Towards General Purpose, High Level, Software Languages

Anneke Kleppe

Klasse Objecten, Netherlands
a.kleppe@klasse.nl

Abstract. A highly significant benefit of MDA is that it raises the level of abstraction at which the soft-ware developer is able to work. However, the languages available to the developer have not seen much change in the last decade. Modeling languages offer high level concepts, but the pre-dominant modeling language (UML) offers too little expressive power to be able to specify a system completely. Meanwhile, the level of abstraction of most programming language con-cepts is the same as 10 to 15 years ago. Although transformation tools may to some extent bridge the gap between modeling and programming languages, in practice the developer still needs to do both modeling and programming. This means switching between the two levels of abstractions, which is difficult for most people. We argue that a general purpose, high level, software language is necessary to get MDA adopted. This language will enable any developer to focus on the problem at hand while the supporting tools - transformation tools or generators- take care of the nitty gritty details. This paper introduces an early version of such a language, which brings together a number of powerful concepts from various sources: UML, OCL, design patterns, existing programming languages, and eventually aspect-oriented languages.

Keywords: Modeling language, programming language, UML, OCL, design patterns, domain specific languages, MDA, model transformations.

1 Introduction

MDA claims amongst others the following benefits: portability, interoperability, and productivity. These benefits are all very difficult to realise. In fact, almost every hype in the last two decades promised similar benefits, most of which were not or only to small extent realised. In our opinion the only real — but highly significant — benefit of MDA is that it raises the level of abstraction at which the software developer is able to work.

In the last decade, the expressive power of programming languages has developed slowly. The latest truly innovative concept that was incorporated in a programming language, is the interface, which dates from around 1994. On the other hand, there were some very interesting new developments, like the emerge of UML, design patterns, aspect-oriented languages, and last, but not least, OCL. Each of these developments offers new high level concepts: associations, patterns, aspects, collection iterators, etc. Few of these concepts have been incorporated into pro-

A. Hartman and D. Kreische (Eds.): ECMDA-FA 2005, LNCS 3748, pp. 220 – 238, 2005.

gramming languages, which means that few of these concepts are easily available for the average software developer. If these concepts could be incorporated into a single language, this language would be very powerful, and would greatly add to the developer's ability to create the complex systems that customers demand.

This paper introduces an early version of Alan (short for A LANguage), which is a new software language that brings together a number of powerful concepts from various sources. Its aim is to bring more power to the software developer, and thereby realising one of the claimed benefits of MDA: increased productivity.

In [1] we defined 6 levels at which software development can take place. These levels are called Model Maturity Levels. Alan is a language that can be used to develop software at Modeling Maturity Level 4 or 5. At level 4 a model/program is a consistent and coherent set of texts and/or diagrams with a very specific and well-defined meaning. At level 5 the model/program contains enough information that the system can be generated completely. No adjustments need to be made to the resulting code.

Large parts of this paper, in particular sections 2 and 3, deal with the question why Alan was created. After we have explained the rationale behind Alan, we sketch the outlines of the language in section 4. Our plans for future work are presented in section 5. Section 6 contains some remarks on related work and some conclusions.

2 Rationale

We call Alan a high level, general purpose, software language. There are a large number of arguments for creating a this new type of language. We will encounter most of them as we explore the various parts of the term *high level, general purpose, software language*.

2.1 *Software* Language

Traditionally modeling and programming are viewed to be different. Differences like the ones in table 1 are commonly mentioned. Furthermore, traditionally there has been a gap between the analysis and design phase, and the implementation phase (the gap that two decades ago was supposed to be bridged by object orientation). Apparently, the expressive power of modeling languages stops somewhere along the line of the development process, and at that point the existing artefacts need to be transformed into one or more programming language artefacts, after which the development process can proceed.

Although we are living in a different era, many of the misconceptions of the previous age remain. It is for this reason that the question "what is the difference between a model and a program" pops up time and again. On the positive side, we see that the interest in MDA has brought us (at least some) agreement that both models and programs are descriptions of software systems. On the other hand MDA opens a wide chasm between platform specific and platform independent models, which at first glance appears to be just a different terminology for what used to be called models and programs. The problem here is the definition of the notion of *platform*.

Table 1. Perceived differences between modeling and programming languages

Modeling Language	Programming Language
imprecise	precise
not executable	executable
overview	detailed view
high level	low level
visual	textual
informal semantics	execution semantics
analysis by-product	end product

Fortunately, Atkinson and Kühne [2] have provided a definition of platform that crosses the divide. In their view a platform consists of the combination of a language, predefined types, predefined instances, and patterns, which are the additional concepts and rules that are needed to use the capabilities of the other three elements. Using this definition each model (or program) is bound to a certain platform. It is 100% platform specific to the language it is written in, and to the types, instances, and patterns associated with that language. In the same manner it is more or less independent of any other platform.

