

A Proposal for a Lightweight Rigorous UML-Based Development Method for Reliable Systems

Richard Paige and Jonathan Ostroff

Department of Computer Science, York University, Toronto, Ontario, Canada.
{paige, jonathan}@cs.yorku.ca

Abstract: A lightweight UML-based software development method for building reliable software systems is proposed. It attempts to combine the coding emphasis of Extreme Programming with the utility of modelling, while offering a counterpoint to Extreme Modelling. The method is built atop of a subset of UML, making use of contracts for documentation and for run-time (and potentially static) checking. Rules are given to establish consistency of views of a system, and a proposal for a tool prototype that implements the diagrams and which helps to establish their consistency is outlined. The key elements of a process, which emphasizes rapid production of code and test drivers, are also outlined.

1 Introduction and Background

Our focus in this paper is the domain of *reliable software systems*. A reliable system is one that has both *correctness* and *robustness* requirements. A correct system is one that satisfies its specifications. A robust system behaves sensibly when it receives unexpected input (e.g., from its environment) or if there is a failure in a component with which the system interacts. We propose a method for constructing such systems, building atop a core of commonly used UML diagrams, and providing tool and process support. The method attempts to combine the utility of modelling for documentation purposes while emphasizing coding. As such, it can be considered a counterpoint of Extreme Modelling [Am01].

One technique that has proven successful in building reliable object-oriented systems is design-by-contract (DbC) [Me97]. The premise of DbC is very simple: methods of classes should be documented with preconditions (expressing constraints on when the methods can be called) and postconditions (expressing the results delivered by the method when it successfully finishes execution). As well, DbC requires that classes themselves be documented with so-called *invariants*, which specify constraints that instances of the class should obey at well-defined stable points in time. DbC is available and has been used in a variety of different settings: in programming languages like Eiffel, Java¹ with iContract [Kr98], C++, etc.; and in modelling languages such as UML, BON, and JML. DbC addresses the construction of reliable systems directly by capturing those constraints that indicate when a program or model is correct (it satisfies the constraints), and what must be done to make a program robust (the states outside of the pre- and postcondition must be dealt with). It is a documentation technique, where specifications on how to use models or

¹ As announced at JavaOne 2001, Java 1.4 will support built-in assertions.

components is kept within the component itself; thus, documenting is integrated into the coding process. DbC is also a *lightweight* formal method: developers use as much or as little formalism as needed in specifying contracts; they can use executable specifications (e.g., contracts that are boolean expressions in a programming language) or more expressive specifications (e.g., using predicate calculus); and it is supported by tools, such as compilers and static checkers.

This paper proposes a software development method, including a proposal for prototype tool support, for building reliable systems. The method makes use of UML, and also focusses on rapid development of executable code. Its original aspects include its emphasis on coding while allowing the use of modelling for (i) deriving consistent graphical views of systems, and (ii) as the means to quickly generate code skeletons. The thread running through the method is the use of contracts, when modelling systems and when coding them. Contracts are part of models and code (where we require that they be at least run-time checkable), and are a form of documentation. The method thus does not distinguish between documenting and coding.

Code is central in the proposed method. The method emphasizes producing reliable contract-annotated code as quickly as possible. As such, developers are able to start producing code deliverables immediately, in much the same way as Extreme Programming [Be99]. Contracts, in particular, can make testing easier to do, since they can be used as a source for extracting test cases automatically, they can be used for run-time and static checking, and they can be used to generate more expressive messages regarding tests that produce unexpected or erroneous results.

The proposed method also applies modelling wherever developers think it will help. In terms of modelling, the method makes use of a subset of the core of UML, including class and collaboration diagrams. We explain the specific rôles in which modelling can be used in the sequel, but in a nutshell, we are proposing to allow developers to model or code whenever they need to, and to provide methodological support (via algorithms and tools) to establish consistency of code and models.

Fig. 1 depicts the basic deliverables of a project – models, source code, and test drivers – produced by the method, and their informal relationships. This diagram will be refined in Section 3. Source code and models are consistent (in a sense that we make more precise in the sequel), and test drivers are consistent with models as well.

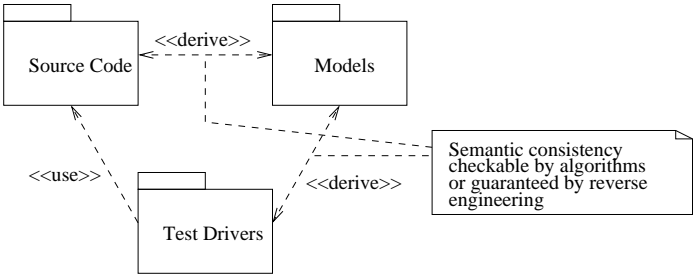


Fig. 1. Relationships between deliverables in the method

The method has the following characteristics.

