Producing Reliable Software via the Single Model Principle

Richard Paige and Jonathan Ostroff

Technical Report CS-2001-02

May 10, 2001

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Producing Reliable Software via the Single Model Principle

Richard Paige and Jonathan Ostroff
Department of Computer Science, York University
Toronto, Ontario M3J 1P3, Canada.
{paige,jonathan}@cs.yorku.ca

Abstract

We contrast software modelling languages and developments that are founded on use of a single model with those founded on use of multiple models. The advantages and disadvantages of each approach are discussed. We propose that to best support seamless and reversible development of reliable software, languages and developments that follow the single model principle are superior for late requirements engineering through to implementation. We define the principle precisely, provide examples of languages that support it, contrast it with multiple model development, and discuss when the principle is insufficient: when dealing with inconsistent descriptions, and for capturing and manipulating early requirements.

1. Introduction

Software development often makes use of modelling languages, e.g., UML [13] and Eiffel [9], for describing systems and their requirements. Modelling languages are usually supported by tools and development processes that commonly target executable programming languages. Modelling languages are typically used to describe early and late requirements, capture architectural and detailed designs, and provide documentation for maintainers of systems.

There are two fundamental kinds of modelling languages: those that are founded on use of a *single model* to describe concerns of interest; and those that make use of multiple, independently constructed models. The purpose of this paper is to critically compare these two kinds of languages, particularly for building reliable software. We focus, specifically, on the use of a single model in late requirements engineering, design, and implementation; early requirements engineering is discussed in Section 4.

1.1. Using multiple models

A representative example of a modelling language that uses multiple models is UML. With UML, a system is described using disparate, independently constructed models [13]. Each model[1] presents the system of interest from a different perspective. For example, one model may describe architectural details of the system (e.g., how the components that make up the system are connected), whereas a separate model may describe behavioural details (e.g., how components respond to messages dispatched from clients). Information presented in one model may also be captured, albeit differently, in another model. That is, the models may overlap in terms of the information that they present.

The UML 1.3 Standard [13] says the following.

> "Every complex system is best approached through a small set of **nearly independent** views of a model; no single view is sufficient. In terms of the views of a model, the UML defines the following graphical diagrams:
>
> - use-case diagram
> - class diagram
> - behaviour diagrams [...]
> - implementation diagrams [...]
>
> These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a **self-consistent** system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and support tools will provide for a number of derivative views."

By using multiple models, developers can work independently on separate parts of a system, and can apply the most appropriate diagrams or notations for describing each part. When it comes time to construct executable code from the

[1] In the UML Standard 1.3 document [13], each model is called a *view*. Section 3.4, where the metamodel of UML is partially presented, clarifies that they are in fact models.

models, the models must be integrated into a single model that satisfies all the constraints and descriptions contained in the individuals. For example, a class may be described by several UML models: a class diagram, pre- and post-conditions for methods (written in, e.g., OCL), and a state transition diagram. For consistency, transitions in the state machine must correspond to precondition clauses. However, there is nothing in the UML nor in its supporting processes to prevent that, for example, a precondition contains the condition P while the corresponding state transition is erroneously guarded by the condition $\neg P$. Thus, the multiple model approach allows inconsistent descriptions of systems. In a language like UML, neither infrastructure nor tools are provided to detect such inconsistencies. Using a single model, as opposed to multiple models, does not guarantee consistency, but it does make inconsistency easier to detect and deal with.

1.2. Using a single model

Modelling languages and developments that make use of a single model to describe a system obey the *single model principle*, defined precisely in the sequel. Informally, such a modelling language provides the means for expressing a unique description of the system under consideration. The principle will require three characteristics of modelling languages: that there be *conceptual integrity* of abstractions that appear in descriptions of models; that there is support for checking *consistency* of descriptions; and that the language is *wide-spectrum applicable.* An example of a modelling language that obeys the single model principle is Eiffel[2]. With Eiffel, a single model can be constructed and used throughout software development, with automatic or semi-automatic generation of multiple views [12] of the model. We provide an overview of Eiffel in Section 2, and explain why it obeys the principle in successive sections.