Anything written in this new type of language that we propose, is therefore 100% dependent upon the platform defined by such a language, and by its types and instances. If the language offers high level constructs, we may call it a modeling language. If the language is textual and/ or executable we might call it a programming language. Because this new type of language aims to combine the good aspects of both, we simply call it a *software development language*, or *software language*.

The question remains how to name the product written in a software language, should it be called *model* or *program*? The answer can be found in the fact that software languages build a bridge between programming and modeling. If a model is precise and executable, why not call it a program? Because the end result of software development has long been called *program* and because in the eyes of the developer the product written in a software language will be the end product, we choose to call it *program* as well.

2.2 *General Purpose* Language

Recently there has been much attention to the subject of domain specific languages [e.g. 3, 4]. In fact, some people argue that all MDA transformations transform domain specific languages to programming languages. In contrast, we think that there are sufficient grounds to introduce a general purpose language.

First, tool development for domain specific languages is at least as complex as for general purpose languages, whereas the potential number of users of these tools is much larger for general purpose languages. Thus, for economical reasons it is a good thing to have general purpose languages.

Second, domain specific languages are positioned as languages that can be developed by the domain experts themselves. If the supporting tools allow each

expert to define his own domain specific language, the world would see a new version of the story of the Tower of Babel [5]. None of the experts, even in the same domain, would be able to understand the language built by one of his colleagues.

Third, the line between domain specific and general purpose, or as one might say *domain independent*, is as blurred as the line between platform specific and platform independent. For instance, there are arguments to say that graphical user interface design is a separate domain; only user interfaces contain buttons and windows. On the other hand, a graphical user interface is part of almost every software system, either in the form of traditional windows and subwindows, or in the form of webpages and frames or tables.

An excellent example of what can be called a domain specific language is the Enterprise Integration Patterns language by Hophe and Woolf [6]. This language is dedicated to the domain of asynchronous messaging architectures. Again, one might argue that with the current advent of web-based systems, asynchronous messaging is part of a large number of software systems. Should such a system be built using two domain specific languages, one for the user interface and one for the messaging, combined with an ordinary programming language for the rest of the system? We think not.

2.3 *High Level* Language

Frederick Brooks argues in his book the *Mythical Man Month* [7] that the productivity of a software developer is constant in terms of the number of statements he/she produces per time unit. He states: "Programming productivity may be increased by a much as five times when a suitable high-level language is used" (page 94 of the 1995 edition). In other words, the use of a high level language could bring us one of the claimed benefits of MDA: increased productivity.

Currently there are a large number of programming languages, all more or less on the same level of abstraction. When we compare them with the level of OCL expressions, it becomes clear that it is possible to increase productivity largely. Take for example the following OCL expression:

```
partners.deliveredServices->forAll(pointsEarned = 0)
```

This expression translates to the following Java code, which means that to implement one line of OCL seventeen lines of Java are needed, as well as an extra method.

```
Iterator it = collect5().iterator();
while ( it.hasNext() ) {
    Service i_Service = (Service) it.next();
    if ( !(i_Service.getPointsEarned() == 0) ) {
           return false;
    }
}
...
private List collect5() {
    List /*(Service)*/ result = new ArrayList( *Service*/);
    Iterator it = this.getPartners().iterator();
    while ( it.hasNext() ) {
           ProgramPartner i_ProgramPartner =
```

```
                                    (ProgramPartner) it.next();
            result.addAll(
            i_ProgramPartner.getDeliveredServices());
        }
        return result;
    }
```

Contrary to programming languages, modeling languages offer constructs at a high level of abstraction. The problem with today's modeling languages is that they do not have enough expressive power. For example, how can you create a system based on a UML model without a concrete syntax (called surface language in latest UML 2 specification [8]) for actions? You can not even indicate the creation of an object.

A combination of the high level constructs of modeling languages with the completeness of programming languages seems the obvious direction for future language developments. In this we feel supported by the words of Richard Soley, managing directory of the OMG, in his foreword to MDA Distilled [9]:

> "Somehow the high level abstraction allowed by programming languages does not always have significant run-time costs, so long as the precision of the abstraction allows complete definition of the algorithm."

High level abstraction is what we should aim for.

2.4 Additional Reasons

Next to the arguments that are packaged in the term *high level, general purpose, software language*, there are two additional reasons for the development of this new type of language.

First, an important aspect of MDA is that models should be transformed automatically. The artefacts of an earlier phase in the development process should no longer be transformed by hand into the format needed in the next phase, instead this part of the software development process is to be automated. There is a debate going on whether the developer should be able to manually alter the output model after the transformation. In our view manual manipulation is currently necessary for a number of reasons. However, when MDA technology has reached maturity, manual manipulation should be an exception, just as manual manipulation of compiler generated byte code or assembler code is an exception. Transformation tools are the compilers of the next decade.

A consequence of this is that either the language of the source model must be at least as powerful as the language of the target model, or the transformation engine combined with the transformation definition must add any lacking information. If the source language has insufficient expressive power, then the output model still needs to be manually developed further. In other words, either the languages used to develop software, or the tools, need to be brought to a higher level. Because it is always wise to investigate all options, it is best to do both.

