Specification Reuse using Data Refinement in Dafny

M. Asif Saleem

Dissertation 2013

Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department : Dr Adam Winstanley

Supervisor : Dr. Rosemary Monahan

June 2013

European Commission

ERASMUS MUNDUS

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:____________________________ Date:____________________________

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Rosemary Monahan for her keen attention, guidance and encouragement. I am also thankful for her creative ideas, suggestions and moral support over the duration of this work. I would also like to extend my gratitude to all of my lecturers in the university, for their knowledge, skills and support they imparted to me.

Abstract

Data refinement is a technique for transforming system specifications into system implementation that differs in data types. It gives us the freedom to write specifications in a way that is independent of its implementation; moreover we can generate multiple implementations without changing the system specifications, the client does not have to worry about the underlying implementation. Abstraction Invariant is used to relate the high level abstract specification to its concrete implementation.

Dafny is a research language developed by Microsoft. Its main focus is data refinement. The language provides the rich mathematical properties such as sequences, sets and multi-set, along with functions, predicates, methods and user defined data types. In Dafny the Abstraction Invariant is in the form of a function, which is added as a pre and post conditions to all of methods and functions. Given this function one can verify that the code is providing the implementation that satisfies its specifications even when the specification is defined in term of one data structure and the code is implemented in term of another data structure. Dafny works with Boogie which is a static program verifier and the SMT solver Z3. These are the main underlying technologies for verification: Dafny code is translated in to Boogie from which the verification conditions are generated for Z3 in order to verify the program.

In this research the programmer over head is identified when replacing one implementation to another in terms of underlying data structure change while preserving the client specification. The motivation behind this work is to assist programmers to come up with a quick solution in situations such as "slow system performance" with new system implementation. Moreover, a semi automatic tool is developed for transforming one implementation to another without changing the client specifications. The result is the generation of a semi verified program whose implementation is in terms of a data structure other than that used in the specifications. The verification can be fully automatic through the provision of implementation details from the user.

List of Figures:

Table of Contents

This Page is left intentionally

1 :-: Introduction

Formal methods [1] are playing a very important role in program verification. These methods provide techniques where the computer can tell if their own programs are correct and meet its specifications. Formal methods are equipped with strong mathematical tools such as (logic and calculi) from where various proofs can be generated; which can be used for the verification of programs. By incorporating these methods into the system development Lifecycle, we can ensure the program correctness and can guarantee that the implementation meets its specification. Despite the difficulty in automating proofs, formal methods have been used in the industry for decades. These industries include but are not limited to aerospace, transport, banking, telecommunication and satellites [1].

Formal methods can be applied to any phase of software specially design, development, verification and writing client specifications. There are many languages and tools for system development which use formal methods to aid the development; some of them include VDM, ADT's, Z, Logic, OCL, JML, Spec# and Dafny. In recent years with the advancement in specification based languages such as Spec# and interactive verifiers such as SMT solvers [2], B [3] and SparkAda [4] small to medium size program verification is possible. The verification of safety critical applications is the main target of the industry such as those that build autopilot systems and Satellite platforms. Mostly specification based languages build on top of Design by Contract principles as described by Meyer [5].

Design by Contract illustrates the principle of client and supplier relationship where the client guarantees that the precondition to be met before executing the system and the supplier in return guarantees that if the precondition was met then the system terminates in a state which satisfies its postcondtion. Pre and post conditions are defined as before and after execution of the program. Modern design by contract languages such as Spec# and Dafny allows its users to write programs that can be verified by the using an underlying SMT solver, making program verification process very interactive as the verifier is constantly running in the background and prompting the users after every written line about the correctness of the program. Boogie [6] a static program verifier is used as an intermediate layer between the specification language and SMT solver. The program code is first translated to Boogie code from where the verification conditions that needs to be verified by the SMT solver are generated.

Dafny (see appendix B) is a specification based language which emphasizes particularly on data refinement. Data refinement is a technique by using which we can write specification without worrying about the implementation details. Refinement is achieved by applying an abstraction invariant that is used to relate specification data (also referred to as abstract) with an implementation data (also referred to as concrete). High level specification languages such as Dafny gives us the freedom to refine data without worrying about the underlying refinement calculus by using high level mathematical constructs such as sets, sequences and multi-sets.

1.1 The Problem Description

Software requirement specification is a technique where a software engineer gathers requirements about the system to develop and then write down it down in an informal way. Requirements may include both functional and non-functional. UML is traditionally used for gathering requirements. By using the theory of formal methods one can write functional requirements in term of logical specifications which can then be refined to produce the final system which is verified and correct.

One of the major issues in software systems is the 'performance degradation' and users of the systems are getting delayed in performing their day to day tasks. Fixing such issues need considerable concentration and time. In many cases software engineers need to think about overall system design and its underlying data structure. For instance if data structure A is being used in the implementation and we need to replace it with a data structure B in order to improve the system performance, then all the system objects who are using data structure A need to be changed to use the data structure B in order to work with new data structure. This means whole design will be changed and the software engineer will need to work further with design team to make a new design from the specification. This will cost the company resources and time. In other situations where we have specifications and design of the system but the underlying tool or language in which system is supposed to be implemented does not support the data structure which is recommended in the system design. For instance data structure A is recommended but due to lack of support for A data structure in the language or tool we want to generate implementation with B data structure which is supported by the tool or language. In this project we focus especially on the correctness of software where the data structure has been modified. We provide the proof of concept tool which translate one implementation into another while preserving the client specification resulting a semi verified program. Moreover programmer overhead will be provided for writing a new implementation manually.

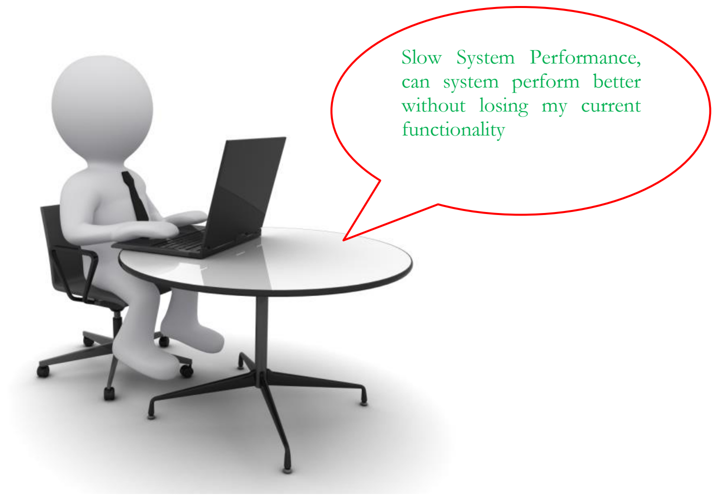

Slow System Performance, can system perform better without losing my current functionality

Figure 1.1: Problem Statement

1.2 Motivation

The motivation behind this work is to reuse client specifications by using data structure based refinement so that implementation can be written independently from its specifications.

1.3 The Goals of this research

The First goal of this research is to provide a proof of concept tool which generates another implementation that differs in data structure given that the first implementation and an abstraction function. The second goal is to provide the analysis on programmer overhead while writing new implementations. The third goal is to propose a generic framework based on data structure refinement. To achieve these goals we analyze the languages that support data refinement and its relation with underlying verifiers. We measure the overhead as a side product for manual writing of implementation by defining a metric. For tool construction two semis verified programs will be taken with the same specification that differs in underlying data structures. The current implementation and an abstraction invariant that relates current specification and new implementation will be taken as an input for generating new implementation.

1.4 Contribution of this thesis

This thesis will contribute by improving programmer overhead for writing different implementations without changing the system specifications. The new implementation will differ in the underlying data structure with provision of semi static verification[1].

This research will provide the starting point for analyzing the automatic generation of implementations for same client specifications that differs in the data structure. It is achieved by using the idea of data refinement where one specification can be transformed into different implementations. Moreover a proof of concept tool will be developed for expressing the idea which will take a program with one implementation and abstraction invariant for new implementation and generates the new implementation. The new implementation will be semi verified program from where programmers can begin to code in order to satisfy the full verification.

1.5 Thesis overview

In this chapter we discussed the motivation of our project and research goals. We have described the problem statement which we want to solve and the context of our work. In the second chapter we present an overview of related work and discuss about the general concept of abstraction and abstraction in specification languages. We analyze the support for writing specifications and verifying the code and review the support for refinement in different languages with code generation for Event-B models. We analyze whether the generated code supports the change in data structure and if it supports than whether the code required changing in specifications.