- It is specifically aimed at building reliable systems. As such, it emphasizes the use of design-by-contract, which has been shown to be successful in constructing such systems. Contracts are used both in executable code (e.g., using an efficient, run-time checkable assertion language such as that provided with Java 1.4 or Eiffel) and in models (e.g., using OCL). Contracts are viewed as documentation, and thus the process of producing documentation is integrated with the processes of coding and modelling.
- The method can be seen as a *lightweight* formal method, because of its use of design-by-contract. It is lightweight in the following senses: formality, via pre/postconditions and class invariants, is available when developers choose to apply it (in this sense, the method has similarities to Catalysis [DW98], wherein formality is used as little or as much as is necessary); and executable specifications can be written (e.g., by writing contracts using a boolean expression language) and can be checked either at run-time or statically, using a tool such as ESC [Es00].
- There are no modelling phases per se in the proposed method. Rather, models are produced by developers when they are useful for the tasks at hand, e.g., when trying to make changes to the architecture of a system, when attempting to explain things to others, or when trying to understand vague or imprecise requirements. As we shall discuss, models can either be *generated automatically* from source code or test drivers, or they can be *checked for consistency* against source code or test drivers.

Our proposed method specializes a commonly used subset of UML to make it more appropriate for reliable software development. The specialisation neither adds nor modifies diagramming elements. Rather, it makes use of existing diagram elements – specifically, class diagrams, collaboration diagrams, and use-case diagrams – in a careful way. It integrates diagrams via use of constraints and tools, and proposes general ways to use diagrams within the context of a process supported by a CASE tool, with precise requirements for its functionality.

1.1 Organisation of the paper

We commence by discussing two different perspectives on modelling and how modelling can be integrated into development processes that emphasize rapid production of code. We then describe the proposed method over three sections: Section 3 discusses diagramming conventions, and how we can use UML for reliable systems development. Section 4 briefly considers metamodelling and how it can be used to help establish consistency of views. Section 5 implicitly describes a development process by explaining the deliverables that are to be produced. In doing so, we describe when and how consistency of the different deliverables can be checked. We do not describe a process explicitly (i.e., by listing tasks and the order in which they should be carried out) since the method is not intended to be prescriptive in this way; instead, we focus on what deliverables should be produced, and the relationships between these deliverables. In Section 6 we discuss a proposal for prototype tools for the method, and the functionality they need to provide. Finally, in Section 7, we briefly contrast the method with Extreme Modelling, and explain our current work towards fulfilling the proposal.

2 Two Ways to Use Models

There are at least two ways in which modelling – via a graphical language like UML – can be viewed within the context of software development. These two perspectives are important, in that they can influence the construction of development and management processes that are compatible with modelling languages, and they can influence the different rôles that modelling languages can play within development. We now discuss these two views since they will be important in producing processes for our rigorous method.

The first view on modelling we call the *refinement* view. With this view, software development commences with modelling. Different models are constructed, information is added to the models, and over time the models are refined with more and more detail. At some point, the models are detailed enough so that code – in some suitable programming language – can be generated automatically from the models. At this point, developers may start working with code, and can make use of automated tools to help keep the models and code consistent, via round-trip engineering.

The second view takes the perspective that code is the most important deliverable from software development. Developers can, in practice, start development anywhere: working with code (in a suitable language) or working with models, though in practice they may want to start development by coding, since it is the key deliverable, and it is manipulable by tools. At any time, when working with one type of deliverable, the developers should be able to produce the alternative type of deliverable, or check that a new type of deliverable is consistent with what they already have. If developers started working with code, they should be able to use a tool to produce their models from the code, or draw a model and check its consistency with their code. If they started working with models, they should be able to produce code skeletons automatically. At any time, developers can switch perspectives and continue working with the deliverable that is currently useful.

It is clear that both perspectives on modelling are supportable by UML. However, in order to follow the second perspective safely, and to be able to either automatically switch between code and models or to check consistency of code and models, algorithms need to be provided and restrictions need to be placed on the modelling constructs from UML that are used. We now discuss these issues in more detail.

