

FLEXIBLE INCREMENTAL DEVELOPMENT BY INTEGRATING SPECIFICATION AND CODE

Patrick Schmid and Christine Hofmeister
{pds2, crh5}@lehigh.edu
Department of Computer Science and Engineering
Lehigh University
19 Memorial Dr.
Bethlehem, PA
USA

Abstract

This paper describes the use of executable specifications to ease incremental development, by providing more flexibility regarding the order in which modules are implemented. In this approach, the architect provides a specification for each module in the module architecture view. While specifications must precisely describe the interactions among modules, they may abstract some of the functionality. These specifications are executable, and interoperate seamlessly with the module implementations. For incremental development this makes the contents of a release more flexible, makes the order of implementation of modules more flexible, and reduces overall effort by reducing or eliminating the need for stubs, drivers, or temporary prototypes of modules. Our approach uses executable specifications written in AsmL and uses the .NET framework for integrating them with implementations.

Key Words

incremental development, executable specification, AsmL, software architecture

1. Introduction

Over the years different software process models have been introduced, and the trend in recent years is towards those with an incremental approach. Rather than developing the entire system at once, or developing it by layer, the goal is to produce successive working versions of the system, increasing the functionality with each release.

There are good reasons to follow this model: early releases are appreciated by the customer, are encouraging to the development team, and give early validation of the design approach. However, it can be quite difficult to partition the system into these releases while satisfying all the dependencies among its components and accommodating the staffing constraints. The overall effort may be higher, since there is likely an increased need for stubs, drivers, and temporary prototypes of certain components.

Executable specifications of the components could mitigate these drawbacks. In the past executable specifications have been used for detecting inconsistencies in the specification, for prototyping, for systematic transformation to an implementation, and for software estimation [1]. To serve incremental development, an executable specification of a component must be interchangeable with the implementation of that component; specifications and implementations must interoperate seamlessly.

A criticism of formal specification languages is that they are often *vertically isolated*. This means that the specification has no formal relation with anything upstream in the development process, e.g. requirements specifications, nor downstream, e.g. implementation [1].

Our use of executable specifications for incremental development makes the specification less vertically isolated, by providing the downstream integration. We do so by using an executable specification for each component in the system, and a framework that allows us to integrate these specifications with the implemented components, thus making specification and implementation interchangeable. To accomplish this we use AsmL, a specification language developed by Microsoft Research¹.

In this paper we present our initial study of the feasibility of this approach. After summarizing related work in Section 2, we describe the approach in Section 3. Section 4 describes a detailed example, and Section 5 concludes the paper, discussing the advantages and disadvantages of this approach and future directions.

2. Related Work

The transformational implementation approach, described in [3], is related to ours. In the former one, a specification is created and then in a stepwise process semi-automatically transformed into an implementation. It requires a transformation library that describes mathematically how to transform each possible specification

¹ www.research.microsoft.com/fse

element into an implementation. It is therefore very difficult to use in practice compared to our approach.

[4] combines the transformational implementation approach with an iterative software development process, in which an executable specification is used to produce successive prototypes that are presented to the user. User feedback refines the requirements definition. A new specification is created which is then mathematically verified against its predecessor. This iterative process continues until one arrives at an implementation. The main focus of this approach is to provide a framework for the rigorous development of programs. Our approach however strives to improve existing iterative development approaches by providing more correctness through a specification language and an easier integration.

[5] describes a stepwise and incremental development process along the line of rapid prototyping. It is based upon stepwise transformations on formal developments. As the other approaches described above, it is mostly concerned with mathematical correctness of a development project.

Design by Contract, pioneered in [6], refers to a design methodology in which every computational entity is constrained by a contract. A contract consists of pre- and post-conditions, as well as invariants. [7] and [8] incorporated Design by Contract into the .NET Framework. Design by Contract is concerned with the correctness of a program and provides run-time enforcement of a contract with limited functionality. A specification can be run instead of the implementation, and not just with it. It also provides more functionality.

3. Our Approach

The modern approach to software design is to use multiple software architecture views, each describing the system from a particular perspective. One of these views describes the system in terms of modules and the static usage-dependencies among the modules. In [9] this is known as the “Module Architecture View”, and in [10] this is known as the “Uses” style of the “Module” view-type.

Our approach is to use a specification language to describe each module in the module architecture view. Each specification describes the functionality of the module and its interactions with other modules. In defining the functionality, the architect may choose to abstract certain aspects of the functionality.