In chapter 3 we present the design for Ideal generic framework and provide its class diagram. In following chapter we implement our solution reality that what we have achieved so far with the help of a case study in chapter 4. In chapter 5 we evaluate our work with the help of tables and graphs. We provide the critical analysis of our work along with a comparison of our results with the related work. Finally we summarize our findings in chapter 6 and gives future work in this area.

[1] Checking program correctness without executing it

2 :-: Related Work

In this chapter we discuss about the related work to our problem. We discuss about the data abstraction and refinement in general and review them in different languages such as Dafny and Spec#. We assume that readers have some familiarity with the syntax of Dafny (see appendix B) and Spec# but we will assist them by providing comments for syntax highlighting and their meanings. We analyze the support for writing specifications and verifying the code. We review the two class approach to data refinement and assess the verification of this approach with possible verification issues. We discuss about Event-B [7] refinement and code generation where we discuss the verification of the generated code and whether there is facility for generating different implementations that differs in data structures without changing higher level specifications.

2.1 Problem Context

The literature has shown that not much work has done in the area of data structure based refinement. A step towards this goal was achieved in two class approach where the specification and implementation can be maintained in two different classes separating the client view from the supplier[2] but this approach is achieving specification reuse by defining new subclasses that will inherit from the same abstract super class. By using this approach one can plug out one implementation and plug in another while preserving the specifications unchanged. Our focus is on Dafny because it follows one class approach to data refinement which is relatively simple as compares to the two class approach and provides updateable ghost[3] variables as abstract data to be used in the implementation. We will benefit from its native support of data refinement for building our proof of concept tool in order to achieve the first step towards data structure based data refinement.

Data refinement is a key concept in program verification. It gives us functional and executable program that meets its specifications. Monahan [8] has proposed the two class approach in Spec# [9] for data refinement. The idea behind this approach is to separate the specification completely from the implementation by using the abstract specification class with a subclass which inherit from the specified abstract class for implementation. The implementation class overrides all abstract methods from the abstract class. This approach is not automatic and programmer should have the knowledge and expertise in Spec# with experience with the underlying verifier in order to write another implementation. Another approach to data refinement is in Event-B which is basically a modeling tool that models the system using mathematical properties. The tool has support for data abstraction and refinement from Event-B machines [10] one can generate the code for multiple programming languages such as C, C++ and Java. Code generation for Dafny is also supported from proof obligations in Event-B [11]. We analyze both two class approach and Event-B code generation with the context of our problem in the below sections.

2.2 Data Abstraction

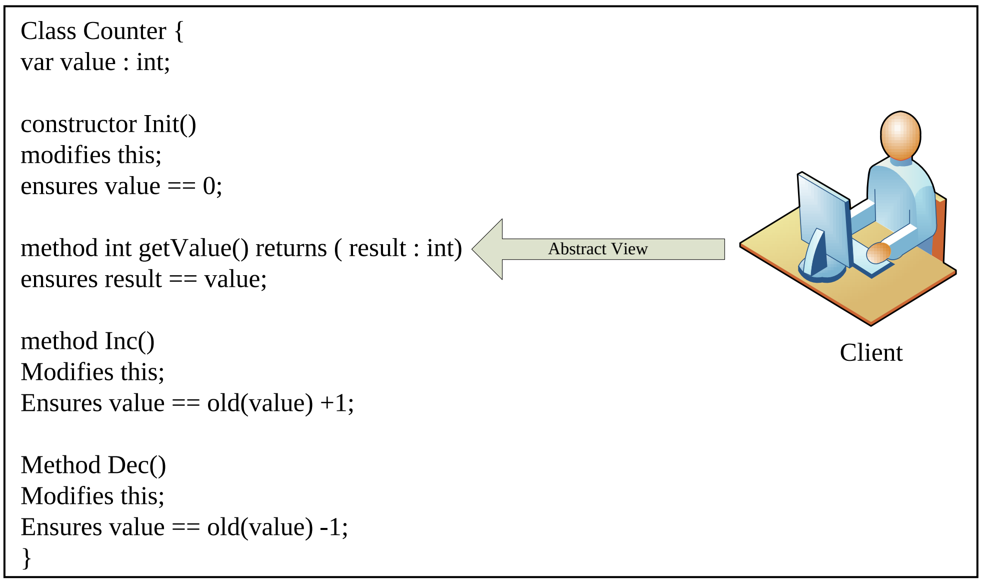
Data abstraction is one of the object oriented principles also refer to as information hiding where actual implementation is hidden from the client. The client can only see the abstract view of the system as shown in **Figure 2.1**. For example if the client wants a mathematical behavior of a sequence than the supplier of the sequence can supply the sequence behavior using underlying data

[2] We are referring implementer as supplier

[3] Specification only variables used in JML [26] and Dafny

structures such as arrays or link-lists. The client is only concerned about the mathematical behavior of a sequence where actual implementation is hidden from a client and it has only abstract view of the system. In this sequence example we are referring sequence as abstract data and underlying data structure as concrete data.

Data abstraction provides clear separation between client and supplier views. The client should not have direct access to the fields of a class because if implementation will change in the future than the abstract view will be changed as well. This situation becomes even worse if an implementer removes those fields of the class to which client have direct access than the abstract view of the system will completely vanish. Specification languages such as Spec# and Dafny provide data abstraction by using ghost and model variables. Dafny uses ghost variables whereas spec# uses model variables. Dafny has an advantage over model variable that it does not allow automatic updating of ghost variables whereas Spec# does for model variables.



```
Class Counter {
var value : int;

constructor Init()
modifies this;
ensures value == 0;

method int getValue() returns ( result : int)
ensures result == value;

method Inc()
Modifies this;
Ensures value == old(value) +1;

Method Dec()
Modifies this;
Ensures value == old(value) -1;
}
```

Figure 2.1: Client abstract view of specifications (Dafny Syntax)

Data abstraction focuses on changing abstract data to concrete data where the data types are same both in the specification and in the implementation, for example integer ghost and non-ghost variables. **Figure 2.1** shows the specification of a counter class in which it increment and decrement the value upon calling increment and decrement methods. The 'value' is a simple way to express the class operations to the client. We will take the same specification of the counter example to model abstraction with model and ghost variables.

2.2.1 Model fields

Model fields are specification only variables that are used to specify the behavior of a class. We cannot directly assign the values to these variables instead they are like functions of concrete fields and whenever concrete data is updated; their corresponding model fields are automatically updated. In program verification with model fields the abstraction relation between abstract and concrete data is also referred to as 'representation'. A model field provides an abstract view of a system where implementation remains private from the client. From an implementation point of view these fields are easy to implement because of abstract data that is automatically updated and implementer don't have to worry about the correctness of the program with respect to specifications.

A model field as compared to their counterpart's ghost fields differs in value assignment. From a client point of view they are same because both are providing data abstraction. Model fields have some drawbacks as compare to ghost fields as they are automatically updated. As model fields are the functions of concrete fields immediate updates of the model fields can cause modularity problems. Problem: 'How can a method specification name all variables that a method will modify without revealing the implementation details' [8]. The solution to this problem was proposed in a verification methodology [12] by introducing restricted updates to special statements in the language. The Methodology introduced 'satisfies' as a constraint for applying to all model fields with boolean expressions of the language. The idea behind is rather than updating model fields immediately update it whenever an object invariant holds. This ensures the correctness of abstraction function. On the other hand Spec# somehow not fully adhering the data abstraction principles because the absence of access modifiers [8]. This means that implementation details can be seen by the client by consulting 'satisfies' clause. Below is an example of a counter which is using model fields.

```
class spec_counter{
model int value {
satisfies value == incs – decs && value == getValue();
}

protected int incs;
protected int decs;

public spec_counter()
ensures value == incs-decs; {
decs = 0;    incs = 0;  }

[Pure] public int getValue()
ensures result == value
{
  return incs-decs;   }
```

```
public void Inc()
modifies incs;
ensures value = old(value) + 1;
{           this.incs = incs + 1;    }

public void Dec()
modifies decs;
ensures value == old(value) -1;
{
       this.decs = decs + 1;    }
}
```

Figure 2.2: Spec# counter class

'Value' in this example is a model field of type integer which has concrete data of two variables 'incs' and 'decs' which are representing the increment and decrement values. Spec# allows both abstract and concrete data to be in methods specifications that may reveal implementation details as we can see in the above example. Spec# provides 'Pure' methods who do not have any side effects and they can be used in the specifications. The "old" clause represents the value before and after method execution.