3 Diagramming Conventions and Models

The method that we are proposing makes use of a subset of UML, with specific restrictions on how this subset of diagrams is to be used. The restrictions are used to make it possible to check that source code or test drivers are consistent with UML diagrams (at user-specified points in time – see Section 4), and to allow diagrams to be automatically reverse engineered from source code and test drivers. We desire to do this so as to allow developers the flexibility to work with either code or models when they so choose, and for the purposes of documentation and maintenance.

Fig. 1 summarized the three main deliverables – source code, models, and test drivers – and their relationships. Fig. 2 refines Fig. 1 and specifies the three types of diagrams that the proposed method uses, along with the diagram relationships. We have used notes to indicate how the relationships are to be established.

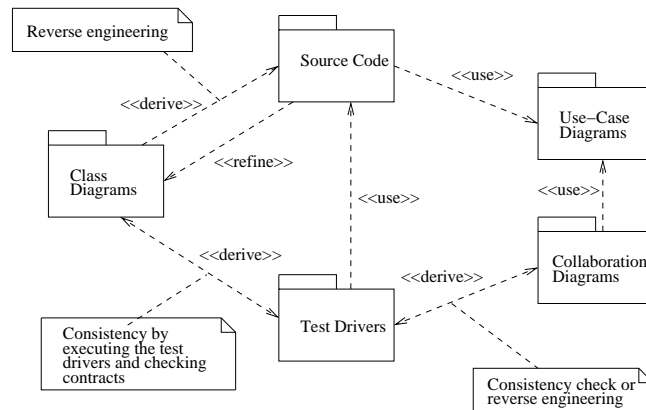


Fig. 2. Relationships between specific deliverables in the method

More specifically, the three types of diagrams to be used in the method are as follows.

- **Class diagrams:** using classes and interfaces, navigable associations, generalisation, and composition. Packages may also be used. Operations appearing in a class can be documented with contracts, written in OCL (see Section 3.1 for an example) or another suitable assertion language. Class diagrams can be reverse engineered automatically from source code, and can also be used for forward engineering, i.e., to automatically generate source code skeletons (including contracts) from diagrams.
- **Collaboration diagrams** are abstractions of *test drivers*. A test driver for an OO program is a set of classes, each of which provides methods to test various interesting scenarios that are relevant to the program. These methods create objects, and call sequences of methods on these objects, to obtain a desired result. An example is given in Section 3.2. Collaboration diagrams can be used in one of three ways in the method: (i) they can be drawn and thereafter used to automatically generate skeletons of test drivers; (ii) they can be checked for consistency, automatically, against a test driver which has been created or modified by a programmer; or (iii) they can be reverse engineered from a test driver. Test drivers use source code, and their consistency with class diagrams can be checked transitively by executing the test drivers: class diagrams are consistent with source code (via reverse or forward engineering). Thus, if the tests implemented in the drivers are satisfied, then the drivers are consistent with the source code, and therefore are consistent with the class diagrams by construction.
- **Use-case diagrams**, which are in all ways standard. In the proposed method, use-case diagrams can be used in the typical way to derive collaboration diagrams or class diagrams. We do not change use-case diagrams in any way: we use them to drive the production of other models, and they will not be guaranteed to be consistent with code.

The method does not use statecharts to capture behaviour: behaviour is captured via contracts in models and in code. If statecharts are desired, they should be automatically generated by a tool from contracts (e.g., akin to the SOMA technology [Gr98]). Note that collaboration diagrams capture the behaviour of test drivers, and not the behaviour of the

system. All diagrams can be annotated with notes, which provide no semantic content. The critical idea with the diagrams used in the method is that, with the exception of use-case diagrams, the diagrams are either consistent with code or test drivers, or can automatically be checked for consistency.

Informally, the deliverables are used for the tasks presented in Table 1. This is not meant to be prescriptive, rather, it is a general guideline as to how the deliverables can be used (so, e.g., conceptual class diagrams are not prohibited). The method will not prescribe an order in which the deliverables are to be produced - developers can and must choose the most appropriate order for their project.

Task	Deliverable
Requirements	use-cases
Design	collaborations and classes
Implementation	source code
Testing	test drivers

Table 1. Deliverables and tasks

3.1 Class diagrams and source code

The proposed method uses class diagrams in a particular way. Class diagrams are either an abstraction of code (i.e., they are reverse engineered from OO code) or they will be used to automatically generate code skeletons (i.e., code is forward engineered from class diagrams). The requirement to be able to keep code and models consistent requires us to restrict some of the flexibility of class diagrams. Specific restrictions on class diagrams will depend on the programming language to be used for implementation; thus, our method effectively specializes a UML profile, consisting of a subset of commonly used diagrams, for a specific programming language. For the purposes of concreteness in discussion, we assume that we are using a typical OO programming language like C++, Java, or Eiffel, which provides facilities for writing contracts.