A second observation is that a key development of the last decades: the emergence of design patterns, is not truly incorporated in either modeling or programming languages. Almost ten years ago (in 1996) Budinsky and others [10] wrote:

> "Some developers have found it difficult to make the leap from the pattern description to a particular implementation, ... Others have no trouble translating the pattern into code, but they still find it a chore, especially when they have to do it repeatedly."

Since then not much has changed. At best the programming IDE offers some support, but the languages themselves have not changed. The support for patterns in the UML is not conveniently integrated. One has to draw a separate collaboration diagram to express that some classes play a part in a pattern. In practice, this is rarely done.

Hence, there are a large number of reasons to invest some effort in the development of this new type of software language. In the next section we will have a closer look at the requirements on such a language.

3 Requirements on General Purpose, High Level, Software Languages

Our new type of language should combine the positive aspects of both modeling and programming languages. So, what are the positive aspects that should be incorporated in software languages? To answer this question we take a second look at the characteristics in table 1. As shown in table 2, there are three negative aspects of modeling languages that should be avoided: non-executability of the model, the informal semantics of the modeling languages, and the model being a by-product. In the table these items have been crossed out. The other characteristics should be present in the new type of language.

Table 2. Characteristics of General Purpose, High Level, Software Languages

Modeling Language	Programming Language	Software Language
imprecise	precise	imprecise in early stages, precise in later stages
~~not executable~~	executable	executable
overview	detailed view	various levels of detail
high level	low level	high level
visual	textual	both visual and textual
~~informal semantics~~	execution semantics	execution semantics
~~analysis by-product~~	end product	end product

It is often considered convenient that a model/program may be imprecise in the early stages of development, but in later stages it should be precise. Furthermore, a model/program should be the end product and therefore, when it has reached the stage of precision, it should be executable. Thus a software language should have at least execution semantics. Different transformations may add different non-functional requirements to the end product. For instance, how the storage is arranged, whether logging is required, etc.

The visual syntax of modeling languages is often considered to be a positive aspect, but not all details can be shown visually in a convenient manner. Therefore it

would be best to have a visual syntax and a textual alternative. The visual syntax will provide overviews, whereas the textual alternative may include many of the details of the program. In the same manner it is wise to provide two textual alternative syntaxes, one that is human readable, and one that is meant to be machine readable. Languages with multiple syntaxes have been created before, one example being the Mjølner BETA language [11] developed in the early nineties of last century.

Likewise, we want to keep several views, more and less detailed. This means not necessarily that the language should provide only two different views. A hierarchy of views, each level a bit more or less detailed than the following, is to be preferred. Traditional data flow modeling as described by Tom DeMarco and Edward Yourdon [12], has an excellent levelling mechanism that has been sadly missed in the UML, although in version 2 some leveling is possible. (Data flow models had a good way of zooming in.)

When listing these aspects it is sometimes difficult to determine whether a certain aspect should lead to the creation of language concepts, or whether it should lead to specific support in the development environment, the IDE. The following list is ordered by the influence the re-quirement has on on the language itself.

1. The modeling language should have a several concrete syntaxes that are all mapped onto the same abstract syntax.
2. The modeling language should have a clear semantics for precise programs. As long as the program is imprecise, the semantics may be unclear.
3. The program will be a complete functional description of the the system.
4. The developer should be allowed to be imprecise at certain stages of the development proces.
5. The program should be precise at most stages of development, specially during the last stages, when it is prepared to be used as input to a transformation.
6. If the program is imprecise it will very likely not be executable, but when all details are present, it should be executable (model simulator, model virtual machine)
7. The modeller should be able to have different views on the same program: overview and more detailed.

And surely, we should not forget the ultimate requirement, because otherwise the new language will be nothing more than a programming language in pictures: the language should provide constructs that are more abstract than current day programming languages.

4 ALAN: A Software Language

The goal of the Alan project is to gather and combine the concepts that are already well-known and have shown their use, into a format that is usable for a developer, not to create new software development concepts. Therefore, we use a number of different sources. The first source is UML [8, 13], for instance, the two-directional association is a powerful concept that is not present in current day programming languages. The second source is OCL [1, 14]. The possibilities OCL offers on collections are far more powerful than any programming language offers. The third

source is design patterns [15], which beyond a doubt have been a landmark in software development. Currently, support for patterns can be found in some IDEs, but little support can be found in languages. The fourth source is found in current day programming languages, like Java, C#, and Python.

In the future we hope to include constructs from aspect-oriented languages as well. In our opinion a user of Alan should be able to define a number of cross-cutting aspects and weave them into a single output. Whether this effects the design of the language, or merely the design of the Alan IDE, is a question that remains to be answered.