The interactions, however, should be completely described. The architect must specify the interface(s) the module provides, along with preconditions to enforce constraints on the usage of the interface. The architect must also specify what the module requires. There is no interface mechanism to describe this, so the specification

must use the same interfaces as the implementation, and should ideally follow the same pattern of usage as the implementation.

While each module has exactly one associated specification, its implementation typically consists of multiple classes. Thus abstraction is at work here also.

To support this approach two requirements must be fulfilled:

1. An executable specification of the software is necessary.
2. The executable specification language must use the same framework, e.g. Microsoft .NET or Java VM, as the implementation.

The executable specification of the software is the basis for the implementation. Instead of starting with nothing, we start with the specification and replace module by module of it with the implementation. This allows system testing at every integration step. In addition, a “working” version of the software is available from the beginning on.

It can be argued that devising such a specification and then writing the implementation constitutes double the work. This argument is very valid, and therefore a specification should always be an abstraction of an implementation. For example, a module could be represented in a specification by one class, but consist of multiple classes in the implementation.

The second requirement, the framework, is crucial. The combination of the specification with the implementation cannot occur on the source code level. Therefore our proposed, seamless, swapping of specification and implementation modules has to occur on the binary level. An operating system could support this, e.g. in the form of Windows COM, but this is quite frequently cumbersome and difficult to use. Frameworks, such as Microsoft .NET or the Java environment, are much better suited for binary level integration.

The framework also needs to ensure type compatibility between modules written in different languages. This implies that method signatures do not change between specification and implementation. This property has to be supported by the framework and taken into consideration by the developer.

AsmL from Microsoft Research is a specification language that fulfills both requirements. It is a formal executable specification language built upon the Microsoft .NET framework. The next two sections introduce both briefly. The discussion of our example follows.

3.1 Microsoft .NET Framework

The Microsoft .NET Framework is a software platform developed by Microsoft. At its heart lies the Common

Language Runtime (CLR) which executes Common Intermediate Language (IL) code using just-in-time compilation. The CLR serves the same purpose as Java's Virtual Machine, except not for Java Byte code but for IL code. Situated above the CLR are the .NET Framework classes that provide common system services, database access and many more things. Those classes can be used in Visual C++ .NET instead of the Standard Template Library for example.

A compiler for a .NET language generates IL code in the form of an *assembly*. .NET provides language interoperability through these assemblies, as every language can use any other assembly. For example, an application could be composed of assemblies written in VC++, VB, C# and J#. To the user, such an application would appear as if it was written in a single language, e.g. no performance differences are detectable between assemblies. In addition, a developer uses an assembly the same way no matter what language it was written in.

Microsoft .NET differentiates between *managed* and *unmanaged* code. Managed code fully conforms to the framework, e.g. it always uses the CLR garbage collection, and is fully compiled into IL. Unmanaged code may only conform to parts of the framework, or may not even fully compile into IL. Most .NET languages only support managed code. VC++ is the only .NET language that supports both in the same source file. The C++ compiler will attempt to generate IL code for the unmanaged source code portions, but if not possible will generate native x86 code wrapped by IL code. For an in-depth discussion of the managed extensions for VC++, see [11].

Besides the languages mentioned above which are provided by Microsoft, there are a lot of other languages available for .NET. For a thorough discussion of the CLR and language integration, see [12].

3.2 AsmL for Microsoft .NET

AsmL is Microsoft Research's Abstract state machine Language. It is an executable specification language and a full .NET language, which is the key for our approach to incremental software development.

The syntax of the language, as seen in the example below, should look somewhat familiar to C++, C# or Java programmers. In contrast to those languages however, there are no symbols that indicate blocks. Blocks are solely determined by indentation. The unfamiliar "step" statement marks the beginning of a new block that will be executed in parallel when the next step statement is encountered. In other words, a state transition occurs, which in essence treats all statements as one transaction. AsmL has one global state, which is broken into smaller sub-states by nested step statements. As the concept of a global state does not work in an environment where normal programming languages are mixed with AsmL, we

chose to treat every module as its own separate state machine. This can be achieved in AsmL by adding the statement "[EntryPoint]" before a class.

AsmL is a powerful specification language. In addition to its transaction based concept, it provides data abstraction through high-level data structures, correctness conditions, a very strong type system and feature abstraction. For a further discussion of the language and its applications see [13], [14] and [15]. The following example demonstrates our approach with AsmL. AsmL syntax that is not intuitive is explained in the text.