We first require that associations in class diagrams are navigable (one or both participants in an association should be given responsibility for knowing of the association). If the target programming language that is being used supports subobject types (e.g., C++ or Eiffel), then composition relationships may be used, otherwise they are avoided. Aggregation is not used. Contracts are used to capture the behaviour of operations. The contracts also serve as formal documentation for users of the components. Fig. 3 shows an example of such a class diagram. Contracts are expressed using OCL, as constraints. Note that the contracts are kept within the diagram for the appropriate class; in this manner, it is difficult for contracts to get out of synch with their containing class (e.g., if attributes, types, or operations are changed), since the contracts are never separated from the operation or modelling element to which they apply.

Class diagrams can either be reverse engineered from code (in which case, contracts appearing in the code will also appear in the class diagrams), or they can be drawn and

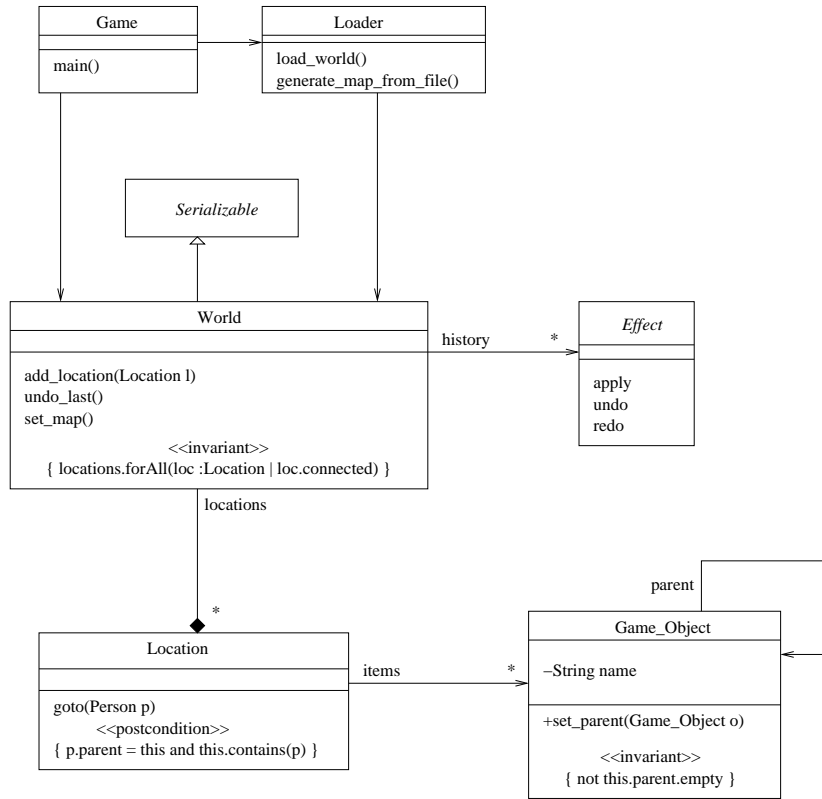


Fig. 3. Class diagram example

thereafter used to generate code skeletons (in which case, contracts appearing in the class diagram will appear in the code - although some contracts may have to be expressed as comments since they need not be executable and thus need to be checked via verification technology, e.g., theorem provers or static checkers). By using contracts throughout, static and run-time analysis can be applied to check contracts, and verify and validate the system - something that is essential for reliable software development.

3.2 Collaboration diagrams and test drivers

A test driver for an OO program is a class or collection of classes that tests the behaviour of the program. The driver accomplishes its tasks by declaring objects and by calling methods on objects; executing the driver's *main* routine results in the testing of a class's functionality. During the testing process, conditions may be checked to ensure the validity of the results being provided by the calls. Many checks will be instrumented by using contracts, i.e., pre- and postconditions of methods, and class invariants; others can be implemented by using program code (e.g., if-then-else statements) or inline assertions. If any contract fails during testing, the test driver will flag it. Similarly, if any of the test driver's checks fail, it will also be flagged. Fig. 4 contains an example of a regression test driver. Suppose that a class *World* (as in Fig. 3) has been constructed, and we wish to test it. A regression test driver is shown in Fig. 4.

A collaboration diagram is an abstraction of a test driver. It can be used for this purpose in several ways. The first, and perhaps least useful (and most expensive) way is to use them to depict the sequence of operation calls appearing in a test driver. In this manner, a collaboration diagram is reverse engineered from a set of test driver classes, and the messages appearing between objects represent method calls appearing in the code.