What we present in this paper is an early version of Alan. Our ideas need to be developed further, but we feel it is already worthwhile to share them with the community and get some feedback. In the following sections we will present examples of language constructs from the above mentioned sources and explain how they are incorporated. The length of this paper does not allow us to be complete. Alan comprises more than just the examples given below. The language has been fully implemented and the Alan IDE and compiler are available as an Eclipse plug-in. This paper does not merely describe ideas, everything presented here was tested in our implementation, and shown to be feasible.

Alan's textual syntax is based on the Java syntax. One notable difference is that the equal sign is reserved for comparisons; assignments are denoted using the Pascal notation (":="). Alan's visual syntax is basically the same as the UML class diagram syntax. However, the semantics of Alan are much more strict than those defined for UML. The semantics of Alan are defined by a mapping to Java. This mapping is implemented as an MDA transformation.

4.1 UML Constructs in Alan

Apart from having used the syntax of the UML class diagram for the Alan visual syntax, we have borrowed a number of UML constructs. Some of which are explained in the following sections.

4.1.1 Associations

The UML association is a powerfull construct that needs to be implemented carefully. Specially, the two-directional association leads to complicated code, because setting the field that implements the association in the class at one end, must also ensure that the field that implements the other association end has the correct value.

In Alan, associations are always two-directional. A uni-directional association, as is present in the UML, is in Alan simply an attribute, or in ordinary programming terminology: a pointer. In our view, mixing the concepts of a reciprocal relationship and a reference, as is the case with the UML association, is confusing and, to some extent, overkill. As we explained elaboratedly in [16], if Eve is in the bag of Adam's girlfriends, then Adam must be in the of Eve's boyfriends, otherwise one could not speak of a relationship called friendship. If Eve insists in calling Adam her boyfriend, even though Adam does not regard Eve to be his girlfriend, then this fact can only be represented as a reference from Eve to Adam (or to a bag of not-interested boyfriends including Adam).

In Alan associations may have no more than two ends, and each association abides to the following characteristics, which we call the ABACUS rules.

- **Awareness:** Both objects are aware of the fact that the link exists.
- **Boolean existence:** If the objects agree to end the link, it is dissolved at both ends.
- **Agreement:** Both objects have to agree with the link.
- **Cascaded deletion:** If one of the objects is deleted, the link is dissolved as well.
- **USe of rolenames:** An object may refer to its partner using the role name provided with the association.

A simple example of associations depicted using the visual syntax can be found in figure 1. The Alan textual syntax that maps to the same abstract syntax is the following.

```
class Man {
   public Woman wife otherside husband;
   public Bag[Woman] girlfriends[1..10] otherside boyfriends;
   ...
}
class Woman
   public Man husband otherside wife;
   public Bag[Man] boyfriends otherside girlfriends;
   ...
}
```

The multiplicities in the figure need not always be part of the textual syntax for associations. In Alan the exact lower and upper bounds need only be present when they differ from 1..* or 1. The exact lower and upper bounds of multiplicities are considered to be invariants, which will be explained in section 4.2.3. Currently Alan does not support association classes. We are investigating if and how this could be done.

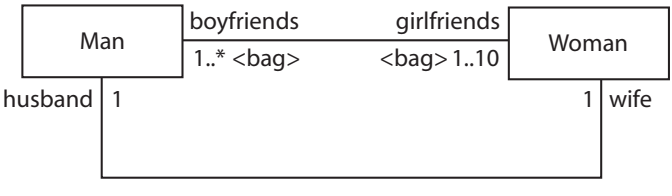

Fig. 1. An Alan association example

4.1.2 Enumeration Types

Typesafe enumeration types may not look like a powerful language construct, but in practice they come in very handy. However, implementing a typesafe enumeration type is not a simple matter. Joshua Bloch spends as much as 10 pages on this subject in his book Effective Java [17]. Still, when you have learned the trick, you see that every enumeration type can be handled in the same fashion. This is where the MDA transformation techniques can provide much assistance: a single line in a higher level

language can be automatically transformed into a much more verbose text in a lower level language.

In Alan, typesafe enumeration types can again be written in the same simple way that they were written in C, while the translation of Alan into Java takes care of all the details of implementing the type safeness. An example of a declaration of a typesafe enumeration type:

```
enum myColor { red; white; blue }
```

This declaration may be part of a package, and thus have package scope, or it may be part of a class declaration, and thus have class scope.

4.1.3 Composite or Aggregate Objects

Another example of a higher level construct that is part of UML, but not part of any programming language, is the composite object. Much has been said on the semantics of the UML aggregate and composite, e.g. in [18]. The Alan composite object has deletion semantics; when the container is deleted, so are all its parts. Furthermore, although other objects may refer to the object by means of an attribute or association, this object may be part of no more than one composite object. Figure 2 contains a simple example of an Alan composite object. The code below is the textual alternative.

```
class Bike {
    public part Set[Wheel] wheels[2];
    public part Frame frame;
    ...
}
class Wheel {
    public part Tire tire;
    ...
}
```

As for associations, each part in a composite object is aware of the link to its owner. In fact, the role name *owner* may be used in the part object to indicate the containing object. If instances of the same class may be part of multiple composite objects, as for instance the class *Wheel* may be used as part in a class *Car* as well as the class *Bike*, the *owner* always refers to the one composite object that contains the specific instance.