4. Example

Our chosen example is a simple magnetic swipe-card authentication system. The system uses different numerical levels of security access with 0 being the lowest one and 10 the highest one. The user specifies a desired access level and then swipes his card. The systems extracts the ID from the string stored on the swipe card and compares it to a repository of IDs and allowed access levels. If the user has a high enough access level, the system grants access; otherwise access is denied.

4.1 AsmL Specification

There is one specification per module. In our example each specification consists of one class in one namespace. We do not use an explicit interface for the module, although in general we would recommend this. Our module hierarchy is very simple: ClassA is on the top and represents the user interface. ClassB is directly below it and it uses ClassC and ClassD.

```

namespace namespace_ClassA
import namespace_ClassB

[EntryPoint]
public class ClassA
private var Authenticator as ClassB = new ClassB()

public run()
step Write("Please enter the security level you wish to access: ")
step intLevel as Integer = ToInteger(ReadLine())
step Write("Please swipe your ID card: ")
step strCard as String = ReadLine()
step
if (Authenticator.HasAccess(strCard, intLevel)) then
WriteLine("Access Granted")
else
WriteLine("Access Denied")

```

Figure 1 AsmL Specification for ClassA

Figure 1 shows the AsmL specification for ClassA. The class first instantiates ClassB, and then asks the user for the desired level of access. After that, the user is asked to swipe his card. ClassA then calls ClassB to determine whether the user has access to that level, and either prints out "Access Granted" or "Access Denied". For this example, the swipe card string is assumed to be a four digit number representing the ID number of the user.

```

namespace namespace_ClassB
import namespace_ClassC
import namespace_ClassD

[EntryPoint]
public class ClassB
  private var Store as ClassC = new ClassC()
  private var Extractor as ClassD = new ClassD()

  public HasAccess(strCard as String, intLevel as Integer) as Boolean
    require strCard.Length > 0 and intLevel >= 0
    return
      (Store.getAuthenticationLevel(Extractor.getID(strCard))>=intLevel)

```

Figure 2 Asml Specification for ClassB

```

namespace namespace_ClassC

[EntryPoint]
public class ClassC
  private var IDs as Map of Integer to Integer = {1234->4, 2323->2,
1212->10, 4534->7};

  public getAuthenticationLevel(ID as Integer) as Integer
    require ID > 0 and then IDs.Contains(ID)
    ensure result >= 0 and result <= 10
    return IDs(ID)

```

Figure 3 AsmL Specification for ClassC

Figure 2 shows ClassB. ClassC, in Figure 3, acts as repository for the user IDs and their associated access levels. It defines a map of integers in which the user ID can be looked-up and the associated access level retrieved. Data structures like this map are one of the previously mentioned strengths of AsmL. In this specification, we pre-initialize the map with data, which is sufficient for the size of the example. It is an abstraction (simplification) of the full behavior. The “require” statement represents a pre-condition, the “ensure” statement a post-condition. Execution of the specification is stopped and an error message displayed if a condition fails.

```

namespace namespace_ClassD

[EntryPoint]
public class ClassD
  private const intBegin as Integer=0
  private const intLength as Integer=4

  public getID(strCard as String) as Integer
    require strCard.Length >= intLength and then strCard.Length >=
intBegin + intLength
    ensure result > 0
    return ToInteger(strCard.Substring(intBegin, intLength))

```

Figure 4 AsmL Specification for ClassD

Figure 4 shows the class that is used to retrieve the ID number from a swipe card string. It extracts the substring, specified by its beginning and length, from the provided swipe card string that represents the ID number. As mentioned above and reflected in the values of the two constants, the example expects a four digit number and not a real swipe card string. Figure 5 below shows the last piece necessary to achieve an executable specification, namely Main().

```

namespace AuthenticationExample
import namespace_ClassA

Main()
  step A as ClassA =new ClassA()
  step A.run()

```

Figure 5 Asml Specification Main()

4.2 First Incremental Development Step

We chose to use Visual C++ 2003 .NET as our implementation language. We decided to first implement ClassC.

```

#pragma once
using namespace System;

namespace namespace_ClassC {
  public __gc class ClassC {
  public:
    ClassC();
    int getAuthenticationLevel(int ID);
  };
}