Fig. 5 depicts a collaboration diagram used in this sense, produced from the test driver for *World* shown in Fig. 4. Each message corresponds to a method call in the test driver. In order for a message to appear in the collaboration diagram, it must be well-formed. For example, in the message 4, corresponding to `connect(cave1)`, (a) the class of `Location` must provide an operation `connect`; (b) `WorldTestDriver` must have a call to `connect`; and (c) the test driver must have permission to make this call. A compiler will check all of these conditions and thus a type-correct test driver can be used to produce a collaboration diagram.

In this manner of use, automated support is necessary to reverse engineer the collaboration diagram from the test drivers. We point out that reverse engineering will typically produce very large collaboration diagrams (and very large collections of such diagrams), and it will be left to the user to organize them in a useful way. One useful way of organizing such diagrams is to have the user select an *initialization object*, from which the sequence of calls will be extracted. Clustering techniques should also be used, to group objects whose classes appear in the same package.

The second use of collaboration diagrams is for automatically generating test driver skeletons; this is *forward engineering*. A modeller produces a collaboration diagram, and a tool automatically transforms the diagram into the skeleton of a test driver. An algorithm must be provided for the transformation; clearly, the generated test driver skeleton will have to be extended by the developers. The outline of the algorithm used by this transformation is as follows.

```

class WorldTestDriver {
    public World w;
    public Location start_room, cave1, cave2, jail_cell;
    public Game_Object sword;

    public void main() {
        File f = new File("regression_test_World");
        FileWriter out = new FileWriter(f);
        w = new World("Adventure");
        start_room = new Location();
        cave1 = new Location();
        start_room.connect(cave1,Location.east);
        cave2 = new Location();
        cave1.connect(cave2,Location.south);
        start_room.connect(cave2,Location.south_east);
        jail_cell = new Location();
        cave2.connect(jail_cell,Location.west);
        w.add_location(start_room);
        w.add_location(cave1);
        w.add_location(cave2);
        w.add_location(jail_cell);
        out.write("the world is connected and add_location is correct\n");
        sword = new Game_Object();
        cave1.add_item(sword);
        if(cave1.has_item(sword))
            out.writer("has_item: the sword is owned by cave1 already.\n")
        else
            out.write("has_item: invalid result from has_item.\n");
        out.close();
    }
}

```

Fig. 4. Test driver for *World* class

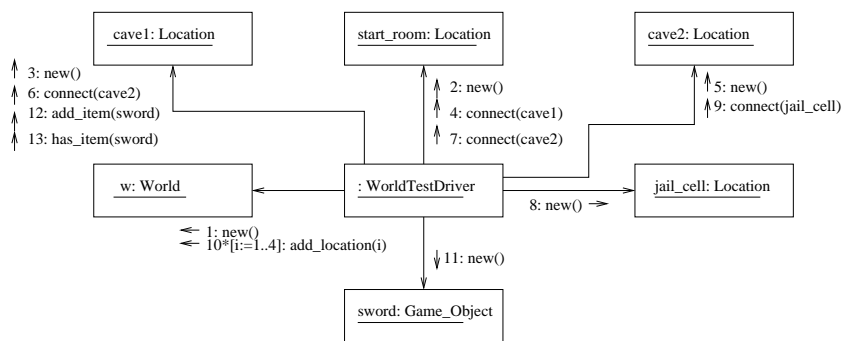


Fig. 5. Collaboration diagram example

- Each collaboration diagram is mapped to a test driver class with an initialization routine (e.g., `main()` in Java).
- Each object in the collaboration diagram is mapped to an entity (variable declaration) and allocation statement in the test driver.
- For each message f from an object of type B to an object of type C in the collaboration diagram, in order of sequence number:
 - Ensure that operations of class B can access operations of type C . If it cannot, emit a message.
 - Transform the message to a call $c.f$ in the initialization routine of the test driver. Each call will be represented with the appropriate syntax, whether it is an attribute or function or procedure.

The final way in which collaboration diagrams can be used effectively extends the previous two uses, while at the same time providing a perhaps more lightweight way to use collaboration diagrams and test drivers together. Suppose that an initial collaboration diagram has been drawn, and a test driver skeleton has been generated (as above). The developers have then modified the test driver directly, as needed. After a while, it is decided to update documentation, i.e., the collaboration diagram. Clearly, the collaboration diagram can be reverse engineered (as above), but it may suffice for the developers to know where the original collaboration diagram is out-of-synch with the test driver code; thereafter, the collaboration diagram can be updated manually if desired. Thus, a tool must be provided to check the consistency of a collaboration diagram with the test driver. An outline of the algorithm to be implemented by the tool is as follows.

