point be combined to form a consistent single model of the system that can be used to produce executable program code. The process of combining the descriptions should identify inconsistencies – typically using counter-examples that are useful as test cases – that need to be resolved by the developers before executable code can be produced.

In general, consistency checking of development deliverables involves the use of constraints, algorithms, and tools to check that information described in one deliverable (e.g., source code, a UML model) is not contradicted by information described in another deliverable. In a setting where formal specifications are available, this can be reduced to the problem of checking that a conjunction of predicates – each a formal specification of a deliverable – is satisfiable. In general, complete formal specifications of models are usually unavailable, and thus the problem of consistency checking is

made more complex and challenging.

Specific to UML is the problem of checking that the following views are consistent: the class diagram view (specifying static, structural aspects of a system), the dynamic diagram view (specifying the message passing behaviour of systems in response to actor-generated events), the statechart view, the deployment view, and the use case view.

Some checking can be done at the level of the *metamodel*, the set of constraints that establish what is a semantically valid set of UML models. The metamodel establishes, for example, that messages are sent between objects that have their classes associated in a class diagram. But the metamodel does not cover all constraints, particularly those related to operation behaviour and contracts. Checking the consistency of the contracts with respect to other views is significantly more challenging.

To illustrate some of the problems, consider the example diagram shown in Fig. 1 and Fig. 2. The class diagram depicts the static structure of part of a simple adventure game; the collaboration diagram depicts the message passing behaviour of a scenario in the game (a maze is populated with rooms and objects). Operations in the class diagram can be annotated with OCL constraints, depicting pre- and postconditions. The preconditions state constraints on when operations can be invoked by clients, whereas the postconditions state constraints on what results can be assumed by clients after the operation has terminated its execution.

Messages in a collaboration diagram correspond to operations in the class diagram. Messages are triggered only under conditions stated in their precondition. So, for example, message 4, $connect$, can only be sent if its precondition is $true$. This is only the case if the preceding messages 1, 2, and 3 enable it, i.e., if the state changes specified in the postconditions of messages 1, 2, and 3 are sufficient to enable message 4.

In general, view consistency checking can range from checking simple syntactic constraints, to lightweight semantic constraints (e.g., related to naming conventions), to heavyweight constraints that require semantic and run-time properties to be checked. In general, it will be desirable to be able to project subsequences of scenarios out from collaboration diagrams in order to carry out consistency checking. This will be done based on risk assessment.

The workshop on consistency checking in the context of UML [6] demonstrates state-of-the-art techniques that consider different views (particularly static and dynamic) and different lightweight and heavyweight techniques for implementing the checking. Some of these approaches require use of new, speciialised tools – such as xlinkit [5] – or languages, e.g., B [9] or PVS [14] – for expressing the well-formedness constraints that establish view consistency. By contrast, our approach does not make use of any additional tools – beyond those used to construct the object-oriented models, as well as a compiler – for carrying out consistency checking. Some work has been carried out on mapping UML diagrams into formal specifications – e.g., the B language [9] – and thereafter using tools for the formal notation to carry out consistency checks. This has the advantage of being able to use the (typically richer) formal notation for a variety of types of reasoning, but it also requires that the mapping from the UML diagrams into formal specifications be checked for soundness.

A theoretically complete, though heavyweight approach to view consistency checking makes use of theorem proving technology [12, 14]. The basic approach used is to