1.3. Seamlessness and reversibility

The ability to support *seamless* and *reversible* software development has been identified as important characteristics for a modelling language to possess if it is to be used successfully to develop reliable software [9, 15, 18]. Seamless development is founded on the use of a set of common modelling abstractions – e.g., classes, binary components, etc. – throughout the development process. In this manner, impedance mismatches that potentially introduce errors during development can be avoided, and errors that arise in abstractions during one phase of development can easily be traced back to errors made during previous phases. As well, abstractions used in early phases can be directly mapped to the same kinds of abstractions used in later phases [9]. Reversible development means that models can be generated automatically, via software tools, from programs. In this manner, it is easy to keep programs and models consistent, thus aiding in documentation and maintenance.

We propose, in this paper, that to best enforce and support seamless and reversible development, a modelling language should obey the single model principle for late requirements engineering through implementation. We will demonstrate that modelling languages that support the principle will support seamless and reversible development.

Using multiple models is not a good way to produce reliable software.

- Seamless development becomes difficult, if not impossible, because of overlapping descriptions. An abstraction, e.g., a class, can be described in multiple models, in different ways, thus prohibiting the direct mapping of a class in a model into a class in a program.
- It is easier to introduce inconsistencies into a description of a system by splitting the description up into several models[3].
- Separate models that are themselves consistent may lead to inconsistencies when they are integrated. Further, integration may introduce undesirable properties or consequences, when models and descriptions are thereafter composed. Checking the consistency of composed specifications or models is a challenging problem [19].

1.4. Insufficiency of a single model

There are two key points to make before we present further technical details. We are positing a single model-based approach as superior to multiple model approaches for late requirements engineering through implementation. We are not claiming that multiple *views* are unnecessary or valueless. Multiple views of a model are very useful, if not essential, in software development. Our perspective is that multiple views should be consistent by construction wherever possible, thus implying that they should be produced – ideally automatically – from a single model. We discuss this more in following sections.

The second point is that a single model-based approach will not be sufficient for all development tasks. We discuss such tasks in Section 4, particularly focussing on dynamic

[2] It is a mistake to consider Eiffel as just a programming language. We discuss this more in Section 2, but we point out for now that Eiffel does support specification independent of implementation.

[3] Nuseibeh et al [12] claim that inconsistency in requirements descriptions must be tolerated, and that consistency checking of descriptions need not be done routinely, as a matter of course. [6, 12] provide frameworks for detecting and managing inconsistency in requirements descriptions. This is discussed more in Section 4.2.

modelling and early requirements gathering and manipulation.

We will illustrate the single model principle using the Eiffel language [9], and the multiple model approach using UML and Java. We thus commence with a brief overview of Eiffel and its concepts, before turning to technical details.

This paper is motivated by Meyer's *self-documenting principle* [9] (p55), where it is stated

> "...software becomes a single product that supports multiple views. One view, suitable for compilation and execution, is the full source code. Another is the abstract interface documentation of each module, enabling software developers to write client modules without having to learn the module's own internals... Other views are possible."

2. Overview of Eiffel

Our perspective of Eiffel is that it is a modelling language, and thus can be used for describing systems, not just programs. This is compatible with that of Meyer [9], who sees Eiffel as a method for software development. Eiffel obtains this generality in part because of its support for specification, via assertions that can be embedded within classes (in the form of class invariants) and *contracts* which are associated with features of classes. A valid Eiffel model may consist only of specifications – that is, it may possess no program code whatsoever; or a combination of specification and code; or code only. It is thus a true wide-spectrum language in the style of [5, 11].

Eiffel has both ASCII-based and graphical dialects; we use only the ASCII dialect here (see [18] for the graphical dialect, BON). Eiffel provides mechanisms for specifying inheritance, association, and aggregation relationships between classes. It also possesses techniques for expressing dynamic relationships, via feature calls. The assertion language for Eiffel is first-order predicate logic. Not all assertions that can be written are executable (though with the Eiffel compiler's support for agents [10], this is less of an issue).

Here is an example of part of an Eiffel model of a class *CITIZEN*. The class has four attributes, a boolean-valued function *single*, and a state-changing procedure *divorce*. **require** clauses are preconditions, and **ensure** clauses are postconditions. Postconditions can refer to the value of an expression when the feature was called by prefixing the expression with the keyword **old**. Classes may also have *invariants*, which are predicates that must be maintained by all visible routines. Visibility of features is expressed by annotating **feature** clauses with lists of client classes permitted to access the features.

class *CITIZEN* **inherit** *PERSON*
feature *NONE*
children, *parents* : *SET*[*CITIZEN*]
spouse : *CITIZEN*
feature *ANY*
single : *BOOLEAN*
ensure *Result* = (*spouse* = *Void*) **end**
divorce
require not *single*
ensure *single* **and** (**old** *spouse*).*single* **end**
invariant *c* *children* • *p* *c.parents* • *p* = *Current*
end

Eiffel supports inheritance relationships (e.g., between *PERSON* and *CITIZEN* above), association relationships (e.g., between *CITIZEN* and itself, via attribute *spouse*) and aggregation relationships. Visual descriptions of relationships can be found in [18]. Description of dynamic behaviour, e.g., message passing, is supported by describing function and procedure calls, and can be visually represented as in [18], using object communication diagrams.