The client only has an abstract view of the system which is a 'value' variable in **Figure 2.2**. Underlying two concrete integers are connected to abstract integer value and whenever these two concrete values updates corresponding abstract value will also update thus achieving data abstraction. Sometimes we need data abstraction but we cannot use the same data type or constructs which are in specification because we cannot execute the specifications. Data refinement gives us the freedom to change data types in implementation about which we will talk later in this chapter.

2.2.2 Ghost Fields

Ghost variables are also specification only variables same as model variables but they give us freedom to update them manually thus avoiding problems associated with model variables. Dafny achieves data abstraction with the help of ghost variables. We take the same example of counter and demonstrate the data abstraction using ghost variables.

```
class dafny_counter{
ghost var value: int;
var incs : int;
var decs : int;
```

```
function valid():bool
reads this;
{ value == incs – decs;
}
```

```
constructor Init()
modifies this;
ensures valid();
{
incs, decs, value := 0, 0, 0;   }

public void Dec()
requires valid();
modifies this;
ensures valid();
ensures value == old(value) -1;
{
decs := decs + 1;
value := value -1 ;  }
```

```
public int getValue() returns (result : int)
requires valid();
ensures result == value;
{
result := incs – decs;   }

public void Inc()
requires valid();
modifies this;
ensures valid();
ensures value == old(value)+1;
{
incs := incs +1;
value := value +1;   }
```

Figure 2.3: Client abstract view of specifications (Dafny Syntax)

Above example is separating the client view from implementation by declaring 'value' as a ghost variable and providing increment and decrement functionality with the help of two concrete integers where abstract and concrete variables both have the same data types thus achieving data abstraction. Concept of validity function is same as a model block in Spec# which is defining the abstraction relation. In Dafny this relation needs to be validated for every method as pre and post condition for consistency of the system.

2.2.3 Refinement

We usually model the client specifications by using mathematical properties such as set theory and unbounded sequences. The problem with writing specifications using mathematical properties is that computers cannot directly execute these specifications. These specifications should be converted into code for execution. The process of converting high level abstract specifications into executable code is called refinement. Below the diagram is representing a refinement from high level specifications into executable program.

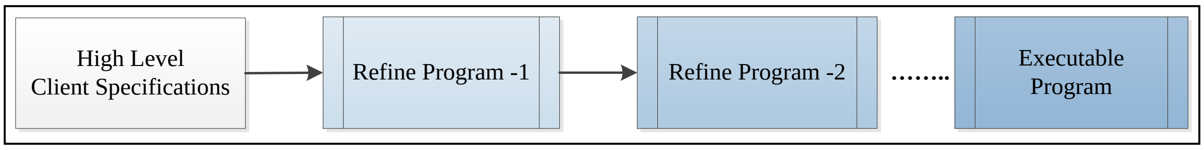


Figure 2.4: Program Refinement

2.2.4 Data Refinement

Data refinement is a subset of refinement where high level specifications are transformed into implementation which differs in data types. Formally we can define data refinement as a 'process of converting abstract specification into concrete implementation using data type that are different from specification e.g. specification in term of set and implementation in term of arrays'.

Specifications are typically written using higher level mathematical properties such as sets, sequences and multi-sets which are all non executable constructs. The need for data refinement exists for multiple reasons. For instance if we have specifications written in term of sets and let's assume that the specification can execute by the computer and we want to extract an element on a particular location from the set. We cannot perform this operation because set does not have such property instead we can refine this set to some concrete implementation such as arrays or link list to perform this operation. Modern languages such as Dafny provide the facility to use abstract data within the concrete implementation solely for verification purpose.

Object oriented specification languages such as Dafny and Spec# provides one class-approach to data refinement where client specification and implementation remains in one class. This approach has disadvantages. First the implementation is exposed to the client because the client has access to abstract data. Second disadvantage is that it is difficult to distinguish between abstract and concrete data within one class because for understanding the logic of the program you have to review all the code. Another approach is called 'two class approach' in which specifications and implementations are kept in separate abstract and concrete classes where the implementation class extends the abstract class and override all its abstract methods. This approach provides clear separation between client and supplier and these two classes are related through abstraction invariant. Abstraction invariant which is a key component in refinement is placed in the abstract class. An abstract class consists of three components, abstract data, abstract data invariant, and constructor where methods specifications are written in term of abstract data. Concrete class has same three components but in term of concrete data. Monahan [8] has proposed this approach in Spec# for achieving modular data refinement. This model represents data refinement through inheritance and preserves the property that the implementation meets its specification. This approach has its own drawbacks about which we will talk later in this chapter. Below is an example of data refinement using Z [13] which uses a simple birthday book system that records the name and birthdays of different people.

Specifications: A system which can record the birthdays of different peoples and can issue the remainder to people who have birthdays on the same date. Moreover system should able to find birth date for any particular person.

Above specifications can be written by using mathematical sets. If we analyze the specifications we can easily judge that we need two sets, one for name and other for corresponding birth dates.

¶NAME : = set of all names {Bob, John, Alice, Agatha}
¶DATE : = set of all dates {'12-02-70', '02-12-70', '04-07-70', '04-07-70'}

Where ¶ is representing a set. We require some sort of function or relationship so that we can relate each name to its corresponding birth date i.e. Bob has a birthday on 12-02-70

ʃ birthday(NAME) = {Bob -> 12-02-70,
John -> 02-12-70,
Alice -> 04-07-70,
John -> 04-07-70}

Let's introduce one more set 'present' where present contains all the names currently in the systems. Initially system look likes below.

BirthdayBook
¶present : ¶NAME
¶birthday : ¶ ʃ(NAME) ⟶DATE
Invariant: present == domain of birthday

Now we can use above information to define different operations.

Δ AddBirthdayday
⟹*Input to system*
name? : Name
date? : DATE
⟹*processing*
name? ∉ present
birthday := birthday U {name? ⟶ date?}
present : = present U {name?}

Where ? symbol is representing input variable and Δ is representing a function that can change the system state. Whenever system changes its state invariant should hold before and after the state. From the above method we can see that name is added to 'present' which is satisfying the invariant.

₪ FindBirthday
⟹*input to system*
name? : NAME
date! : DATE
⟹*processing*
name? ϵ present
!date = birthday (name?)

Where ! is representing the output and ₪ indicating that the system state will not change as the result of operation.

₪ RemindBirthday
⟹*Input to system*
today_date? : DATE
nameslist! : ¶NAME
⟹*processing*
nameslist! = { n : present | birthday(n) == today_date? }

The 'nameslist' is containing all the names who have birthday days on "today_date". This set can be interpreted as 'forall n in present such that birthday(n) == today_date'. Now we design our program based on the above specifications. The idea behind this design is to convert abstract data from specifications to concrete data structure so that the computer can execute this system. Let's choose arrays to represent the birthday book such as
:_: names, NAME : array[..] : array of names
:_: dates, DATE : array[..] : array of dates
Initially the array is infinite but in real implementation we have to specify the size of the array but the behavior remains same. For example names[i] where 'i' is the index of the element which is equivalent to name(i) in specifications and names[i] := "elem" is equivalent to
:_: name := name U {i->elem}
Where 'U' is representing a union. The concrete function of initial birthday book looks like below

BirthdayBook
names [] : array of NAME
dates [] : array of DATE
size: size of the array
condition: ∀ i,j : 1 .. size • i ≠ j ⟹names(i) ≠ names(j)

Birthday book now contains two arrays with their respective size and a precondition

Abstraction Invariant (Abs)
present == { i: 1 .. size • name(i) }
∀ i : 1 .. size • birthday(names(i)) == dates(i)

Abstraction invariant shows that the set 'present' is containing all the names up to the size of the array and every name has corresponding birth dates.

AddBirthday
⟹*Input to system*
name? : Name
date? : DATE
⟹*processing*
∀ i : 1 .. size • => names? ≠ names(i)
size : = size +1
names : = names ⊕ {size ⟼ name?}
dates : = dates ⊕ {size ⟼ date?}

This ‘AddBirthday’ function has the same input and outputs to ‘AddBirthday’ function in the specifications except it is operating on concrete data. The results of the operations are same in both cases. In concrete implementation every name and date have a unique index in order to get the ‘date’ against any ‘name’. It is same as we have added $\{name \rightarrow date\}$ in the birthday book in specifications. We can translate above function easily to our programming language as below

```
method Add(name: NAME, date: DATE)
{
size : = size +1 ;
names[size] : = name;
dates[size] : = date; }
```

FindBirthday

$\Rightarrow$ *input to system*

name? : NAME

date! : DATE

$\Rightarrow$ *processing*

$\exists\, i : 1 .. size \bullet name? = names(i) \wedge date! = dates(i)$

We can interpret ‘FindBirthday’ function as ‘there exists some ‘i’ such that names(i) is equal to input name and date! Is containing the corresponding date value of the same index. This function can easily be translated to programming language as below

```
method FindBirthday(name : NAME) returns (date: DATE)
{   var i : int;
i : = 1;
while (i <= names.Length){
If (names[i] != name)
{i : = i+1; }
else { date : = dates[i]; break; } } }
```

‘RemindBirthday’ is returning the set of all names whose birth dates are on some particular date.