1. The test driver must be type-correct (as checked by a compiler). This includes checking the information hiding model, i.e., that permitted clients make calls to methods.
2. There must be a message from an object of class A to an object of class B in the collaboration diagram for each: attribute a of type B ; function q of result type B ; and, command c with an argument of type B .
3. The ordering of calls expressed in the collaboration diagram must be *included* in a trajectory of calls possible in the test driver. This can be done by first extracting call numbers for operations from the collaboration diagram. Each number is associated with an operation and object, and is stored in a table. We then start execution of the test driver, and for each call made to a specific target, the number is found in the table. This is compared against the sequence numbers of messages appearing in the collaboration diagram.

Any and all discrepancies between the test driver and the collaboration diagram must be flagged with an appropriate, clear warning.

3.3 Use-case diagrams

The use-case diagrams applied in the method are in every way standard, and they can be applied in their usual ways, e.g., for late requirements capture or for user interface design. Our particular view on use-cases is that they are helpful in producing candidate classes, and also for suggesting testing scenarios. However, we do not treat them as *formal* modelling constructs, in the sense that they will be used directly for the production of code. Nor do

we expect to have automated support for keeping code consistent with use-case diagrams. In this sense, we view use-case diagrams as *rough sketches* [Ja95] - they are useful in the early stages of requirements engineering and for producing candidate classes. Other diagrams may be useful during early requirements engineering as well, and the proposed method does not disallow their use.

4 Metamodelling and Consistency Checking

A metamodel captures the well-formedness constraints on diagrams. It is a set of rules that diagrams must obey in order to be valid. A metamodel is vitally important for builders of tools to support modelling languages. A metamodel is thus a specification for tool builders, and it is part of their design process determining how to best implement the rule set in an efficient manner, e.g., through a constrained user interface, or by appropriate algorithms.

The metamodel for the subset of UML that we use is the standard one. However, there is one issue that we need to discuss related to metamodels, and that is *when* to check metamodel constraints.

Typically, a CASE tool that implements a metamodel prevents diagrams that do not satisfy the metamodel from being constructed. This, effectively, enforces a specific process on the use of the diagrams: some diagrams must, by their nature, be constructed before others in order for the metamodel constraints to be established. This may be inflexible and does not necessarily capture how developers may choose to work. We propose that some, but not all, metamodel constraints be automatically checked *on developer demand*. That is, a developer may construct a set of diagrams of interest, and in the process, some but not all metamodel constraints may be checked (for example, the constraint that a classifier in UML cannot self-generalise would be checked automatically). When the developer is satisfied, they can press a button on the CASE tool and the remaining metamodel constraints, e.g., those for establishing consistency of views, are checked.

For example, suppose that a developer has created a set of class diagrams, and now chooses to draw collaboration diagrams in order to start developing test drivers. The collaboration diagram may mention objects for which classes may not yet exist in the class diagram (e.g., `WorldTestDriver` in Fig. 4 may not appear in a class diagram, or classes that the developer simply has not modelled yet). Thus, the collaboration diagram and class diagram may be inconsistent. But this flexibility is necessary in order to let developers use the diagrams as necessitated by their processes.

Effectively, we are proposing that selective metamodel constraint checking be done in a manner similar to static checking tools, e.g., type checkers, ESC [Es00], etc. This allows developers more flexibility, and lets them construct models in an order and in a way that best suits their process. The main constraints that we propose to have checked in this manner are those associated with inconsistency across different views of a system. It is well-known that inconsistency is useful in modelling and reasoning about requirements [EC01], and so we allow inconsistent views (specifically, the collaboration diagram view and the class diagram view).

Inconsistency, while useful in early requirements engineering (particularly when developers and customers are not clear on the requirements for the project), is undesirable during detailed and architectural design, and particularly during coding. This is especially

true of reliable systems, which is our particular focus with this method. The use of contracts in class diagrams allows us to automate consistency checking of views.

5 Process

We do not explicitly prescribe a development process for our method in terms of workflows and activities. Rather, we describe the process implicitly in terms of deliverables. The method is focussed on the production of code: code is the main work product of development, with models and tools being used to assist in its generation. Fig. 2 suggested hints of a process for the method. The following key characteristics are possessed by the process.