3. The Single Model Principle

The single model principle offers an alternative to multiple model development that is preferable for building software seamlessly and reversibly for late requirements engineering through implementation. The informal idea is simple: a single description of the system of interest is constructed. Whenever different views of the software are required, they are produced automatically or semi-automatically from the description. In this manner, views are guaranteed consistent because they are all constructed from the same source. We now describe the principle more precisely.

In developing a modern software system, two general kinds of abstractions are typically produced[4]. Modelling languages such as Eiffel and UML, and programming languages such as Java, are used to implement these different kinds of abstractions.

- *Modules:* units of encapsulation, which have a notion of interface that describes what aspects of the module are accessible and inaccessible to other modules.

[4]The abstractions that we describe should more properly be called *meta-abstractions*: modelling and programming languages may provide different subtypes of each meta-abstraction. For example, the Object-Oriented Turing language provides both an abstract data type construct and a class construct.

Modules encapsulate both state and behaviour. Typical kinds of modules include abstract data types, binary components, and classes.

- *Systems:* collections of interacting modules and other systems; thus, systems contain modules and possibly other systems along with the relationships that are defined between them.

Table 1 illustrates the different kinds of abstractions that are created during software development, and how they are described using Eiffel, UML, and Java. They are discussed more in Section 3.1 and 3.2.

Based on these abstractions, we can define the single model principle precisely.

Definition 1. *Single Model Principle.* A modelling language that obeys the single model principle satisfies the following three criteria:

1. **Conceptual integrity.** Conceptual integrity refers to the abstractions possessed by a modelling language and how they are described. A language with conceptual integrity first possesses physical integrity of descriptions: for each different kind of abstraction that can be produced by the language, all information about the abstraction is kept in exactly one physical place, e.g., a file. Second, a language with conceptual integrity provides exactly one way of describing concepts of interest. For example, Java possesses physical integrity, but not conceptual integrity: the concepts of class and interface are semantically redundant (as shown in [9], an interface is just a special kind of class). UML possesses neither physical integrity nor conceptual integrity (as discussed in Section 3.2).

2. **Consistency of views.** The language must provide infrastructure that makes the checking of the consistency of different views of a model as automatable as possible, e.g., the consistency framework of [12], or the framework for composing specifications of [19]. In terms of tools, we might provide automatic generation of views or support via a theorem prover. It is preferable to guarantee consistency of views by construction.

3. **Wide-spectrum applicability.** The language is applicable to modelling concepts and constructs throughout late requirements engineering (when customer goals are well-defined, and alternatives have been considered and selected) through to implementation.

Conceptual integrity is a necessary requirement for seamlessness: if information about an abstraction is kept in more than one description, that information cannot be directly mapped to abstractions in successive phases of the development process. Similarly, having multiple ways of describing abstractions can make it difficult to seamlessly map models into executable code. Consistency of views is vital for building reliable software; a reliable system cannot permit inconsistency. A modelling language for building reliable software should provide methodological and tool support for reasoning about consistency. Finally, wide-spectrum applicability is also a necessary requirement for seamlessness: without wide-spectrum capabilities, a seamless mapping from requirements and design models through to programs (and the reverse) will not be possible.

To better understand the principle, we now discuss how Eiffel and a development method founded on UML and Java supports, or fails to support, the single model principle.

3.1. Eiffel support for the principle

By construction, Eiffel obeys the single model principle. The criterion of conceptual integrity is satisfied: in Eiffel, modules are classes, classes are types, and all information about a class - including interface details and executable code - is kept in a single `.e` file. Systems in Eiffel are described via directories containing classes and subdirectories, as well as a single `Ace` file explaining how to execute the software. Thus Eiffel provides physical integrity of descriptions. Conceptual integrity arises since Eiffel only provides one abstraction, the class, for modelling and implementing software.