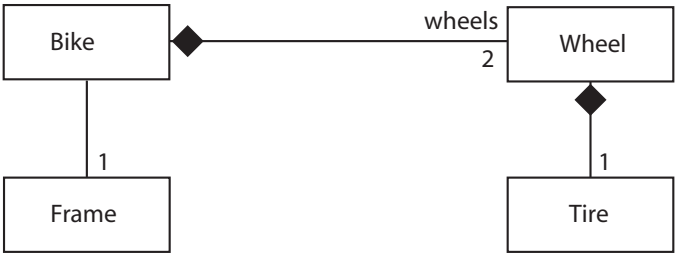

Fig. 2. An Alan composite object example

Although on the surface the notion of the composite object may appear to be nothing more than a more specialized version of the association, it serves a larger purpose. It enables us to easily specify the Visitor pattern, as explained in section 4.3.1. Furthermore, we are experimenting with a specific form of delegation, in which operations offered by the part objects become available in the composite object. A call to such an operation on the composite object will delegate the call to the part object or objects that implement it. For instance, when the operation *turn* is specified in the class *Wheel*, a call to a *Bike* object, as in *myBike.turn()*, will delegate this call automatically to both wheels.

4.2 OCL Constructs in Alan

The constructs that were defined in OCL, are incorporated in Alan completely, but Alan takes things one step further. For instance, OCL expressions may be used in statements, like assignments, and concepts like invariants are integrated in the textual syntax for the definition of a class. Again, this is not a new idea, it has been done in, for instance, Eiffel. What makes Alan different is the combination of existing ideas.

4.2.1 Primitive Types

The primitive types that are available in Alan are the same as the UML/OCL primitive types: *Integer, Real, String,* and *Boolean*. We believe that the abstraction level that Alan targets, has no need for low level details, like the differences between *char[]* and *String*, between *float* and *double*, and between *int* and *long*.

4.2.2 Collection Types and Iterators

OCL defines four collection types: *Set, OrderedSet, Sequence,* and *Bag*. These types are also available in Alan. Furthermore, the iterators defined on OCL collections, like *select, collect, exists,* and *isUnique*, are all part of Alan as well. In the future, we hope to augment Alan with a syntax for defining new iterators, to enable users to specify their own iterators in terms of existing ones. The example in section 2.3 shows clearly the power that OCL collection types and iterators bring to Alan. Because the available collection types reside at a much higher level of abstraction, Alan does not support arrays.

Furthermore, we are investigating two additional collection types: the *SortedSet* and the *SortedBag*. Both should sort their elements based on the "natural" order of the element's type, defined by the *equals, greater-than,* and *smaller-than* operations.

4.2.3 Invariants and Pre- and Postconditions

As can be expected, Alan supports OCL invariants and pre- and postconditions completely. In the textual syntax they are integrated in the class definition, as in the following example.

```
class Man {
   public Woman wife = oclUndefined otherside husband;
   public Bag[Woman] girlfriends[1..10] otherside boyfriends;
   public Integer age = 0;
   private Real moneyEarned = 0;
   inv ofAge: age < 16 implies wife = oclUndefined;
```

```
    public void work()
    pre: age >= 14
    post: moneyEarned =
              moneyEarned@pre + 100 and notassert(girlfriends-
              mult)
   {
        ...
    }

}
```

All invariants are checked at postcondition time of all operations of the class, as well as after the setting of the value of an attribute. If the developer needs to speed up processing, he can choose to check only some invariants or none at all by using the keyword *notassert*. The *notassert* takes as parameters the names of the invariants that should not be checked. If no parameters are given, then none of the invariants will be checked.

In section 4.1.1 we mentioned that the upper and lower bound of the multiplicities for an association are considered to be invariants. This means that the bounds will also be checked at postcondition time of any operation execution. When this check is not necessary, this too can be indicated in the *notassert* clause. As name of the invariant the role name of the association end concatenated with "-mult" may be used, as in *girlfriends-mult*.

4.2.4 Derivation rules

OCL derivation rules can be expressed in Alan as well. Before the execution of an operation of the class, the value of the derived attributes is determined, during execution this value remains the same. The next example contains two examples of derived attributes: *frontwheel*, and *speed*.

```
class Bike {
   public part Sequence[Wheel] wheels;
   public derived Wheel frontwheel := wheels->first();
   public derived Real speed :=
         frontwheelsize * frontwheel.revolutionsPerSec;
     ...
}
```

4.3 Design Patterns in Alan

Currently, only the most popular patterns are incorporated in Alan. We expect to add to this list in the future.

4.3.1 Visitor

The visitor pattern in Alan is linked to the composite object concept. Only composite objects can be visited. In general composite objects have the form of a directed graph; only some of the graphs will have the form of a tree. The graph is not necessarily acyclic, therefore the implementation algorithm ensures that the execution will terminate. Figure 3 shows an example, where node 3 is about to be visited twice, once

caused by the link with node 1 and once caused by the link with node 5. Here the algorithm ensures that node 3 will not be visited because of its link with node 5.