1. Developers can start work with either models or code, as they choose. Any model or diagram can be constructed in any order (though it may be more convenient to construct an initial, perhaps rough sketch of a model, before any coding is done).
2. Both models and code are documented by contracts that can automatically be analyzed and checked via tools.
3. At any point in time, developers can switch between using models or code.
4. Automated support is to be provided to support the code/model switching in point 3.
5. Test drivers must be available to help verify code. Test drivers may be written by programmers or by constructing collaboration diagrams and thereafter by code generation and further programmer augmentation.

The proposed process has commonalities with Extreme Programming [Be99], wherein code is the principle deliverable, and testing is emphasized. In our proposed method, modelling is integrated into the process by using models as an alternative view of code products, or as an abstraction of code. Modelling provides a form of documentation for systems, but models are more than just documentation: they are machine checkable, can be analyzed and reasoned about, and can be used for generating code.

The order in which the developers choose to produce development deliverables is up to them. The following must be produced.

- Source code in an appropriate programming language that supports contracts.
- Test drivers that use the source code, produced either by developers or by code generation via collaboration diagrams.
- Class diagrams for the source code, either produced by reverse engineering techniques, or by developers who then use the diagrams for automatic generation of code.
- Collaboration diagrams for the test drivers, either produced by reverse engineering, or by developers who then use the diagrams for automatic generation of code.
- Use-case diagrams for generating candidate classes or for suggesting collaboration diagrams

Developers can start by producing any deliverable: code, class diagrams, use-case diagrams, et cetera. Tools will be used to produce related deliverables (as shown in Fig. 2), and development will be completed when all deliverables have been produced, and all tests have passed.

6 Tools

We have, in the preceding sections, discussed the need for a tool to support the method. This will provide the following functionality.

1. *Editors.* The tool will provide editors for drawing class diagrams, collaboration diagrams, and use-case diagrams that are UML compliant and that obey the additional restrictions discussed earlier. A service will be provided with the user interface to check consistency of diagrams on request.
2. *Code Generators.* The tool will allow automatic generation of code and test drivers from diagrams, as discussed in previous sections. Code will be generated in suitable object-oriented languages, e.g., C++, Java, and Eiffel. As a minimum, the target language should provide the following features.
 - support for reference types, collections (e.g., sets and sequences), and value-types (e.g., expanded types in Eiffel)
 - support for contracts, either built-in (e.g., Eiffel, Java 1.4), or via a preprocessor (e.g., iContract [Kr98] for Java) or library (e.g., a reflection package for Java or C++). These contracts must be run-time checkable when they appear in source code.
3. *Reverse engineering.* The tool will provide reverse engineering facilities for suitable programming languages, e.g., Java, C++, and Eiffel. This tool will automatically extract a UML class diagram (which obeys the constraints discussed earlier) from code. Collaboration diagrams will be produced from user-identified test driver classes, which are to be considered as *initialization classes*.
4. *Metamodel and consistency checks.* The tool will implement metamodel constraints. The metamodel constraints related to consistency of views presented by collaboration and class diagrams will be checkable at the discretion of the user, via some user-interface option. All other metamodel constraints will either be implemented using suitable data structures or user interface features.

We have implemented a prototype tool, currently under extension to complete these requirements, in the context of an alternate modelling language, the Better Object Notation (BON) [WN95], which is equivalent to the subset of UML that we have identified in this paper. Specifically, BON class diagrams are the same as UML class diagrams with directed relationships, BON dynamic diagrams are equivalent to collaboration diagrams, and BON's contract language is used in a similar fashion to the Object Constraint Language (though we note that BON's contract language is based on first-order predicate logic, unlike OCL). When the use of BON is combined with a language such as Eiffel (which provides built-in support for contracts and tools for run-time contract checking), then can be used in a fashion identical to the subset of UML combined with a contract-supporting programming language that we have identified in this paper. We chose to implement the prototype of the tool with BON due to its superior built-in support for contracts, the availability of tools to do run-time checking of BON contracts, and due to its simplicity. We are also in the process of generating a version of the tool that uses UML instead of BON. The tool will be straightforward to adapt to UML (and other modelling and programming

languages), because of its design: the presentation style of models is a separate, independent component in the tool, and thus its modification without affecting other components is straightforward.

Currently, the tool implements 1 (use-cases are in alpha release), 2 (for iContract-annotated Java [Kr98] and Eiffel, as well as the JML modelling language) for class diagrams, and much of 4. Parts of 3 are in the process of being implemented. Shortly, it is expected that all four requirements listed above will be complete.