Eiffel also satisfies the criterion of consistency: systems are described using classes. All information related to each class is contained within its description, and all pertinent information can never be separated from the description of the class. Different views of the class can be generated automatically or semi-automatically (see Section 3.3). A module in Eiffel can be checked for consistency. The simplest form of consistency to establish can be done simply by compiling the module, which ensures static syntactic and semantic consistency. Other forms of consistency, e.g., that preconditions and postconditions are satisfiable, that all routines in a class obey the class invariant, etc., can be established via verification. The paper [16] provides a framework for doing this semi-automatically using PVS. The key point is that consistency checking is carried out by examining the module's text directly.

Eiffel is wide-spectrum applicable, as discussed earlier and in [9]. It can be used to describe programs, requirements, and designs [9, 18]. All modelling abstractions used in one stage can be directly mapped to abstractions used in later stages of development.

	Eiffel	UML	Java
Systems	Directory with `.e` files and subdirectories, with an Ace file that indicates the root class	System model with designated main class diagram	Package containing classes, interfaces and other packages. One class with `main()` method.
Modules	Class in a `.e` file	Class, interface, datatype, node, component, etc.	Class or interface, in a `.java` file

Table 1. Abstractions produced during software development

3.2. UML and Java support for the principle

We start by considering conceptual integrity. The Java language supports physical integrity directly, because each class, interface, and package resides in its own file (`.java` files) or directory. The language does not satisfy conceptual integrity: Java possesses syntactic constructs for classes, interfaces, and primitives, all of which describe the same semantic concept[5]; as Eiffel and C++ show, there is no need for separate syntactic constructs. UML does not satisfy physical integrity at the level of modules. At the module level, UML uses classes, interfaces, datatypes, nodes, and components as classifiers, and further constraints (e.g., in OCL, multiplicity, etc.) can be associated with classifiers; however these constraints need not be kept in the same description as the classifier they constrain [13, 15].

Turning to consistency of views, Java satisfies the criterion because all information about a module or a system is kept in one place (i.e., a `.java` file, a package, or a directory), and checking the consistency of such a description can be done using a Java compiler (or a suite of syntax/semantic checking tools). However, UML does not satisfy the criterion at the module level. As we discussed with physical integrity, UML classifiers may be constrained in several views; e.g., a class may be constrained in a class diagram, in an OCL constraint written elsewhere, in a state transition diagram, etc. Checking the consistency of such a collection of descriptions and views is difficult, and is currently a research problem.

Finally, we consider wide-spectrum applicability. Java, by itself, is not sufficient as a wide-spectrum language: it possesses no language features for *specifying* behaviour. Java coupled with a design-by-contract tool such as *iContract* [8] would satisfy the criterion. UML itself is not wide-spectrum (as it is independent of any programming language); however, UML combined with Java and *iContract* would satisfy the criterion.

[5]Interfaces are also included in Java to eliminate apparent complications that arise with multiple inheritance.

3.3. Deliverable dependencies

A simple way to illustrate the principle is by considering development deliverables and their dependencies. Fig. 1 represents a subset of the Eiffel deliverables and their relationships.

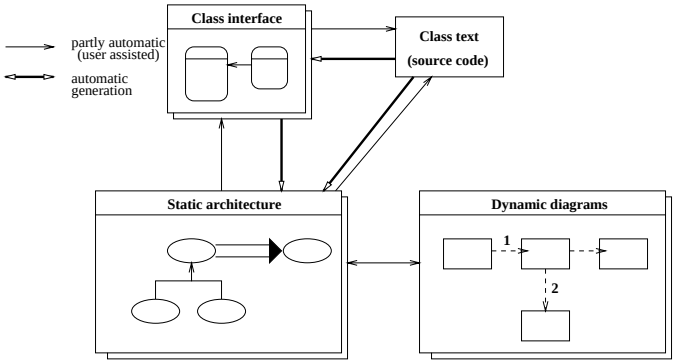

Figure 1. Eiffel deliverable dependencies (from [18])

With Eiffel, all deliverables (i.e., class interfaces, source code, static architectures, dynamic diagrams) are related by automatic generation via tools, or by user-guided semi-automatic generation. The key point to note is that all deliverables with Eiffel are dependent: they cannot be independently constructed, and thus inconsistent deliverables cannot be produced.

If we contrast this with the approach offered by UML/Java (depicted in Fig. 2), we find a different situation.