RemindBirthday

today_date? : DATE

namelist [] ! : NAME

ncount! : int

$\{ i : I .. ncount! \bullet namelist!(i) \} = \{ j : 1 .. size \mid dates(j) == today\_date? \bullet names(j) \}$

RemindBirthday’ can easily be translated to programming language as below

```
method RemindBirthday(today : DATE) returns (namlist[] : NAME, ncount: int)
{
var j : int;
while ( j < size ) {
j : = j +1
if (dates[j] == today)
{
ncount : = ncount +1;
namelist[ncount] : = names[j];   }}}
```

2.3 Abstraction in Software Engineering

Abstraction in software engineering refers to very high level requirements of a software system that cannot be directly modeled and coded. These requirements are also referred to as abstract specifications of a system. The process of decomposing high level requirements into more understandable is called refinement. In software engineering practices refinement is mandatory for designing robust software applications and also proving that the end user product meets its specification. In addition it may reduce the time, effort and risk associated with the software system. Design patterns [14] are used to model the abstraction in software development process.

2.4 Specifications and Refinement Support

There are many tools and languages which provide support for writing specifications from which some of them provide rich mathematical properties and many of them have limited support. Below is the description of some tools and languages

-:-<u>UML and OCL</u>

UML is a standard modeling language providing a rich set of tools for modeling the system such as use case, class, and component and deployment diagrams. Object Constraint Language is used to define constraints over the UML model such as invariant, pre and post conditions. OCL [15] has limitation to write specifications over the models only such as for class diagrams.

-:-<u>Specification Modeling</u>

<u>Z Method</u>

Z [13] is a powerful modeling language that supports writing of specifications by providing a richer mathematical tool-kit such as set, multi-sets, relation, functions, number and finiteness.

<u>B Method and Event-B</u>

B [3] is powerfuler modeling language that supports refinement. It's capture the system specifications as an abstract design model and refine it gradually with new requirements until the construction of concrete model. B method provides greater support for rich mathematical properties such as Predicates, Set theory and logical structures. Event-B is a modeling tool based on the B language. This tool provides interactive user interface for modeling the system and available under the name of Rodin [16].

<u>Circus</u>

Circus [17] is a specification language that was designed specifically to support refinemenofor concurrent programs. Circus was developed by combining process algebra CSP [18], Z method anthe refinementnt calculus.

<u>Temporal Logic</u>

Temporal logic [19] has an important application in formal verification. Temporal logic is used to write specifications for concurrent programs in which we can define order of things that may happen with respect to time.

-:-<u>Design by Contract Languages</u>

<u>Spec#</u>

Spec# is a DbC[4] language supporting general specification constructs such as requires, ensures and class invariants but it does not support mathematical specification constructs such as sets, multi-sets and sequences hence the support for writing the specification is very limited.

<u>Dafny</u>

Dafny (see appendix B) is a DbC language supporting general and mathematical specification constructs such as requires, ensures, sets, multi-sets and sequences. Support for writing specifications in Dafny is much higher than Spec#.

[4] Design by Contract

Eiffel

Eiffel [20] is a DbC language supporting general specification constructs such as requires, ensures and class invariants. The language supports rich mathematical constructs such as sets and multi-sets hence providing good support for writing specifications.

-:-Theorem Provers

With the advancements in verification technology program verification is becoming interactive with the development of automatic theorem Provers such as SMT [2] and extended static verifiers [21].

2.5 Data Refinement Support

Data refinement is a special case of refinement in which data types require changes from what are in the specifications. Spec# and Dafny both supports data refinement. Below are the details

2.5.1 Spec#

Spec# is a powerful language for data refinement. The only hurdle for using this language is the absence of high level mathematical properties such as set theory and unbounded sequences. Spec# provides refinement using one and two class approaches. Spec# classes contain the following information

- Abstract data, object invariant, method specifications and constructor specified in term of abstract data
- Concrete data, object invariant, methods and constructor written in term of concrete data and abstraction invariant relating abstract and concrete data

Below is a classic example of data refinement which is converting a high level sequence to an array. Sets, sequences and multi-sets are not supported in Spec# hence we are simulating the sequence with two integers where one integer is representing the sum of a sequence and another represents the number of elements in the sequence. This limitation is overcome by Dafny by providing all these specification constructs such as sequence, set and multi-sets hence focusing more on data refinement support.

```
public class Seq {
[SpecPublic] private int seq_sum;
[SpecPublic] private int seq_count;
[Rep] private int []! array ;
private int array_count ;

invariant 0 <= seq_count ;
invariant 0 <= array_count && array_count <=
array.Length ;
invariant seq_count <= array.Length && seq_sum ==
this.array_sum();

Seq ()
ensures seq_count == 0 && seq_sum == 0;
ensures seq_count == this.array_count; {
this.array_count = 0; array = new int [100];  }

[Pure] public bool isEmpty ()
ensures result == ( seq_count == 0); {
if (array_count == 0) return true ;
else return false ;  }
```

```
void add(int elem)
requires 0 <= elem;
modifies this.*;
ensures seq_sum == old( seq_sum ) + x && seq_count
== old( seq_count ) + 1;
{
expose (this){
array [this.array_count ] = elem;
array_count ++;   }  }

[Pure] public int array_sum ()
ensures result == sum{int i in (0 : array.Length ); array
[i]};
{
int s = 0;
for(int j = 0; j < array.Length ; j ++)
invariant j <= array.Length ;
invariant s == sum{int i in (0 : j); array [i]};
{s = s+ array [j];  }
return s;
}
```

Figure 2.5 Data refinement in Spec#

[SpecPublic] annotation is indicating that the fields are specification only and simulating sequence as a pair of two integers. The ‘array’ and ‘array_count’ are two concrete fields containing elements and the number of elements in the array. ‘Rep’ annotation is for representation which is showing the ownership hierarchy that the array object is owned by the Sequence class. ‘Seq’ constructor is initializing concrete fields and initially declaring the array with hundred elements. ‘isEmpty’ is a pure method because pure methods have no side effects on the behavior of the class so they can be used in the specifications. The ‘isEmpty’ method is returning a Boolean value based on the current state of the array. The state can be empty or it contains some elements.

Three object invariants were used in this example. First invariant for abstract data, second is for concrete and third for relating abstract and concrete data. ‘Add’ method adds an array element in exposing block which tells the verifier about skipping the object invariant constrain check within the expose block. It’s the user's responsibility to check whether object invariant is established or not after the execution of exposing block. Pure method “array_sum” is calculating sum by comparing it with the sum of a sequence. Model fields and “get” property of Spec# can be used as well with the help of object oriented inheritance principle for data refinement about which we will talk later in this chapter.

2.5.2 Data Refinement in Dafny

Set theory is a strong mathematical concept that can be used for writing specifications. We can express natural language in the form of a set such as a set of traffic lights or set of names. Spec# does not support set theory and other mathematical constructs. For writing specifications we have to define simulations such as we did in the last section. In this case refinement will be to convert simulated abstract variables to some underlying data structure such as arrays or linked lists. While on the other hand languages such as Dafny which have support for set theory and other mathematical constructs. In this case data refinement will be the conversion from the high level set or sequence specifications to arrays, link-lists or any other suitable data structure. We will use Dafny as a source language upon which we will build our tool.