The textual syntax for the visitor pattern is shown in the next example. The visual syntax shows only the fact that the class *BikeVisitor* is a visitor to *Bike* objects, the details are not shown visually.

```
class BikeVisitor visits Bike <breadthfirst> {
    String visit(Bike bike) {
            String brand;
            before {
                    brand := bike.getDefaultBrand();
            }
            after {
                    brand := resultOf(bike.frame);
                    brand + resultOf(bike.frontwheel);
                    brand + resultOf(bike.wheels->last());
                    return beautify(brand);
            }
    }
    String visit(Wheel wheel) {
            after {
                    return wheel.brand + resultOf(wheel.tire);
            }
    }
    String visit(Frame frame) {
            return 'frame';
    }
    String visit(Tire tire) {
           return 'dunlop';
    }
    String beautify(String brand) {
           ...
    }
}
```

The keyword **visits** indicates that instances of this class are visitors of composite objects of type *Bike*. Directed graphs can be transversed in several ways. The most

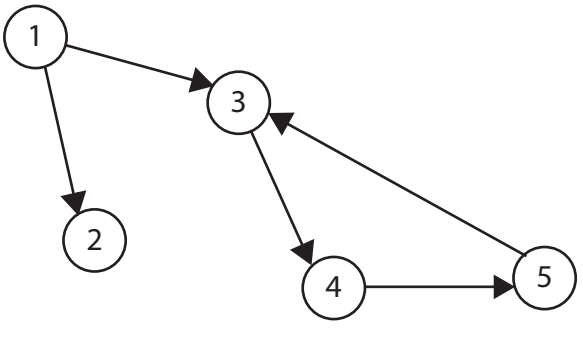

Fig. 3. An Alan composite object as directed graph

important traversal methods are breadthfirst and depthfirst. The visitor in the example visits *Bike* objects breadth-first, that is it visits all direct parts of the *Bike* instance before visiting the parts of the *Wheel* instances. Alan also supports visitors that use the depthfirst method.

Any visit operation is divided into two parts. Before visiting parts of an composite instance the statements in the *before* clause are executed. After visiting parts of the instance the statements in the *after* clause are executed. Classes that are part of the composite, but are not composite objects themselves, are the leaves of the tree or directed graph. For them the distinction between the before and after clause cannot be made.

Visit operations may have a result. In the example all visit operations have a String result. This result can be used in the after clause of the visit operation that visits the containing object, using the keyword **`resultOf`**. The type of the value that is returned by **`resultOf`**, is the type of the corresponding visit operation. If, in the example, the visit of a Tire instance would have re-turned an Integer value, then the expression **`resultOf`**`(wheel.tire)` in the visit operation for Wheels would have returned an Integer as well (which would have resulted in a type error).

The visitor class need not define a visit operation for all nodes in the composite graph. When a visit operation for a certain type of node is not present, the traversal algorithm simply proceeds. Next to the visit operations, visitors may have 'normal' operations as well. The operation *beautify* is an example.

Visiting may start at any node within the directed graph. You simply create a visitor instance and tell it to start visiting a certain object. If the object is not within the composite object that the visitor was defined for, a type error occurs. The next example shows how the *BikeVisitor* can be used.

```
myBike := ...;
visitor := new BikeVisitor( );
if (visitor.visit(myBike).equals( "someString")) { ... }
System.out.println(visitor.visit(myBike.frontwheel));
visitor.visit(myBike);
```

4.3.2 Singleton

Another popular design pattern is the singleton pattern. This pattern is easy to use in Alan, one extra keyword suffices, as shown in the next example. The output of this example is, of course, twice the String 'this is the unique instance of MyFirstSingleton', followed by two occurrences of 'changed name of unique singleton'. Note that the singleton user is unaware of the fact that it is using a singleton instance, which is different from the use of a singleton in e.g. Java, where you cannot use "new", but you have to use a specific class method to get the instance.

```
singleton class MyFirstSingleton {
    public String name :=
       'this is the unique instance of MyFirstSingleton';
    ...
}
```

```
class SingletonUser {
    public useSingleton() {

            MyFirstSingleton a := new MyFirstSingleton();
            System.out.println(a.name);
            MyFirstSingleton b := new MyFirstSingleton();
            System.out.println(b.name);
            a.name := 'changed name of unique singleton';
            System.out.println(a.name);
            System.out.println(b.name);
    }
}
```