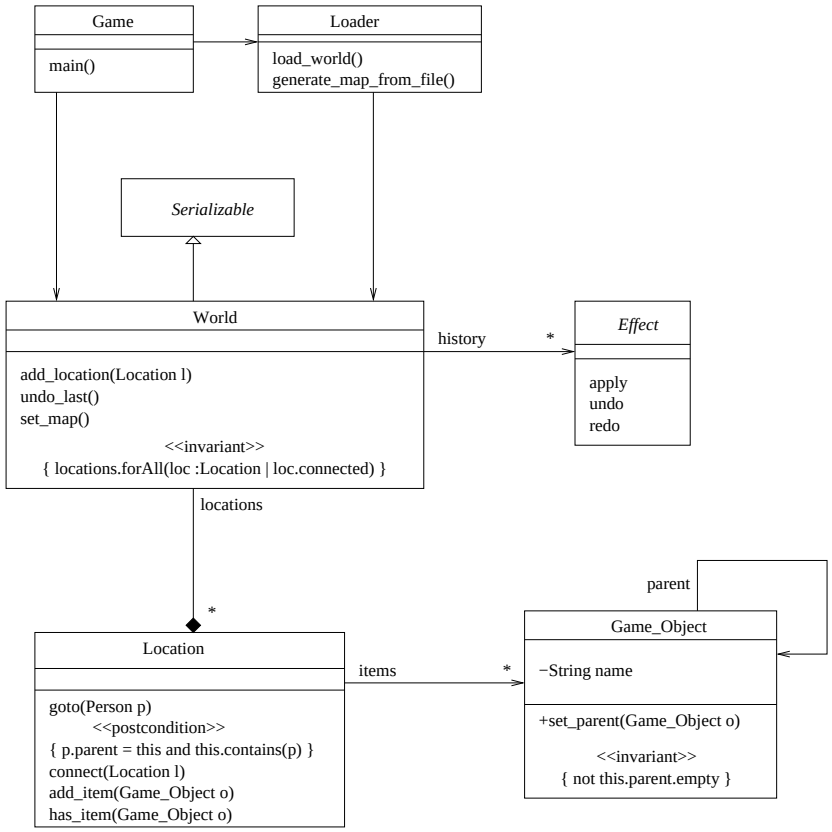

Figure 1: Class diagram in UML

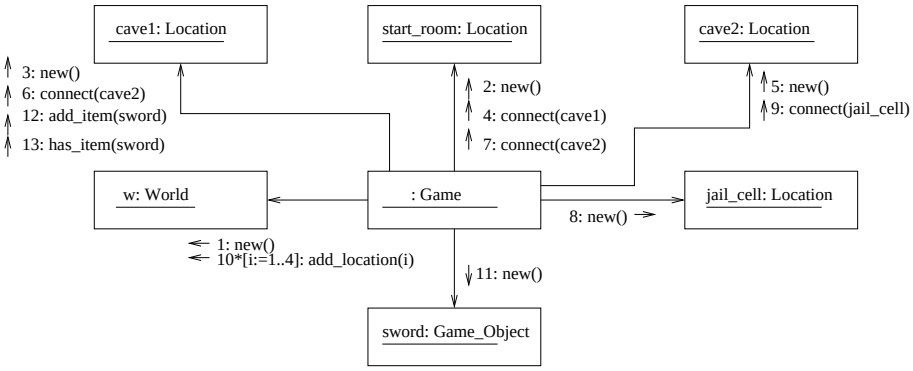

Figure 2: Collaboration diagram in UML representing a scenario for the partial population of a maze

specify collaboration diagrams and class diagrams in the PVS specification language, and then to specify axioms defining view consistency: specifically, the conditions described informally in the preceding section that require previous subsequences of messages to enable the next element in a message sequence. This has been done in [14], and an example of using the PVS prover to check consistency has been documented. This is, of course, a difficult process – though the mapping from UML models to PVS specifications can be automated – and it requires a great deal of knowledge and experience with the PVS prover to get the proofs of consistency to complete. On the other hand, the approach is complete in the sense that it will detect any inconsistencies, and will return counter-examples in the form of unprovable steps in a proof. Interpreting these unprovable steps, though, is oftentimes difficult, requiring detailed understanding of PVS's syntax, and the formalisation of metamodel rules in PVS. A similar approach is taken by the PRUDE toolset [16], which presents an integrated toolset that, amongst other things, generates PVS automatically from UML models, particularly statecharts. The tool aims to provide an integrated verification environment for UML, but also provides some lightweight consistency checking facilities by implementing parts of the UML metamodel.

Lighter-weight approaches to consistency checking are desirable and have been explored, but a limitation with many of these approaches is that they do not consider contracts. The work of Bhaduri and Venkatesh [2] suggests the use of message sequence charts and model checking for view consistency checking. They express the semantics of the object life cycle and scenario views as a labelled transition system, thus enabling the use of a model checker to identify inconsistencies. The advantage of this approach is that the model checking will be automatic; however, there are limitations on what can be expressed in terms of properties and models. UML model consistency is also checked via the Sherlock tool [15], wherein actions, models, and a knowledge base are combined. The latter approach is particularly promising as it also considers extensions to profiles.