The advancement of research in languages that supports specifications to be written as a part of the program, multiple steps can be omitted in refinement process often called direct refinement. For example Dafny support the concept of “ghost variables” by using these variables one can write specifications in term of mathematical sets, multi-sets or sequences and can implement the code using arrays, sequences, link-list or any data structure supported by the language. In addition we can use sequence for both specification and for implementation what we need to do is to define an abstraction invariant which relates the abstract data (specifications) to concrete data (implementation). For instance comparing the length of a sequence and array such that ‘sequence.length == array.Length’ and each element of a sequence with each element in array such that set[i] == array[i] ‘where ‘i’ iterate from 0 to length-1’ can be an abstraction invariant. In some cases for writing abstraction invariant we may need to keep track about operational differences between mathematical specification constructs and underlying implementation. For example set might use union operation to add an element and sequence uses concatenation for adding an element. In this case we require defining a mapping function between the set and array in order to validate the abstraction invariant. Below is an example of data refinement

```
class SumFind
{
ghost var abst_array: seq<int>;

function Valid(conc_array: array<int>) : bool
 reads this, conc_array;
 requires conc_array != null && conc_array.Length >=0 ;
{
conc_array.Length == |abst_array| &&
(forall i :: 0 <= i && i < |abst_array| ==>
```

```
conc_array[i] == abst_array[i] )
}

method Sum(conc_array: array<int>, arlength: int)
returns (array_sum: int)
requires 0 <= arlength && conc_array != null &&
conc_array.Length == arlength;
requires Valid(conc_array);
ensures Valid(conc_array);
{
```

```
array_sum : = 0;
var i : = 0;
while (i < arlength)
invariant i <= arlength;
decreases arlength-i;  {
array_sum : = array_sum + conc_array[i];
i : = i + 1;
} }

method Find(conc_array: array<int>, elem: int,
arlength: int) returns (ghost index: int , res_elem :  int)
requires arlength >=0 && conc_array != null &&
conc_array.Length == arlength ;
requires Valid(conc_array);
ensures Valid(conc_array); {
```

```
index : = 0;
var i : = 0;
while (i < arlength)
invariant i <= arlength;
decreases conc_array.Length-i;
{
if (conc_array[i] == elem) {
res_elem : = conc_array[i];
index : = index +1 ;
return;
} else {
i : = i +1;
index : = index + 1; } }
}
}
```

Figure 2.6: Data refinement in Dafny

The class 'SumFind' is declaring an abstract sequence using ghost variables. Valid function is the abstraction invariant which is relating abstract and concrete data which is an array in our example. This function is always returning boolean and used as a pre and post condition to methods. Method 'Sum' is calculating the sum of the array and checking the abstraction invariant as its pre and post condition. Method 'Find' is searching for an element in the array and returning both index and the element. The invariant is the 'loop invariant' which remains true trough out all loop iterations and decreases clause ensuring that the loop will terminate. In this example abstract data is in term of a sequence where concrete implementation is using an array. We will provide an analysis for writing multiple implementations for same specifications using data refinement later in this chapter.

2.6 Two Class approach to data refinement

The basic idea behind this technique is to provide modular data refinement [8] where the client haa completely separate viewew of the system and implementation details are completely hidden from the client. In two class approach, specification and implementation are kept in separate classes. The abstract class where we can define abstract variable and methods and whoever class implements this abstract class will in turn provide the concrete implementation of all the abstract methods. In Spec# two class data refinement approach has proposed by Monahan [8] who is using existing Spec# and C# properties.

The approach is using inheritance and other language features as a backbone for verification. Specification class does not contain any implementation details and abstraction invariant which relate the abstract and concrete data is placed in the subclass. As specification class contains only abstract variables and abstract methods 'Additive' clause is used to allow abstract data fields to be referenced in the subclass invariant. Fields are also kept protected in order to use them in the subclass. Pure methods are also the part of the specification class for using within the specifications. These two classes will also adhere to the inheritance principles in which post conditions can be strengthened and preconditions may weaken where frame conditions shouldn't be modified. Abstract classes require their fields to be "SpecPublic" and "Additive" for fulfilling this principle.

We take the same Sequence example as stated in **Figure 2.5** and demonstrate this approach on it. Below is the abstract class

```
public abstract class SeqAbstract {
[SpecPublic] [Additive] private int seq_sum;
[SpecPublic] [Additive] private int seq_count;

Invariant 0 <= seq_sum;
Invariant 0<= seq_count;
```

```
Seq()
ensures seq_sum ==0;
ensures seq_count ==0;

[Pure] public bool isEmpty()
ensures result == (seq_count == 0)
```

```
public void add(int elem)
requires 0 <= elem;
modifies elem, seq_sum, seq_count;
ensures seq_sum == seq_sum +1
ensures seq_count == seq_count+1
}
```

Figure 2.7: Abstract Class

The abstract class contains only specification and invariants of abstract data. These specifications and invariants will be inherited in the subclass and will be conjucted with implementation class. The problem here is the constructor of abstract class which is initializing only the abstract data fields whereas constructor in the subclass will only initialize the concrete data fields. Abstract constructors will not be inherited in subclasses and as per language property its subclass responsibility to establish the object invariant after constructor execution so we have two constructors one for abstract data and one from concrete data and verifier has no knowledge how to check object invariant to verify such program. The solution to this problem was proposed by using C# properties 'get' and 'model fields' [8]. A subclass of **Figure 2.7** is as follow

```
public class SeqConcrete : SeqAbstract {
 [Rep] private int []! array ;
private int array_count ;

invariant 0 <= array_count && array_count <=
array.Length ;
invariant seq_count <= array.Length && seq_sum ==
this.array_sum();
SeqConcrete ()
{
array [this.array_count ] = elem;
array_count ++;   }  }


[Pure] public int array_sum ()
ensures result == sum{int i in (0 : array.Length );
array [i]};
{ int s = 0;
this.array_count = 0;
array = new int [100];   }

[Pure] public bool isEmpty ()
{
if (array_count == 0) return true ;
else return false ;   }

void add(int elem)
{
expose (this){
for(int j = 0; j < array.Length ; j ++)
invariant j <= array.Length ;
invariant s == sum{int i in (0 : j); array [i]};
{
          s = s+ array [j];
}
return s;   }   }
```

Figure 2.8: Concrete Class

The **Figure 2.8** is representing the concrete implementation of the abstract class by using inheritance relationship. Defining an abstraction relation between abstract and concrete data is a major part in data refinement. As per our above example abstraction invariant is now part of the implementation class. The first drawback of this approach is the mixing of abstract and concrete variables for object invariant. It's now difficult to distinguish between these two variables in the implementation class. The abstraction relation now contains both abstract and concrete fields. The abstract class is allowed to modify his abstract fields that may create inconsistency with object invariant and verification errors. Additive is used in order to avoid the violation of object invariant. Second drawback is the constructor in Spec# where it has the freedom to violate the object invariant during his execution. The abstract class constructor is not inherited in the subclass and it has his own abstract fields to initialize where on the other hand concrete constructor have his own fields to initialize and object invariant is required to have both abstract and concrete data for establishing so in this situation verifier has no knowledge that how to establish the object invariant. The solution to this problem is to use C# properties and hence enabling Spec# for modular data refinement [8].

Spec# provides the 'accessor' methods to read and write private fields that have the same functionality like any other regular methods. 'get' is a built in accessor method in Spec# that can be used as a part of the property. This method is pure by default and can be used in the specifications. We can now write abstract data as property using 'get' method which will be overridden in the subclass and conjucted with the concrete data. Overridden version of property provides the concrete representation of abstract fields. Below is an example

```
public abstract int Count{
get:
ensures 0 <= result; }
```

Count is a property representing the abstract data with the invariant that 'Count >=0'. This property is overridden in the subclass to provide the abstraction invariant for both abstract and concrete data as below

```
private int array_count;
public override int Count{
get:
ensures result == this.array_count;
{return this.array_count; }
```

The abstract data Count is mapped to 'array_count' concrete data by the abstraction invariant "result == this.array_count". As post conditions can be strengthened in a subclass, 'array_count' property is using this principle and providing the abstraction invariant as a property post condition. By using the C# 'get' property, two class approach is now practical because the abstraction invariant issue can now be handled without breaching the program verification.
Another approach proposed by this technique for handling the abstraction invariant issue by using model variables. Rather than providing abstract data in properties, bundling it under model clauses which satisfy the given assertion. Below is an example

```
model int Count{
satisfies 0 <= result; }
```

For providing concrete implementation model fields can be overridden in the concrete class as follow

```
private int array_count;
override model int Count{
satisfies result == array_count;  }
```

In this approach abstraction invariant can directly be written in satisfying clause instead of post condition. Two class approach is a perfect start to think about data structure based refinement. This approach has provided perfect modular data refinement but it has some drawbacks that are outlined below.

2.7 Critical Analysis

One of the basic ideas behind two class approach was to separate specification from implementation which was achieved by using inheritance, Spec# and C# properties so that we can reuse the specifications. There are some limitations and issues for using this approach as outlined below

- Programmer overhead is high for writing new implementation
- The programmer must have knowledge of the existing system in order to write new implementation.
- The programmer has to write all abstraction invariants for the new data type in order to relate abstract data with concrete data.
- New abstraction invariants mean programmer has to define all properties or model fields for the new implementation.
- Need to provide a new implementation for all abstract methods.
- No support to express specifications in term of mathematical properties such as sequence, set and multi-sets
- It is very hard to reuse specifications because the lack of support for mathematical specification constructs.