4.3.3 Observer

The third pattern that is incorporated in Alan, is the Observer pattern. The key to this pattern are two predefined operations that are available on every class: *observe* and *disobserve*[1]. The *observe* operation takes as parameters the object to be observed, and the name of the operation that should be called whenever a change occurs in the subject. This operation must be defined in the class of the observer, and it must have one parameter, which type is the type of the object to be observed. The *disobserve* operation takes as parameter the object that should no longer be observed. The next example defines a simple observer that observes two other instances, one of type *Subject*, and one of type *OtherSubject*.

```
class MyFirstObserver {

    public start() { Subject mySubject1 :=
            new Subject(); OtherSubject
            mySubject2 := new OtherSubject();

            System.out.println('>>>observing
            mySubject1');self.observe(mySubje
            ct1, 'uponChange1');
            mySubject1.attr := 'blue';
            mySubject1.attr := 'red';

            System.out.println('>>>observing
            mySubject2');self.observe(mySubje
            ct2,
            'uponChange2');mySubject2.attr :=
            "black';mySubject1.attr :=
            'green';

            System.out.println('>>>DISobserving
            mySubject1');self.disobserve(mySubje
```

[1] The name *disobserve* is still under debate. Other options considered are *unlink*, and *unsubscribe*. The term *disobserve* is closely related to the term *observe*, which seems preferable.

```
            ct1);mySubject1.attr :=
            'white';mySubject2.attr := 'yellow';
        }
        public uponChange1(
        Subject mySubject ) {System.out.println("The value of
        Subject.attr is "
                                        + mySubject.attr);
        }
        public uponChange2(OtherSubject mySubject)
            {System.out.println("The value of
            OtherSubject.attr is " + mySubject.attr);
            }
            ...

    }
```

The output of operation *start* is:

```
>>>observing mySubject1
The value of Subject.attr is blue
The value of Subject.attr is red
>>>observing mySubject2
The value of OtherSubject.attr is black
The value of Subject.attr is green
>>>DISobserving mySubject1
The value of OtherSubject.attr is yellow
```

4.4 Programming Language Constructs in Alan

Most of the constructs known in programming languages are also present in Alan, although some have been discarded because their level of abstraction was considered too low. A few programming language constructs in Alan deserve more attention. They are explained in the following sections.

4.4.1 Generic Types

Only few programming languages support generic types. Alan offers full support, in fact, the collection types are considered to be predefined generic types. Generic types may be defined independently of any other types, but, as the next example shows, one may also define a new generic type by inheriting from one of the collection types.

```
class MySetType [ TYPEVAR ] extends Set [ TYPEVAR ] {

    public attr : TYPEVAR;
    public setAttr : Set[TYPEVAR];

    ...

    public oper1(newV : TYPEVAR) : TYPEVAR {
            attr := newV;
            return res;
    }
}
```

4.4.2 Visibility and Set and Get Operations

In Alan, explicit definition of *get* and *set* operations for attributes is not necessary. When these operations are not defined for a certain attribute, they will be generated according to the visibility of that attribute. If the developer wants to execute some extra statements and/or checks in the *get* or *set* operation, he may define the operations himself. This is similar to properties in C#.

4.4.3 Loops

In Alan, the OCL iterators are available for many of the cases where you would normally use a loop construct in a program.Therefore, the need for loop constructs in Alan will be much less than in one of today's programming languages. However, we still need a loop construct for some special cases, like simply doing the same thing for a fixed number of times.

Alan provides two primitive loop constructs: the for-loop and the while-loop. The for-loop must be used with two Integer values separated by two dots, that indicate the lower and upper bound of the number of times the body of the loop must be executed. The while-loop takes a boolean expression as guard, as in the next examples.

```
for(1 .. someInt) { ... }
while(someBoolean) { ... }
```

5 Future Work

As explained earlier, this paper describes an early version of Alan. Many aspects of the language still need to be fleshed out. We have already mentioned the inclusion of constructs from aspect-oriented languages, the support for association classes, and support for other patterns. Another issue are the libraries that should accompany this language, which should include extra predefined types like the *SortedSet* and *SortedBag*. Key to the success of Java has been the enormous number of predefined types available. We are convinced that languages like Alan will need a similar set of libraries, which should also be full of higher level constructs ready to use.

The Alan IDE is currently being implemented as an Eclipse plug-in. Because of the ongoing work the Alan IDE is not yet available for a large audience, but if you want

Table 3. The realisation of the requirements by Alan

Software Language	Alan
imprecise in early stages, precise in later stages	yes, visual syntax allows imprecision, textual syntax does not
executable	yes, by translation to Java code
various levels of detail	not yet established
high level	yes!
both visual and textual	yes!
execution semantics	yes!
end product	not yet, but going strong

to have some idea of the Java code being generated, you can take a look at the Java code generated by the OCL tool Octopus [19]. The code for the associations and for the OCL expressions is the same.

What remains is a check to see whether the requirements we defined in section 3 are met by Alan. Alan meets almost all of the requirements, as shown by table 3. We strongly believe that it is only a matter of time (and hard work) to realise the remaining requirements, and that in the near future Alan will be the general purpose, high level software language that we envisioned.