The authors are currently investigating two complementary approaches to the problem: one involving model checking, the other involving testing technology in the context of an agile development process. We discuss the latter in the next section. The advantage of this approach is that it requires no new tools or technologies – beyond a CASE tool and compiler – in order to carry out substantial consistency checking.

3 Lightweight Checking using Testing

One of the key practices of the Extreme Programming (XP) [1] class of software development approaches is *test-driven development.* Test cases are written before code is produced, and each modification or extension to the system is checked against the test cases before the project repository is updated. Certain so-called *agile* methodologies attempt to generalise XP to include brief, focused modelling phases, while retaining the emphasis on testing. A key question in such methodologies is how to balance modelling with test-driven development? One approach to this is to automatically generate test drivers and test cases from models. Thus, development would start with lightweight modelling of requirements, or system architecture (or whatever other elements of the

system are of interest to developers), as well as models of typical scenarios of use. These latter models could then be used for automatically generating test drivers and test cases. In particular, the generated test drivers and test cases could then be used for establishing the consistency of the different models generated during the lightweight modelling phase.

The basic approach that we are proposing is to generate test *drivers* from UML collaboration diagrams. When appropriate during agile development – e.g., when making revisions to models (refactoring), or when implementing selected methods in class diagrams – code can be generated automatically from UML class diagrams in a suitable language, such as C++, Java, or Eiffel. The test drivers can then be executed against the generated code – perhaps augmented with additional statements by programmers – to check for consistency. If the test drivers run successfully, we infer that the code and the test drivers, and hence the collaboration diagrams and class diagrams, are consistent[1].

This is an indirect consistency check: effectively, the test drivers are viewed as a *refinement* of collaboration diagrams, and the code as a refinement of class diagrams. This can be depicted as in Fig. 3. The stereotype **¡¡derive¿¿** on the dependencies indicates that the source of the dependency can be automatically derived (and is therefore consistent) with the target. The stereotype **¡¡consistent-refine¿¿** indicates that a consistency checking process must take place. Note that the dependency between the executable work products (the source code and the test driver) is a refinement of dependency between the models; the implementation of the consistency checking algorithm must guarantee this.

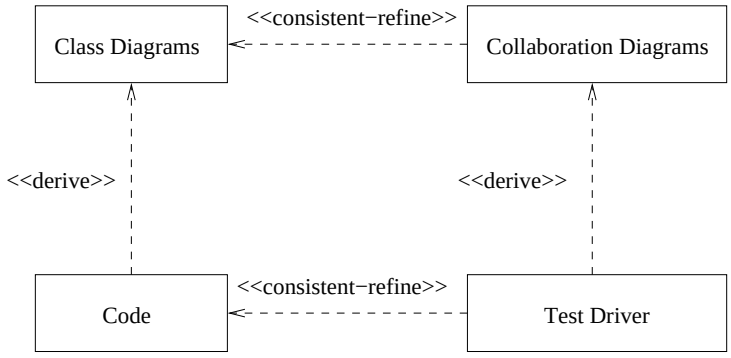

Figure 3: Refinement structure of consistency checking scheme

We are currently working on implementing the consistency checking scheme in the framework of the Eiffel programming language [10]. We have chosen Eiffel because it is object-oriented, because it provides built-in support for assertions, and because it has a suite of extensible tools, e.g., BON-CASE [13]. Thus, we will automatically generate Eiffel code from UML class diagrams. Generating test drivers from collaboration diagrams is slightly more complex. To do this, we need to extend the collaboration diagram with two additional pieces of information: initial and final states. The initial state

[1]Assuming the correctness of the code and test generation algorithms.

specifies the state of the objects in the collaboration before the first message is sent. The final state specifies conditions that must be true of the objects in the collaboration after the last message has been sent; this might be a termination condition, or a sanity check, of some kind.

Given that a collaboration diagram typically refines a scenario of use in an object-oriented model, it is useful to be able to specify conditions or properties that should be true when a scenario ends; this is the purpose of the final state: when the sequence of message calls in the collaboration diagram ends, the final state should be reached. We currently require that the initial and final state specifications be in the Eiffel assertion language, and thus that they are machine checkable and executable. They will therefore be expressed as **check** statements (similar to C's `assert`) in Eiffel. Given the expressive power of Eiffel – especially with its *agent* notion, which enables quantifiers over finite domains to be implemented efficiently [10] – this is a not unreasonable restriction to work with.