- No automatic support discussed for generating new implementation
- One class approach is better to use for automatic data structure based refinement because we do not have to deal all the complexity that comes with two class approach by compromising the separation between specification and implementation.

2.8 Refinement and Code generation in Event-B

Event-B [7] is a modeling tool for modeling software systems. Event-B uses rich mathematical structure to write models and generated proof obligations[5] for these models to prove their correctness. The tool uses contexts and machines where context used to model all static properties of a system and machine models all dynamic properties. The Event-B model consists of an abstract machine or refinement of an existing machine. Event-B uses B [3] structure to write context and machines. Static properties include, sets, constants, axioms and theorems where sets are used to specify the behavior of a system, constants defines all constants needed to fulfill that behavior of the system, axioms define the properties of sets and constants where theorems includes all those theorems that can be derived from the axioms. Contexts can be extended in order to add new system requirements or system change.

The machine model dynamic properties and 'sees' the context for expressing the use of constants and sets that satisfy axioms and theorems. The machine consists of invariants, guard conditions and events. Invariant defines the safety properties of the model so that model remains in a consistent state throughout his lifetime and nothing bad will happen. Events are different states of the machine triggered upon satisfying the guard conditions. One of the main purposes of safety property is the correctness of states on occurring of different events. Proof obligations [22] are generated for ensuring the correctness of model which guarantees that the system is in a safe state for all possible occurrences of events. Machines can be refined in order to add new requirements or to handle system change requests. The idea behind is to keep refining machine as new requirements come unless it reaches the final state where all system requirements have been met. The resulting system will be correct and meets its specifications. Event-B provides refinement proofs to be validated while refining machines which preserve the correctness of the system throughout refinement levels. Idea of refinement in Event-B is slightly different as compared to specification languages that we have discussed so far. Event-B considers refinement in context of machines where machines contain dynamic properties and whenever new feature will be added in the system machine refinement is needed so that new features cannot disturb the previous properties of the system. Refinement proof helps to ensure this correctness.

Code generation is an active area of research from formal specifications. The idea is to generate code for execution by the computer that meets its specifications. **Figure 2.9** shows the plug-in architecture for code generation in Event-B.

We start from defining an abstract machine model and keep on refining unless final version is reached. Code generation is started after final refinement of the model for formally executing it into the computer system. Automatic code generation provides many benefits such as testing. We can test our generated code against the specifications which we modelled in Event-B. Many tools exist for automatic code generation from formal specifications such as Classical B [3] but our focus is on the Event-B code generation here. EB2C, EB2C++, EB2J and EB2C# tools are available as plug-in to Rodin [16] platform for code generation from Event-B model to C, C++, Java and C#. These tools work by taking context file of the target language and generate code by handling mapping of contexts, machines and events from Event-B model to target language.

[5] A proof obligation is a mathematical proving the correctness of the model

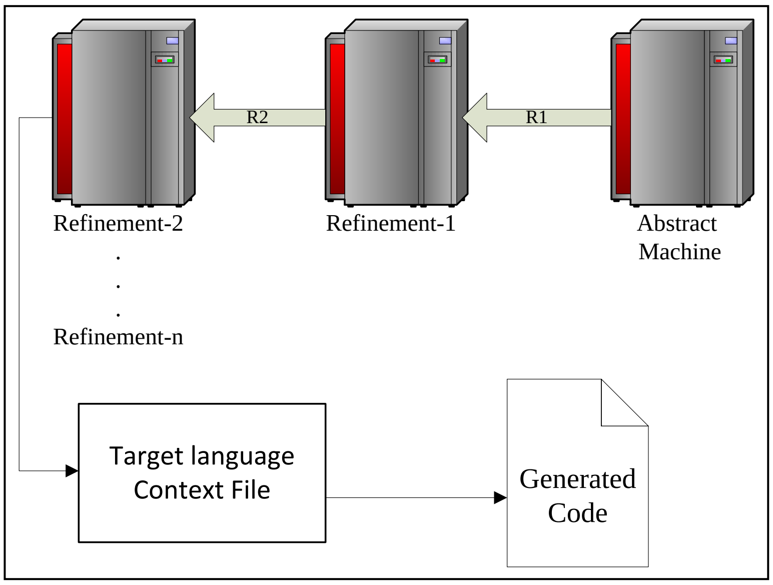


Figure 2.9: Code generation in Event-B

2.9 Critical Analysis

Code generation plug-ins provides greater flexibility for generating target language code in Event-B but these tools have some limitations and drawbacks because of the nature of modelling tools.

- ♦ No formal verification exists for generated code, as it relies on the correctness of Event-B model such as C code verification with VCC.
- ♦ Tools does not support all formal symbols of Event-B such as "IFF" (<-->) and forall (∃ and∀) quantifiers, that's mean user intervention is required to complete the code. Which may breach the correctness of the model
- ♦ Integrating user define or external code that may be required for functions the system may breach the correctness of generated code.
- ♦ Multiple implementations are not supported that differs in a data structure for the same Event-B model.
- ♦ For supporting multiple implementations separate mappings are needed for each data structure from Event-B model. For instance Event-B set model to link-list or arrays. Currently plug-ins are providing one to one mapping such as set model to target language enumerations and model functions to language functions.
- ♦ In case of slow performance of generated code the user has to write the implementation manually which may breach the code correctness and may result unreliable software because of unverified code.
- ♦ We cannot use Event-B model to generate multiple implementations that differ in the underlying data structure as there is no such plug-in support.

Although these plug-ins are very useful for executing the system but due to the lack of multiple implementations support their usage may be limited. Usually in large programs one of the user requirements is faster system performance so these tools should provide support for specification reuse based on change in the data structure.

2.10 Programmers overhead

Below table is comparing the effort of implementing and verifications of different data structure for same specifications with particular focus on Dafny.

Specifications	Data Structure	Implementation	Verification
Sequences	Sequences	Implementation is easy if we have specification in terms of sequences because it is simple to define the abstraction relation between abstract and concrete data due to the available functionality in Dafny such as getting the length of the sequence, getting value on a specific index, updating the value on the specific index and adding elements dynamically. This is one to one mapping and representing data abstraction instead of refinement.	-***Easy*** to write pre and post conditions based on the method behavior -***No*** manual loop termination logic is required because of the 'length - |seq|' functionality provided by the sequence where we can check the bounds of the sequence -***Easy*** to traverse a sequence which helps for writing complex post conditions for state modification.
Sequences	Link list	Implementation is hard as compare to sequences because we need to define all operations such as getting an element or setting an element in a link list. For writing abstraction invariant now we have to compare abstract sequence with link-list in terms of length and contents of the elements. We need to define initial static node say "head" of the link list and also there should be a relation between abstract data and link-list next node data.	-***hard*** to write pre and post conditions based on method behavior where we need to take care of current and next node. -***Loop*** termination logic is required because link list do not have any bounds. Suitable invariants and decrease clauses needed -***hard*** to traverse link-list for writing complex post conditions after state modification.
Sequences	Arrays	Implementation is easy with arrays when using as underlying data structures. One issue for writing abstraction invariant is the difference in the way the arrays behave in the language. For example we can grow sequence dynamically up to an infinite number of elements but for arrays we have to specify the size at the time of allocation. This imposes the restrictions for defining the abstraction relation between arrays and sequences. The solution is to pass arrays as method parameters to every method because arrays are treated with reference in Dafny and verification is possible.	-***Easy*** to write pre and post conditions based on method behavior -***No*** manual loop termination logic is required because of "array Length" function which is providing the bound checking of the data. -***Easy*** to traverse the array for writing complex post conditions after state modification
Sets	Link-list/Arrays	Available functionality in Dafny so far for "sets" manipulations is only to check the presence of an element. In the latest 1.6 version of Dafny new functionality for iterating over the sets is introduced. Still direct refinement from sets is not supported. We need to write a mapping function from set to any underlying data structure.	-***hard*** to write pre and post conditions which are based on method behavior. -***Loop*** termination logic is required in case of link list but not required in case of arrays -***Mapping*** function needs to be define from sets to underlying data structures for supporting data refinement. -***hard*** to traverse link-list for writing complex post conditions after state modification.