With UML/Java, deliverables (e.g., class diagrams, state machines, OCL constraints, collaboration diagrams, etc.) can be constructed independently and thus can introduce overlap and inconsistencies. For example, consider a UML model consisting of a class diagram (where methods have pre- and postconditions written using OCL) and a state transition diagram. For consistency, preconditions must correspond to guarded transitions in the state machine, but nothing in the modelling language enforces this, nor does the language provide necessary theory, methods, or tools to

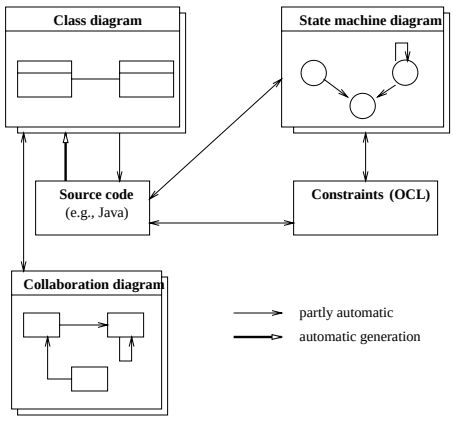

Figure 2. Deliverable dependencies using UML/Java

check or enforce consistency. It is thus left to the modeller to ensure or enforce consistency in their descriptions.

The single model approach, which is taken by Eiffel, is more fundamental than the multiple model approach, exemplified by UML/Java. Consider the problem of checking the consistency of independently constructed multiple models. To do this, the models must be combined in a common framework and reasoning must be carried out. But this is the process of constructing a single model — containing all descriptions regarding a software system of interest. Now, the construction problem and the consistency checking problem is more difficult than had development started with a single model in the first place. To construct a single model, we must combine information and constraints from several possibly very large, complex separate models. In contrast, with the single model approach, a large model is constructed piece by piece, by adding new descriptions (e.g., method signatures, code for methods, contracts) over the course of the entire development, and different views are produced from the model automatically or semi-automatically.

3.4. The single model principle and metamodelling

Both Eiffel and UML are modelling languages, possessing metamodels that specify the syntactic well-formedness constraints that all models must obey. By examining the metamodels for each language, we gain further insight as to why Eiffel supports the single model principle and why UML does not directly.

Fig. 3 depicts a fragment of the Eiffel metamodel; we write the metamodel in BON (Eiffel's graphical dialect). The diagram shows that in Eiffel a model consists of a set of abstractions. An abstraction may be a class, a cluster, an object, or an object cluster. These abstractions may have relationships with other abstractions. Each metaclass in Fig. 3 has constraints (written as clauses in class invariants) that we omit; see [17] for further details. The diamond on the association between the two clusters indicates the cardinality of the association: there are two associations directed from the abstractions to the relationships cluster.

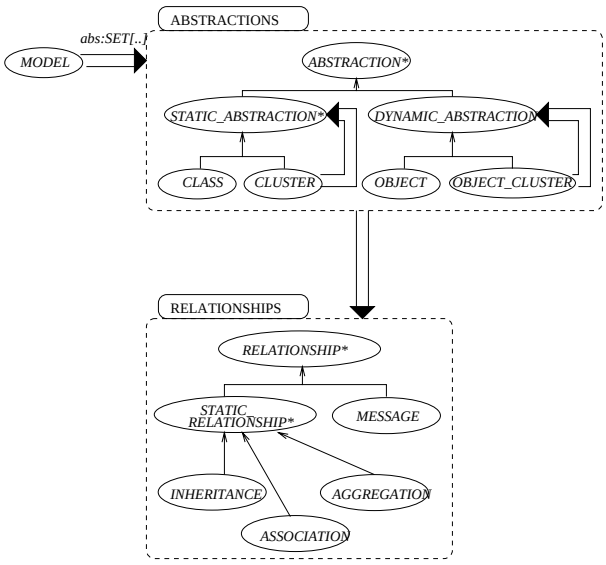

Figure 3. A fragment of the Eiffel metamodel

This diagram clarifies why Eiffel supports the single model principle: all abstractions that are associated with a model are contained and constrained within the model itself. Well-formedness constraints on messages, for example, ensure that each message corresponds to a feature provided by a class; objects must have classes in the model, too.

Consider now Fig. 4, presenting a fragment of the UML metamodel, extracted from the complete metamodel in [13], and depicted in BON. The fundamental concept in a UML model is a *MODEL_ELEMENT*, which is subclassed by concepts such as *CLASSIFIER*s and *INTERFACE*s.