```

Figure 6 C++ Header for ClassC

Figure 6 shows the C++ header file for ClassC. Besides the compiler directive in the first line, the using statement following it and the “public __gc” it looks like an ordinary C++ header. The first two items were automatically added by VC++, the last one declares ClassC as a managed class using garbage collection that is public.

```

#include <iostream>
#include <fstream>
#include "ClassC.h"
using namespace std;

namespace namespace_ClassC {
  ClassC::ClassC() {}
  int ClassC::getAuthenticationLevel(int ID) {
    int intIDFile, intLevelFile;
    ifstream RepositoryFile;

    RepositoryFile.open("IDs.txt");
    RepositoryFile>>intIDFile>>intLevelFile;
    while (RepositoryFile.good()) {
      if (intIDFile == ID)
        return intLevelFile;
    }
    RepositoryFile>>intIDFile>>intLevelFile;
  }
  return -1;
}
}

```

Figure 7 C++ Code for ClassC

The definition of ClassC is straight C++ code. However, this class uses the C++ standard template library, which is by default unmanaged. Therefore ClassC is considered to be mixed code.

There is one crucial difference between the implementation and the specification. The specification uses a pre-initialized in-memory structure to represent the repository, whereas the implementation reads from a text file. We chose this form of abstraction to highlight that our

architectural specification only has to show what a module does (return the access level associated with a specific ID) and not how it does it (use an in-memory structure, a file or even query a directory service). Another difference is that a pre-condition of the specification fails, if the ID is not in the map; whereas the C++ implementation returns -1 if the ID cannot be found in the text file.

ClassC's "getAuthenticationLevel" has the same signature in the implementation and specification as the method name and the arguments are the same. In addition, the primitive C++ types are automatically compiled as their corresponding .NET types.

We compiled ClassC as a .NET Class Library and joined it together with the rest of the unchanged AsmL specification. The application was then tested for functionality and performed as expected.

4.3 Second Incremental Development Step

As second step, we replaced ClassB with its implementation in C++. ClassB is more interesting because it calls an AsmL class, but is also called by one. Therefore it demonstrates that both are feasible.

```
#pragma once
using namespace System;

using namespace namespace_ClassC;
using namespace namespace_ClassD;

namespace namespace_ClassB {
    public __gc class ClassB {
    public:
        ClassB();
        Boolean HasAccess(String *strCard, Int32 intLevel);
    private:
        namespace_ClassC::ClassC *Store;
        namespace_ClassD::ClassD *Extractor;
    };
}
```

Figure 8 C++ Header for ClassB

In contrast to the previous header file, Figure 8 looks less familiar. The method "HasAccess" uses .NET types instead of primitive or standard template library C++ types. We decided to implement ClassB as fully managed class. Figure 9 below shows that the C++ standard template library is not being used.

```
#include "ClassB.h"

namespace namespace_ClassB {
    ClassB::ClassB() {
        Store = new namespace_ClassC::ClassC();
        Extractor = new namespace_ClassD::ClassD();
    }
    Boolean ClassB::HasAccess(String *strCard, Int32 intLevel) {
        return (Store->getAuthenticationLevel(Extractor->getID(strCard))
        >= intLevel);
    }
}
```

Figure 9 C++ Code for ClassB

We compiled this method also into a .NET Class Library and joined it together with ClassC, ClassD, ClassA and Main(). The application continued to execute properly.

4.4 Further Steps

In a next step, ClassA and Main() could be replaced by a graphical user interface. We decided not to show this step, because there is not much new to learn from.

ClassD can be left in AsmL even in the implementation, or converted to C++ code as well. If left in AsmL, then the AsmL library has to be distributed with the finished application. In addition, AsmL's state machine concept adds significant overhead to this simple class. Therefore, for performance and deployment reasons, it is advisable to change even such a trivial class as ClassD to a C++ implementation.

5. Conclusion

We have shown a feasibility study for a more flexible approach to incremental software development. In this, a formal executable specification in AsmL is obtained first. As development proceeds, individual specifications are replaced by the corresponding implementation of the module until the system is entirely composed of implementation modules. The modules can be implemented in any order, regardless of the dependencies among them.

The advantages of our approach are clear. One can demonstrate the system at any point in the development process, which also means that system testing is possible at any time. There is also no need to develop stubs and drivers for integration testing, as the specification or already implemented modules can take on these roles. Dependency issues between modules that are problematic in the traditional software development approach do not exist in our approach. In addition, developers can be allocated as they are available and not as they are needed. Also, unit, integration and system testing can be supported with AsmL's testing tool, which is described below. If that tool is used for all testing phases in a project, the development of stubs and drivers might be avoided completely.

One issue with our approach is the question of what and how to abstract. The specification must have at least one AsmL class per module, must provide the same interface as the implementation, and must use the same other modules in the same way as the implementation. In other words, its interaction behavior must be the same as the implementation's.