6 Conclusion and Related Work

On the topic of related work we can be short. A lot of work is being done in the area of domain specific languages, e.g. [3, 4], including work in the area of Executable UML [20], as well as in the area of formal specifications [21, 22], but virtually none is done in the area of general purpose, high level software languages. One might argue that we too have created a DSL, one dedicated to programming, but in our view this argument stretches the concept of domain far too much. If programming itself can be identified as a domain, then COBOL, Ada, and all other programming languages should also be called DSLs.

Currently, it is possible to create a complete visual representation of Java, or any other programming language in UML. This fact does not in any way diminish the need for a general purpose, high level software language. The essence of Alan is not that it combines a visual and a textual representation, as stated in section 3, this has been done successfully before. Instead Alan's merits lie in the fact that it incorporates higher level concepts, and makes them available to the programmer in a way he or she is likely to understand.

Please note that the creation of general purpose, high level, software languages will not make informal models obsolete, it will just raise the level of abstraction. This is a normal phenomenon in the history of any technology (or culture). Old ways become the stepping stone for future developments. In the same way current day middleware will, in time, take its place on one of the lower levels of our technology stack.

Raising the level of abstraction does not mean that the old ways are crooked or misformed. One has to build a wall by putting in the first stone, which will support all the other stones. Therefore, the first few stones need to be solid and well fitted. By creating Alan we do not criticize any other technologies, for instance, Java 5 has done some good work on enumeration types. We simply argue that it is time to start building the next layer of stones, and that the next layer should include general purpose, high level software languages.

We hope that Alan will not be the only software language at this level. Software development needs the boost that this new type of language can give. Therefore, in the future we hope to see a large family of general purpose, high level, software languages.

References

[1] Anneke Kleppe, Jos Warmer, *The Object Constraint Language Second Edition, Getting Your Models Ready for MDA*, 2003, Addison-Wesley

[2] C. Atkinson and T. Kühne, "A Generalized Notion of Platforms for Model Driven Development", in *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*, ed. S. Beydeda and V. Gruhn, Springer Verlag. 2005.

[3] Jack Greenfield and Keith Short with Steve Cook and Stuart Kent, *Software Factories, Assembling Applica-tions with Patterns, Models, Frameworks, and Tools*, Wiley, 2004

[4] Alexander Felfernig e.a., *UML as Domain Specific Language for the Construction of Knowledge-Based Con-figuration Systems*, InInternational Journal of Software Engineering and Knowledge Engineering, Vol.10 No. 4 (2000) pp. 449 - 469, World Scientific Publishing Company

[5] The Bible, Genesis 11: 1-8

[6] G. Hophe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003.

[7] Frederick P. Brooks, The Mythical Man-Month, Addison-Wesley, 1995

[8] *UML 2.0 Superstructure Specification*, OMG document ptc/04-10-02, October 2004

[9] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise, *MDA Distilled, Principles of Model_Driven Ar-chitecture*, Addison-Wesley, 2004

[10] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu, *Automatic code generation from design patterns*, IBM Systems Journal, 35(2), 1996.

[11] *Object-oriented environments: The Mjolner approach*, Editors: Jorgen Lindskov Knudsen (Aarhus University, Denmark), Mats Lofgren (Telia Research AB, Sweden) , Ole Lehrmann Madsen (Aarhus University, Denmark), Boris Magnusson (Lund University, Sweden) , Prentice Hall, 1994

[12] T. DeMarco, P.J. Plauger, *Structured Analysis and System Specification*, Prentice Hall, 1985

[13] *Unified Modeling Language (UML) Specification: Infrastructure*, OMG document ptc/04-10-14, October 2004

[14] UML 2.0 OCL Specification, OMG document ptc/03-10-14, October 2003

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Ob-ject-Oriented Software*, Addison-Wesley, 1995

[16] Anneke Kleppe and Jos Warmer, *Wed Yourself to UML with the Power of Associations, part 1 and 2*, online publication at http://www.devx.com/enterprise/Article/28528 and http://www.devx.com/enterprise/Article/ 28576

[17] Joshua Bloch, *Effective Java, Programming Language Guide*, Addison-Wesley, 2001

[18] Brian Henderson-Sellers and Franck Barbier, *Black and White Diamonds*, in "UML" '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 1999. Proceedings, Editors: R. France and B. Rumpe, LNCS 1723, pp. 550 - 565, Springer-Verlag, 1999

[19] Octopus: OCL Tool for Precise Uml Specifications, available from http://www.klasse.nl/english/research/octopus-intro.html

[20] Stephen J. Mellor and Marc J. Balcer, *Executable UML, A foundation for Model-Driven Architecture*, Addison-Wesley, 2002

[21] A. Nymeyer and J-P. Katoen, *Code generation based on formal bottom-up rewrite systems theory and heuris-tic search*, Acta Informatica, Vol. 8, pages: 597 - 635, 1997

[22] V.M. Jones. Realization of CCR in C. in Bolognesi T, Van de Lagemaat J and Vissers C.A. (eds.), LOTO-Sphere: Software Development with LOTOS, pp. 348-368, Kluwer Academic Publishers, 1995.