Notice that a *MODEL* is also a subclass of *MODEL_ELEMENT*; that is, according to the metamodel, a valid model may be constructed from several different models (where each is an instance of the metaclass *MODEL*), each potentially containing different abstractions and relationships. These models may therefore be independently constructed – indeed, each view is of itself a model – and nothing in the metamodel or in the semantics of UML guarantees consistency of these views. This is why UML is a multiple model approach. Dependencies between separate views must be dealt with by the developer. There is no theory provided with the UML that can help to test or verify that a set of models is consistent or inconsistent.

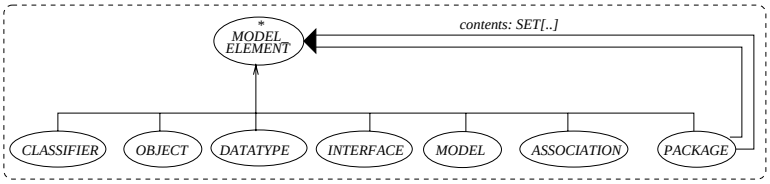

Figure 4. A fragment of the UML metamodel

4. Where the principle is insufficient

We have advocated the single model principle for building software seamlessly and reversibly. In effect, the single model principle defines a process (or, more properly, a meta-method) for building software, wherein a single suite of expressive modelling abstractions and technologies are applied throughout development, and multiple consistent views of a single model are generated systematically and semi-automatically when needed. An example of a process that is founded on the single model principle is discussed in Meyer [9], viz., the *cluster process* for development. In this process, a system is constructed from separate clusters (equivalent to packages in UML or Java, consisting of modules and other clusters). Separate clusters can be designed and implemented concurrently and separately, by separate groups of personnel, who make use of precisely specified interfaces of modules.

We now examine situations where the single model principle appears to be, or actually is, insufficient for rigorous development of reliable software. In particular, we want to focus on dynamic modelling, early requirements engineering, and dealing with consistency.

4.1. Dynamic modelling

Modelling languages such as UML support dynamic modelling via, e.g., sequence and state transition diagrams. These diagrams can show the objects created by a system and the messages passed between objects as the system executes. It appears that dynamic modelling may violate the single model principle, particularly when used with a language like Eiffel. However, this is not the case.

- Sequence diagrams can be treated as a generated view, constructed automatically from Eiffel implementations of methods of a class (that is, the messages depicted in a dynamic model simply show the sequence of function or procedure calls within an executing system). From this perspective, dynamic models cannot be constructed independently from static models (e.g., as is permitted with UML). This perspective on the use of sequence diagrams differs from that of Harel [4], wherein they are posited as a modelling technique for capturing requirements.
- Sequence diagrams can be viewed as *rough sketches* [7], which provide informal documentation that is ascribed no precise semantics, and which can be used to construct more rigorous descriptions. For example, a sequence diagram could be used as informal documentation for implementing methods of classes. With such a perspective, dynamic models cannot be used for the purposes of automatic generation.
- State transition diagrams are used in UML for describing the behaviour of objects; Harel posits them as a mechanism for modelling design [4]. Eiffel does not support state transition diagrams, but they can be generated automatically from class interfaces using, e.g., an approach similar to that of the SOMA toolset [2].

Thus, dynamic modelling can be used compatibly with the single model principle: following the principle enforces a specific process for using dynamic models.

4.2. Dealing with inconsistency

Nuseibeh et al [12] suggest that, when dealing with early requirements – i.e., when customers are uncertain about their needs, and are explaining their requirements in terms of "desires" and "goals" rather than functionality – inconsistency in requirements descriptions should be tolerated. As discussed earlier, frameworks have been proposed to allow inconsistent descriptions to be written, and inconsistencies to be detected [12, 19]. Such inconsistencies can be restricted to one, or several, views.

Inconsistent descriptions are incompatible with the single model principle. Consider descriptions written in Eiffel: inconsistent Eiffel descriptions are possible, but are limited to inconsistent classes (e.g., a class with an unimplementable method, or with a class invariant evaluating to *false*), and are viewed as undesirable. Indeed, an Eiffel development environment will flag such descriptions as invalid. Such inconsistent descriptions will also not be expressive enough to capture the goals and desires of customers that are discussed in [12]. One implication of this is that modelling languages based on the single model principle will be inappropriate for capturing inconsistent descriptions of early requirements, in part because of their seamless nature.