Figure 2.10: Implementation and verification comparison

2.10.1Effort Required

We are using Constructive Cost Model (COCOMO) [23] to estimate the effort and development time required for writing the code. This model is a standard for software project estimations and widely used in the industry. This model has three modes of operations, basic, intermediate and advance. Each mode is further divided into three project types. Organic projects: where teams are usually small and have good working experience and less rigid requirements. Semi-detached projects: where teams are usually medium in size and have mixed working experience with moderate rigid requirements. Embedded projects: usually have tight timing constraints and can be mixture of Organic and Semi-detached projects. We are taking the intermediate model of COCOMO and modifying its cost drivers to estimate the implementation and verification effort of a program. This estimation can be used only with specification based languages with a good mathematical toolkit for writing specifications. The table below is defining the standard values of the model

Project Type	a	b	c	d
Organic/Small	3.2	1.05	2.5	0.38

Now we define the additional attributes for calculating the verification effort on a three-point scale where 3.0 denotes the lowest value and 0.0 highest

Cost Driver	Rating		
	Low	Moderate	High
Verification attributes			
Verification Skills	1.25	1.00	0.75
Programming Language Knowledge	1.20	1.00	0.67

Now we define the type of verification rating based on the verification type. Automatic verification refers to the underlying verifier such as SMT or any similar interactive verifiers while manual verification is without using any interactive verifier.

Cost Driver	Rating
Verification Type	
Manual	1.00
Automatic	0.20

Now we define the rating for data structure according to their complexity

Cost Driver	Rating
Data Structure used	
Sequence	0.05
Arrays	0.10
Link list	0.20
Tree	0.60

Total effort required with program verification

Effort = a*(KLOC) ^b * EAF [man-months][6]
Development Time = c*(KLOC) ^d [months]
Total Days = Development Time * 30 [days]
Persons Required = Effort/Development Time

EAF (effort adjustment factor) is the product of all three cost drivers according to the scenario. **Figure 2.11** shows the effort required for implementing different data structure based on the above defined complexity. The figure is showing the effort based on number of lines of code in thousands with effort adjustment parameters as follows. High skills in verification and programming with automatic verification. The effort adjustment factor would be
EAT = (0.75) * (0.67) * (0.20) * (used data structure rating)

[6] Amount of time an average person spends on a software project in a month

Specification	Data Structures	KLOC	Implementation and Verification effort in term of DT
Sequence/Set	Sequence	500 => 0.5	12 days
Sequence/Set	Link-list	500 => 0.5	20 days
Sequence/Set	Arrays	500 => 0.5	15 days
Sequence/Set	Tree	500=> 0.5	30 days

Figure 2.11: Effort required-1

Figure 2.12 shows the effort required for implementing different data structure based on the above defined complexity. The figure is showing the effort based on number of lines of code in thousands with effort adjustment parameters as follows. Low skills in verification and programming with manual verification. The effort adjustment factor would be

EAT = (1.25) * (1.20) * (1.00) * (used data structure rating)

Specification	Data Structures	KLOC	Implementation and Verification effort in term of DT
Sequence/Set	Sequence	500 => 0.5	33 days
Sequence/Set	Link-list	500 => 0.5	56 days
Sequence/Set	Arrays	500 => 0.5	43 days
Sequence/Set	Tree	500=> 0.5	85 days

Figure 2.12: Effort required-2

2.11 Conclusion

In this chapter we discussed about the Data abstraction and refinement in general followed by data refinement in different languages and tools. We discussed the Abstraction in Software Engineering and analyze the different languages and tools that support specification writing. We proposed a metric for measuring the overhead of programmer while writing manual implementation that differs in the data structure. In the next chapter we will design our ideal solution for generic framework for data structure based refinement.

3 :-: Solution Design

In this chapter we present the solution requirements and design for the problem which we have stated in chapter 1. We review the problem statement again and suggest the solution to the problem in case of already verified program. Moreover we present the design of the generic framework for refinement based on data structure.

3.1 Problem statement

As we discussed in chapter 2 writing new implementation is cumbersome especially in large programs. Our effort metric in chapter 2 is showing the effort required for manual new implementations. In addition to that effort programmer should have the existing knowledge of the system and have a good understanding of verification technology underneath together with the time and cost of writing new implementations. The Solution to this problem can be achieved if we can generate new implementation without programmer intervention in a fully automatic way where we already have a verified program. We name our system a "Specification Reuser" because it can use existing specifications for generating new implementation. Below is the block diagram

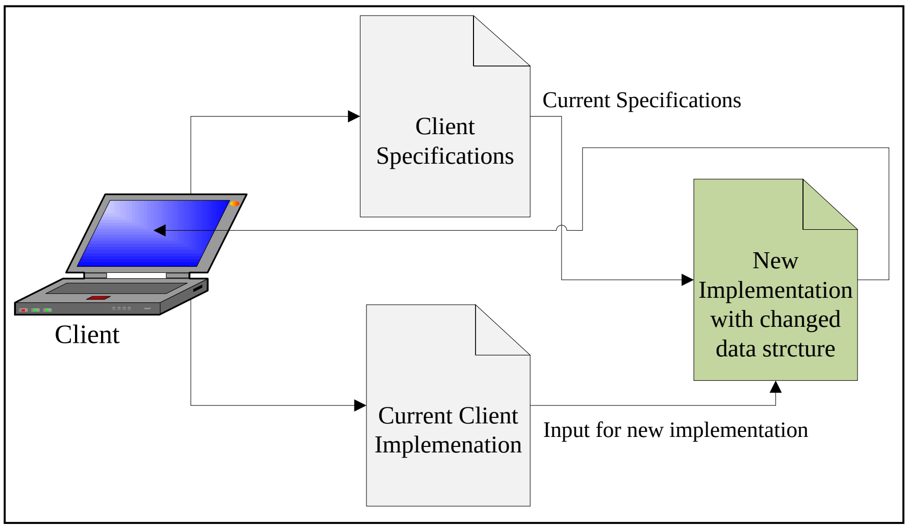

Client Specifications
Current Specifications
New Implementation with changed data strcture
Client
Current Client Implemenation
Input for new implementation

Figure 3.1: Solution

The above diagram is representing the solution in case of already verified program. For supporting refinement regardless of any specification and programming language we are designing a generic framework based on data structure refinement as below.

3.2 Solution Requirements

Below are the requirements for designing our ideal system which will be fully automatic and support data structure based refinement with the capability to generate new implementations.

- Library of data structures to be chosen for new implementations
- Data refinement Language
- Current Specifications

- ♦ New Implementation details
- ♦ Abstraction invariant which relates the current specification with new implementation
- ♦ Operational knowledge of the new implementation such as metadata
- ♦ A GUI building tool which supports drag and drop features

3.3 Solution Design

Below is the block diagram of the system where different components of the system are interacting with each other

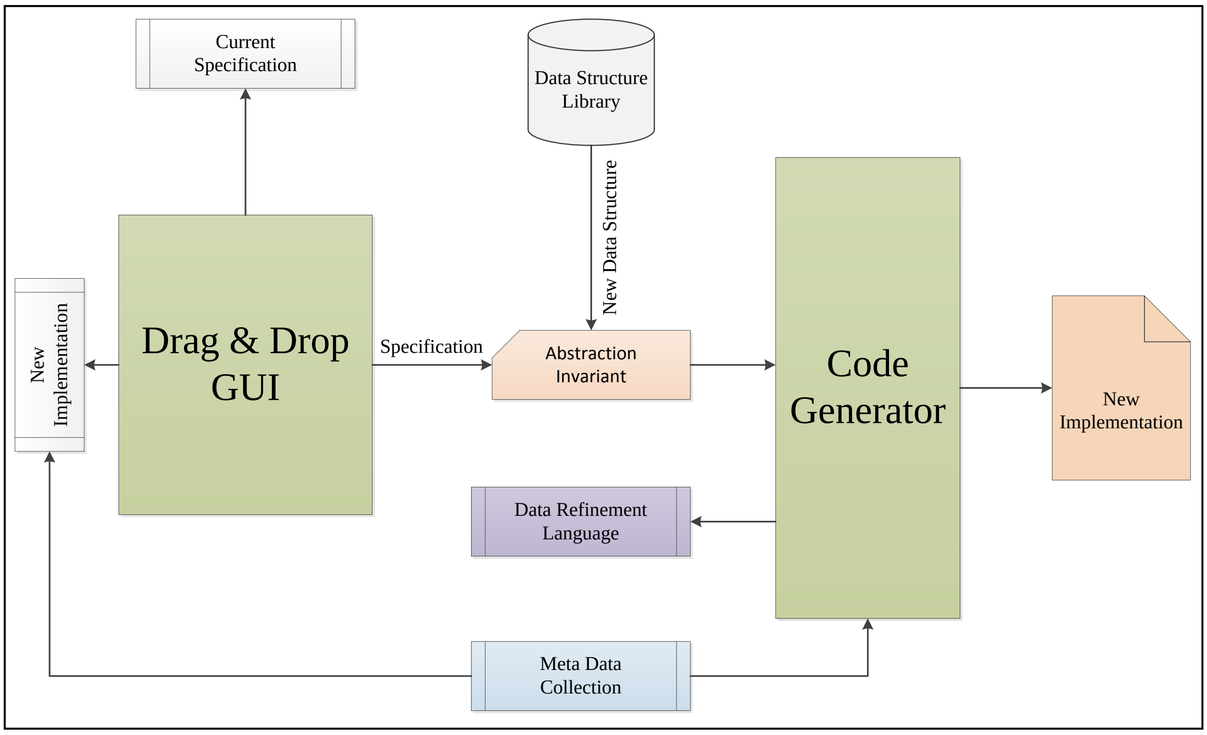


Figure 3.2: Generic System Design (Specification Reuser)