The test driver that is to be generated from the collaboration diagram takes the form of a single Eiffel class $TEST\_DRIVER$. The class possesses a **creation** routine $make$ that is executed when $TEST\_DRIVER$ is instantiated. The creation routine executes a sequence of feature calls, generated according to the sequence of messages appearing in the class diagram. If guards appear on messages, the feature calls will be prefixed with suitable **if-then-else** structures, or **loop-end** structures in the case where an iterative multiplicity constraint is provided.

Two challenges arise with the refinement process of collaboration diagrams into an Eiffel test driver.

1. **new** messages: these indicate the creation of a new object of the type of the recipient of the message. Each object in the collaboration diagram is being mapped into an entity in Eiffel. However Eiffel classes may have many constructors/creation routines, and unlike languages such as C++ and Java, constructors can have any name. Even when dealing with a language like C++ and Java – which force constructors to have the same name as the class – will require dealing with a choice from multiple constructors. Thus, user assistance will be necessary, in general, to select the appropriate constructor to execute as a result of a **new** message. In the test driver generation algorithm, presented shortly, user assistance is obtained through the use of $select$ features, e.g., `select_create_feature`. This user assistance simply takes the form of selecting a feature from a specified, automatically generated list.

2. **Underspecified messages:**a message in the collaboration diagram may be annotated with the name of the feature that should be called in response; in this case, the feature call is added directly to the test driver. In general, though, a message can be underspecified: names or types of arguments may not be provided, or messages may be overloaded (especially in C++), or the specific target of the message may not be precisely constrained. The last case arises when a multiobject appears in the collaboration diagram. In all of these cases, it is best to ask for assistance from the user, to indicate which feature to call in response to a message, or to specify the name of an object that should be the recipient of the message. This is the purpose of `select_feature` in Fig. 6.

```
class MESSAGE feature
  source, target: OBJECT
  number: INTEGER
  guard, multiplicity: STRING
  has_guard, has_multiplicity: BOOLEAN
end -- MESSAGE
```

Figure 4: Messages in a collaboration diagram

It is important to point out that the generated test driver is not sufficient for full test coverage of the system – it is to be used for checking view consistency. For test coverage, the collaboration diagram would have to be refined to include exceptional conditions and cases – for example, what messages would be sent in Fig. 2 in the case where a room cannot be added to the maze? Providing that modellers are able to extend their collaboration diagrams to include exceptional cases, the approach can be used to generate test drivers for functional and integration testing as well. However, we suggest that in practice this approach might not be sensible, since the collaboration diagrams will quickly become large, cumbersome, and difficult to understand. A pragmatic approach might be to extend the generated test drivers to carry out more detailed functional and integration testing, i.e., to modify the test drivers directly.

To describe the test driver generation algorithm, we need to refer to the metamodel for UML; the metamodel, implemented in a CASE tool, will be used to (i) provide the infrastructure for representing class and collaboration diagrams and their annotations; and (ii) to provide basic consistency checks (e.g., freedom from name clashes). We assume that these basic consistency checks have been carried out, particularly:

- each object in the collaboration diagram has a corresponding instantiable class in a class diagram;
- each message has a corresponding operation/feature for invocation.
- the class and collaboration diagrams are syntactically valid.

All of these checks are trivial to implement and verify.

To specify the algorithm, we assume the following definitions of collaboration diagram and message, extracted (and simplified) from the UML metamodel. We express this part of the metamodel in Eiffel, and we will use Eiffel to specify the algorithm as well. First, the definition of message.

Each message has a source, target, and number, and may optionally have a guard and a multiplicity constraint (which are represented as strings). Next, the definition of collaboration diagram.

We can now use the specifications in Figs. 4 and 5 to specify the test driver generation algorithm. This appears as part of the definition of class $GENERATOR$.