However, other aspects of the behavior can be abstracted. All external calls, e.g. I/O access, should be abstracted, if possible, as we did with ClassC. AsmL supports the generation of non-deterministic (random) values as built-in statements, which supports feature abstraction. These random values can be used as arguments. However, when

one module needs to take arguments it receives, process them and call another module with its results, the use of non-determinism causes problems.

Our approach is also limited to Windows platforms, as there is currently no version of the .NET framework available for any other ones. In addition, our approach requires a .NET project, and does not support pre-.NET Windows application development as done with Visual C++ 6.0 for example.

Mixed code represents a possible obstacle for AsmL's powerful testing tool. It is therefore advisable to write only managed code in VC++. The AsmL testing tool analyzes a specification, derives a finite state machine from it and generates test cases from the latter. However, the tool requires changes in variables, specifically non-local ones, to generate a finite state machine. As our example does not do this, the testing tool could not be used. Additionally, AsmL provides conformance testing, in which an AsmL specification is executed concurrently with the implementation and the implementation's results checked against the specification. For a thorough discussion of testing with AsmL, see [14] and [15].

The AsmL tools could be enhanced to support our development approach directly. Things such as better editing support within Visual Studio's IDE, the ability to have more than one AsmL file per project and the improvement of the testing tool would render our approach to incremental software development much more practical.

After having demonstrated the feasibility of this approach, one next step is to use it in a larger development project. We would also like to incorporate the use of the AsmL testing tool for unit, integration, and system testing into our approach.

6. Acknowledgement

We are grateful for the existence of the AsmL mailing list, as it provided us with very fast help on AsmL. Specifically, we thank Mike Barnett, Wolfgang Grieskamp and Nikolai Tillmann for their technical help. We also thank Mike Barnett, Colin Campbell and the anonymous reviewers for providing helpful feedback on this paper.

References

- [1] N. Fuchs, Specifications are (Preferably) Executable, *IEEE/BCS Software Engineering Journal*, September 1992.
- [2] A. van Lamsweerde, Formal Specification: A Roadmap, *Proc. Future of Software Engineering*, Limerick, Ireland, 2000, 147-159.
- [3] R. Balzer, N. Goldman & D. Wile, On the Transformational Implementation approach to programming, *Proc. of the 2nd international conference on Software engineering*, San Francisco, CA, USA, 1976, 337-344.
- [4] R. Terwilliger & R. Campbell, PLEASE: Executable Specifications for Incremental Software Development, *Journal of Systems and Software* 10, 1989, 97-112.
- [5] D. Hutter and A. Schairer, Towards an Evolutionary Formal Software Development using Casl, in C. Choppy and D. Bert, editors, *Proc. Recent Trends in Algebraic Development Techniques, 14th International Workshop (WADT 99)*, 73-88, 1999.
- [6] B. Meyer, *Object-oriented software construction, 2nd ed.* (Upper Saddle River, NJ, Prentice Hall PTR, 1997).
- [7] K. Arnout & R. Simon, The .NET Contract Wizard: adding Design by Contract to languages other than Eiffel, *Proc. TOOLS 39, 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, 2001.
- [8] C. Mingins & C. Chan, Building trust in third-party components using component wrappers in the .NET frameworks, *Proceedings of the Fortieth International Conference on Tools Pacific*, Sydney, Australia, 2002.
- [9] C. Hofmeister, R. Nord & D. Soni, *Applied software architecture* (Upper Saddle River, NJ, Addison-Wesley, 2000).
- [10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond* (Upper Saddle River, NJ, Addison-Wesley, 2002).
- [11] R. Grimes, *Programming with Managed Extensions for Microsoft Visual C++ .NET – Version 2003* (Redmond, WA, Microsoft Press, 2003).
- [12] J. Hamilton, Language integration in the common language runtime, *ACM SIGPLAN Notices* 38(2), 2003, 19-28.
- [13] M. Barnett & W. Schulte, Spying on Components: A Runtime Verification Technique, *Proc. Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001*, 7-13.
- [14] M. Barnett, W. Grieskamp, Y. Gurevich, W. Schulte, N. Tillmann & M. Veanes, Scenario-oriented Modeling in AsmL and its Instrumentation for Testing, *Proc. SCESM 2003, 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, Portland, OR, May 2003.
- [15] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann & M. Veanes, Towards a Tool Environment for Model-Based Testing with AsmL, *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, Montréal, Québec, Canada, October 2003.