4.3. Early requirements engineering

Requirements engineering typically occurs in two distinct phases. *Early* requirements engineering is focussed on understanding, capturing, and analysing specific customer goals (which have a clear-cut criterion for satisfaction) or soft-goals (which need not have a precise specification of satisfaction). *Late* requirements engineering occurs when goals are well-defined and real-world entities can be modelled. Goals are critical in dealing with non-functional requirements. A modelling language like Eiffel, based on the single model principle, is insufficient for modelling goals, in part because of its seamlessness. Eiffel provides no built-in techniques for modelling goals: they would have to be treated informally (e.g., as comments, rough sketches, or informal documentation), and would not be directly expressible in executable code, thus defeating seamlessness. In order to treat goals formally, a richer modelling language, e.g., KAOS [1] could be used, however this would defeat seamlessness and would introduce impedance mismatches, particularly in mapping designs to programs, unless the resultant OO programming language supports goal-based constructs as well. The impedance mismatch can be minimized by providing rigorous (automatic or semi-automatic) translations from a language for goal-based modelling to a language such as Eiffel, which obeys the single model principle. A translation from KAOS to Z has been defined, and mappings from Z to BON (and thereafter to Eiffel) appear in [14]. This approach is also followed by Graham [3], in his task-based modelling techniques that support traceability. Thus, the single model principle seems incompatible with goal-based requirements description, but translation methods could be used to ameliorate the incompatibility in practice. We suggest that the single model principle be followed from late requirements engineering through to the production of code.

5. Discussion and Conclusions

Independently generated multiple models of a system cause more problems than they solve in developing software. It is claimed that they are useful because they allow developers to work independently on different parts of a software system, and thereafter their individual work can be integrated. We have already remarked on problems with this approach, particularly with consistency: checking that one independently created model does not contradict a second independently created model is a very complicated problem, even for small systems. The problem is avoided by obeying the single model principle.

We have been careful to phrase our arguments in terms of developing *reliable* software. Consistency of descriptions is an essential facet of developing reliable software, and thus we suggest that obeying the principle in this specific domain is necessary. However, the single model approach appears incompatible with early requirements engineering and for modelling inconsistency: systematic and automatable translations between appropriate languages may be necessary to help in these phases.

The multiple model approach offered by languages such as UML is not a good way to build reliable software. It is not a good mechanism for ensuring consistency, nor to help trace errors in programs back to errors in models. A single model approach, wherein different views of a system can be automatically or partly automatically generated from a single model of the system, should be preferred for developing high-quality software systems.

We have used Eiffel to illustrate the single model principle, but our arguments are not limited to this modelling language: they apply to any language which provides a unique way of describing abstractions of a system of interest, and which relies on automatic generation of views. It remains to be seen if Eiffel is an appropriate language for implementing the principle in realistic software development situations.

References

[1] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[2] I. Graham. *Requirements Engineering and Rapid Development*. Addison-Wesley, 1998.

[3] I. Graham. *Object-Oriented Methods, Second Edition.* Addison-Wesley, 2001.

[4] D. Harel. From play-in scenarios to code: an achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.

[5] E. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.

[6] A. Hunter and B. Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and actions. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, October 1998.

[7] M. Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[8] R. Kramer. iContract - the Java design by contract tool. In *Proc. TOOLS 1998*. IEEE Press, 1998.

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

[10] B. Meyer. Agents, iterators, and introspection in Eiffel. Technical report, ISE Inc., 2000.

[11] C. Morgan. *Programming from Specifications, Second Edition*. Prentice Hall, 1994.

[12] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[13] Object Modelling Group. UML Standard Guide 1.3, 1999.

[14] R. Paige and J. Ostroff. From Z to BON/Eiffel. In *Proc. Automated Software Engineering 1998*. IEEE Press, 1998.
[15] R. Paige and J. Ostroff. A comparison of BON and UML. In *Proc. UML 1999*. Springer-Verlag, 1999.
[16] R. Paige and J. Ostroff. Developing BON as an industrial-strength formal method. In *Proc. World Congress on Formal Methods 1999*. Springer-Verlag, 1999.
[17] R. Paige and J. Ostroff. Metamodelling and conformance checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*. Springer-Verlag, 2001.
[18] K. Walden and J.-M. Nerson. *Seamless Object Oriented Software Architecture*. Prentice Hall, 1995.
[19] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4), October 1993.