The algorithm works, informally, as follows. First, declaration header information is generated for the test driver (i.e., name of driver class and standard Eiffel syntax), then declarations for objects in the collaboration diagram. A **check** statement (i.e., an `assert`) is generated to test the validity of the initial state of the system. Then,

```
  class COLLABORATION_DIAGRAM feature
    messages: ARRAY[MESSAGE]
    objects: SET[OBJECT]
    initial, final: STRING
 end -- COLLABORATION_DIAGRAM
```

Figure 5: Excerpt of collaboration diagram metamodel

```
class GENERATOR feature
  ...
  generate_test_driver(c:COLLABORATION_DIAGRAM) is
  local
     i: INTEGER;
     m: MESSAGE;
     f: FEATURE;
  do
     generate_driver_header;
     generate_declarations(c.objects);
     generate_check_statement(c.initial);
     from i:=1
     until i>c.messages.length
     loop
       m:=messages.item(i);
       if m.has_guard then
         generate_ifthen(m.guard)
       end
       if m.has_multiplicity then
         generate_loop(m.multiplicity)
       end
       if m.feature.is_create then
         -- return specific creation routine
         f:=select_create_feature(m.target);
       else
         -- return specific feature with arguments
         f:=select_feature(m.target);
       end
       generate_feature_call(m.target,f);
       generate_close_branches;
       i:=i+1;
     end
     generate_check_statement(s.final);
     generate_driver_footer;
   end
   ...
end -- GENERATOR
```

Figure 6: Algorithm for generating test driver from sequence diagram

the messages in the diagram are looped over. Each message is tested to see if it has a guard or multiplicity constraints, and suitable Eiffel **if-then-else** statements or **loop** statements are generated. Then a feature call is generated; this may require consulting the user to select either a default **create** statement or one of several possible **create** routines. Finally, all branches in the generated code (i.e., loops or selections) are closed, and a final check is generated on the final state of the system. The algorithm is interactive in the general case, but if users have specified features for each message (i.e., they have refined away nondeterminism) then the test driver generation is automatic. Any interactions would be limited to selecting a method from a drop-down list of options.

Running the test drivers, and hence carrying out the consistency checking, is the simple matter of compiling and executing the generated code, including the test driver, using the test driver as the root class [10] of the system; the root class contains a method from which execution must start. If using an Eiffel compiler such as EiffelStudio, this simply requires modifying an Ace file (akin to a Makefile) in order to tell the compiler which is the root class.

3.1 Example

We illustrate the use of the test driver generation algorithm by showing the result of applying it to the collaboration diagram in Fig. 2. To carry out the test driver generation, an initial state must be specified. In the scenario specified in Fig. 2, a number of locations and objects in a maze are created (via the new) message and connected by the root object, $Maze$. Thus, a suitable initial state is just $true$, since the root object will be created by the system itself.

The generated code will appear as shown in Fig. 7.

Consistency between class and collaboration diagram can then be evaluated by compiling and executing code generated automatically from the diagram in Fig. 1. If a precondition, postcondition, or invariant is $false$ during execution of the test driver, then the Eiffel run-time system will indicate to us which operation call has failed, and thus where an inconsistency arises. If using an integrated development environment like EiffelStudio, which provides an integrated debugger, then the debugger can be used to isolate statements and objects that lead to inconsistency.

In the case of the collaboration diagram in Fig. 2, generation of the test driver can be done automatically, and in a single pass over the message sequence in the diagram. This is because all messages in the diagram are linked to specific, effective features in the class diagram, and no user intervention is needed to select an operation to implement a message, given a list of choices.

This technique can be used to support an agile style of system development. Initially, it is likely that the class diagrams that are written will not be linked to any code, i.e., the operations may be annotated with pre- and postconditions, but will not be implemented except perhaps with empty operation bodies. Thus, the process of compiling and executing the test drivers will not provide substantially meaningful results. The agile developer will then proceed, as is typical, by implementing methods bit by bit, and re-running the test driver. Gradually, pre- and postconditions will be satisfied, as methods are implemented, and when the test driver runs to completion, the developer will

```
class TEST_DRIVER creation make
feature {ANY}
  cave1: LOCATION
  cave2: LOCATION
  start_room: LOCATION
  jail_cell: LOCATION
  w: WORLD
  sword: GAME_OBJECT
feature {ANY}
  make is
  local i:INTEGER
  do
    check true;  -- initial state
    create w;
    create start_room;
    create cave1;
    start_room.connect(cave1);
    create cave2;
    cave1.connect(cave2);
    start_room.connect(cave2);
    create jail_cell;
    cave2.connect(jail_cell);
    from i:=1 until i>4
    loop
      w.add_location(i);
      i:=i+1;
    end
    create sword;
    cave1.add_item(sword);
    check cave1.has_item(sword); -- final state
  end
end
```

Figure 7: Generated test driver refinement of collaboration diagram

have at least some initial evidence that quality improvement has been made in terms of making the different views consistent.