7 Discussion and Conclusions

Our proposal is for a tool-supported rigorous method for developing reliable software systems. Reliability is supported through use of contracts in both models and in code, as well as by providing tools to support consistent views of a system, via code generation and reverse engineering capabilities.

The method is centered on code: code is the critical deliverable to be produced by development, and as such it plays a key rôle in the production process. Development can either start with code or models, and through use of tools, models and code can be produced and maintained, and test drivers can be produced. In this manner, the process is similar to that offered by extreme programming, but it integrates the use of modelling and modelling languages to the extent that developers find useful for their specific project. Models are used as an abstraction of code, and can either be produced before or after code has been constructed.

The method has much in common with Extreme Modelling (XM) [Am01], which attempts to combine use of models with Extreme Programming practice. XM has the same values as Extreme Programming, except that it recommends first producing in short order as many or as few models as is helpful. It argues against using CASE tools to merely produce documentation, with the understanding that the goal of development is to produce software, not documentation. There are two main differences between XM and the method proposed in this paper.

1. In the proposed method, documenting is part of coding – the two tasks are never separated. Documentation is machine readable and checkable, and is integrated with the code. Programmers produce documentation *by* coding, not as an auxiliary task in the development process.
2. The method does not recommend using models before coding. It permits a more flexible use of models: they should be used whenever developers think that they will help, either before, or during, or after coding, as necessitated by the development and management process. Tools are used to support this flexible process, and are not used merely for documentation purposes.

The value of being able to work immediately with code cannot be overstated. By working with code immediately, we can use existing tools, such as compilers and debuggers, to check our work, and we can also start to test at a very early stage. Static- and runtime checkers can be used to validate even skeleton implementations. But models are very important as well, especially for capturing architectural designs and conveying complex systems with many interacting parts to developers. As well, when dealing with vague or

potentially contradictory requirements, it is oftentimes premature to begin work with code. Instead, a modelling language can be usefully applied, which abstracts away much of the irrelevant details that do not need to be considered at such early stages of development. In other words, both modelling and programming languages are vitally useful during development, and a software development method should not dictate which type of language should be applied at the start of development - it should allow the developers to select the right tool at the right time for their project, and should allow them to start development anywhere, with any development product.

Our previous work with BON and Eiffel (e.g., [PK01,PO99]) was instrumental in the development and proposal of this method. BON combined with Eiffel is effectively equivalent to the profile of UML that we are using herein. Our initial work on tool support has, for reasons of simplicity and scope, focussed on BON and Eiffel, but the tools that we have created have been designed in such a way so as to make it straightforward to adapt them to UML and other programming and modelling languages². Our future work thus focusses on completing the tool support described in Section 6, and then we shall turn to experimental work on demonstrating the efficacy of the approach on complex software development problems.

Bibliography

- [Am01] Ambler, S.: A closer look at Extreme Modeling, *Software Development*, April 2001.
- [Be99] Beck, K.: *Extreme Programming Explained*, Longman Publishers, 1999.
- [DW98] D'Souza, D.; Wills, A.: *Objects, Components, and Frameworks with UML: the Catalysis Approach*, Addison-Wesley, 1998.
- [EC01] Easterbrook, S.; Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In (Harrold and Schaefer): *Proc. International Conference on Software Engineering 2001*, IEEE Press, May 2001.
- [Es00] The Extended Static Checker. Compaq System Research Centre. www.research.digital.com/SRC/esc/Esc.html, 2000.
- [Fo01] Fowler, M.: Is design dead? *Software Development*, April 2001.
- [Gr98] Graham, I.: *Requirements Engineering and Rapid Development*, Addison-Wesley, 1998.
- [Ja95] Jackson, M.: *Software Requirements and Specifications*, Addison-Wesley, 1995.
- [Kr98] Kramer, R.: iContract - the Java Design by Contract Tool. In *Proc. TOOLS-USA 1998*, IEEE Press, August 1998.
- [Me97] Meyer, B.: *Object-Oriented Software Construction*, Second Edition, Prentice-Hall, 1997.
- [PK01] Paige, R.; Kaminskaya, L.: A Tool-Supported Integration of BON and JML, submitted July 2001.
- [PO99] Paige, R.; Ostroff, J.: Developing BON as an Industrial-Strength Formal Method. In (Wing and Woodcock): *Proc. World Congress on Formal Methods 1999*, LNCS 1708, Springer-Verlag, September 1999.
- [WN95] Walden, K.; Nerson, J.-M.: *Seamless Object Oriented Software Architecture*, Prentice-Hall, 1995.

² Indeed, already Java is supported as a target language from the BON-CASE tool [PK01].