4 Discussion and Conclusions

We have outlined the problem of view consistency checking in UML, and outlined several approaches to the problem. The contribution of this paper is a general algorithm for generating test drivers semi-automatically from collaboration diagrams in the context of an agile development process that makes use of minimal, targetted modelling. Compiling and executing the test drivers results in the consistency of the collaboration diagram and class diagram being checked, and counter-examples (in the form of error messages and exceptions) being produced in the case where inconsistencies are identified. The approach deals with pre- and postconditions, and is compatible with agile approaches to development.

Work is underway at implementing the algorithm in the framework of the BON-CASE tool [13]. This tool supports a subset of UML (specifically, the diagrams used in this paper, as well as use case diagrams), and integrates an Eiffel-aware editor. In particular, the tool has an extensible code generation component that is designed explicitly to enable new code generators to be added.

An additional point remains to be mentioned: that of message refinement. A message f sent to object o in a collaboration diagram may in fact need to be implemented by a sequence of feature calls when it comes time to implement the resulting program. In this sense, a message is an abstraction of a part of a program. To check for the consistency of views in this situation, it will be necessary to compute a transitive closure of the links in the collaboration diagram – based upon feature calls appearing in the contracts. We have not discussed this issue in this paper due to space restrictions, but our ongoing work on implementing the algorithm and approach does take this issue into account.

One direction for future work is to link use case diagrams into the diagram from Fig. 6. During development, particularly following the Rational Unified Process, individual use case scenarios are refined using collaboration diagrams. Given a suitably precise specification of a use case scenario, the algorithm and scheme described above could be extended, and consistency between the three diagram types could be tested and checked.

Another direction is to integrate the consistency checking algorithm with the *E-Tester* framework [8] for Eiffel. This framework provides automated support, via Eiffel's *agent* mechanism, for carrying out class-level, cluster-level, and system-level testing.

References

[1] K. Beck. *Extreme Programming Explained*, AWL, 1999.

[2] P. Bhaduri and R. Venkatesh. Formal consistency of models in multi-view modelling. In [6].

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The UML Reference Guide*, Addison-Wesley, 1999.

[4] J. Clark, S. Stepney, and H. Chivers. Breaking the model, submitted July 2003.

[5] C. Gryce, A. Finkelstein, and C. Nentwich. xlinkit: lightweight consistency checking for the UML. In [6].

[6] Z. Huzar, L. Kuzniarz, G. Reggio, and J. Sourrouille. *Proc. Workshop on Consistency Problems in UML-Based Software Development*, Blekinge Institute of Technology Research Report 2002:06, September 2002.

[7] P. Kruchten. *The Rational Unified Process: an Introduction (Second Edition)*, Addison-Wesley, 2000.

[8] D. Makalsky. E-Tester: a comprehensive testing framework for Eiffel. Available at www.ariel.cs.yorku.ca/∼eiffel. 2002.

[9] R. Marcano and N. Levy. Using B formal specification for analysis and verification of UML/OCL models. In [6].

[10] B. Meyer. *Object-Oriented Software Construction* (Second Edition), 1997.

[11] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specification. *IEEE Trans. Software Engineering* 20(8), August 1994.

[12] R.F. Paige and J.S. Ostroff. Metamodelling and conformance checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*, LNCS 2029, Springer-Verlag, April 2001.

[13] R.F. Paige, L. Kaminskaya, J.S. Ostroff, and J. Lancaric. BON-CASE: an extensible CASE tool for formal specification and reasoning. *Journal of Object Technology* 1(3):65-87, Special Proceedings of *TOOLS USA 2002*, August 2002.

[14] R.F. Paige, J.S. Ostroff, and P.J. Brooke. Theorem Proving Support for View Consistency Checking. To appear in *L'Objet*, 2003.

[15] J.-L. Sourrouille and G. Caplat. Checking UML Model Consistency, in [6].

[16] I. Traore. PRUDE 1.2 User Manual. Available at http://www.isot.ece.uvic.ca/manual.html. Last visited June 2003.

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

[18] P. Zave and M. Jackson. Conjunction as Composition, *ACM Transactions on Software Engineering and Methodology* 2(4), October 1993.