Drag and drop GUI feature selects the selects the specification and corresponding implementations. Meta data collection process is responsible for collecting the operational data of the new implementation such as method operations, variables information and logic of the program. Abstraction invariant provides the relation between abstract and concrete data where data refinement language refines the data for new implementations. Code generator takes Meta data and abstraction invariant as input and generates the new implementation. Class diagram of the system is present in appendix A.1.

The class diagram is representing different potential classes which are grouped into different packages such as GUI package contains classes which work behind the user interface. Specification reader class reads the current specification of the system and implementation reader class reads the new implementation of the system. The class for new implementation is responsible for collecting data for new implementation by getting the data structure details from the supported data structure library. Meta data collection class is responsible for collecting operational knowledge for the new implementation. The abstraction invariant class checks the validity of abstract and concrete data and generator class is using the rule class and the refinement validation and refinement packages for generating the new implementation.

3.4 Interface Design

Our proposed system has very interactive design so that the user can drag and drop different specification and corresponding implementations. Below is the prototype of the interface

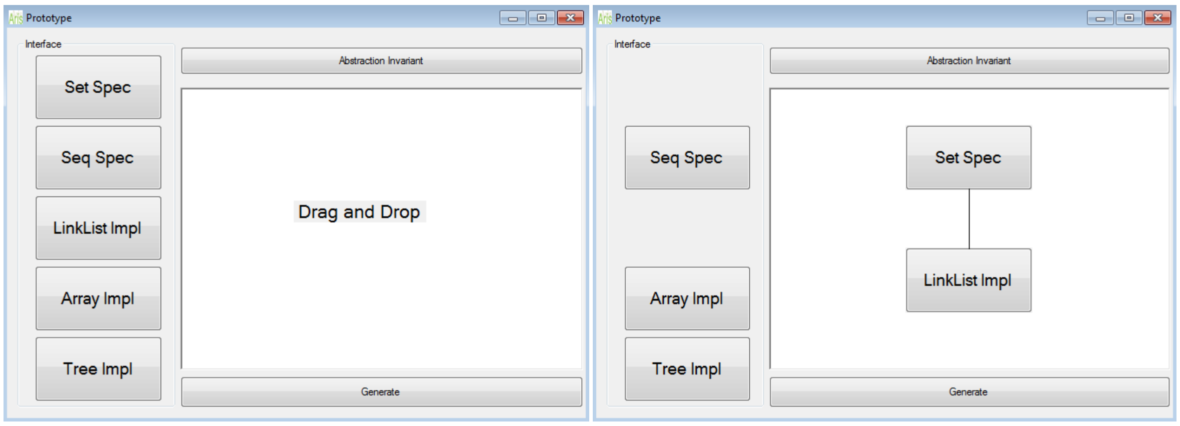


Figure 3.3: User Interface

In **Figure 3.3** 'Spec' tabs are representing specifications in term of sets and sequences where 'Impl' tabs are representing implementable data structures. The abstraction invariant button is getting the abstraction invariant for new implementation with respect to specification from the user.

In **Figure 3.4** specification is same but this time we are generating implementation using array as the underlying data structure. The new abstraction invariant is required in this case that will relate the set specification to array implementation.

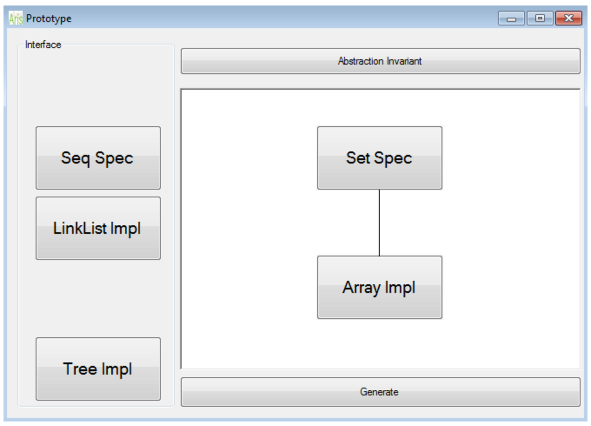


Figure 3.4: User Interface

3.5 Conclusion

In this chapter we provided an overall design of the generic framework. We discussed the generic solution requirement and presented the block diagram and class diagram along with an ideal user interface where users can drag and drop the components for producing new implementations. In the next chapter we will design our proof of concept tool that performs refinement based on data structure.

4 :-: Solution Implementation

In this chapter we present the implementation of our tool and discuss how much we have achieved in reality as compare to our ideal system design in the previous chapter. We use Dafny as a data refinement language and use its native support for implementing the solution.

4.1 Design decision

There are many limitations for implementing our generic solution in the previous chapter such as unavailability of external data structure library lack of data refinement process and lack of support for Meta data collection. We are taking the case of our problem statement where we already have verified program and want to generate another implementation using different data structures from the existing implementation. For implementing the solution to the sproblem we are seeking data refinement support from the language and thus limiting the data structure usage to arrays and link list because they are relatively less complex to verify.

4.1.1 Language Selection

We are seeking data refinement support from the language for implementing our solution; Dafny is the language which is specially designed to support data refinement.

4.1.2 The impact of using Dafny

As we discussed earlier that Dafny support the rich features for writing specifications by providing ghost variables, it handles frame conditions very efficiently and provides greater support for data refinement. Apart from all Dafny do not allow sub classing that makes it very simple to handle refinement as compared to the two class approach. Currently data structure support is very limited in Dafny and hence defining abstraction invariants are relatively simple as compare to other languages such as Spec#. By using Dafny we can move towards for achieving our ideal system goal and advance tools can be developed for Dafny that uses the data structure based refinement feature.

4.2 Tool Implementation

Code generation is dependent on many factors such as specification constructs and support provided by the language for refining specification constructs. As Dafny follows one class approach and currently it does not allow sub typing and do not have support for complex data structure hence our focus of an automatic implementation generation is on one class approach. We are proposing different rules for each separate implementation where we have an abstraction function which maps the relation between abstract and concrete data which goes as input to the tool. The tool will generate the new implementation based on information from abstraction relation, rules and program logic (Meta data). Our approach is to first define these rules and then based on these rules we implement the tool. We are defining different rules for different implementations because different data structure implementations may have different pre and post conditions, loop termination logic, different operational parameters for methods and different conditional logic. Our objective is to provide a tool which can generate automatic implementation close to verification where programmers can start adding code for full verification of the program. We are considering array and linked-list as data structure because they are relatively easy to verify.

4.3 Guidelines

Below are guidelines for generating multiple implementations

4.3.1 Sequence to Array Conversion

In this case we have the specification and implementation both in term of sequences. In Dafny sequences can grow dynamically while arrays cannot, which produces inconsistency while writing the abstraction invariant. The solution to this problem is to send array as parameters to every method including validity function. Below guidelines are Dafny specific

- Collecting Meta data of the current implementation by taking input from the user
- Define arrays corresponding to abstract sequences in the current implementation
- Pass all arrays as parameters to every method
- Pass indexes as parameters as per arrays count because these indexes are needed to adhere design by contract principle
- Replace sequence concatenation operation with array element insertion operator
- Replace sequence length with the length of arrays such as “array.Length” instead of |seq| as no special termination is needed for loop in this case

Providing abstraction invariant, Meta data and above guidelines the implementations that was written in term of sequences can be converted into array implementation without changing specifications. This new implementation may be a semi verified program because right now we do not know fully that how to collect Meta data or operational knowledge of the existing implementation.

4.3.2 Sequence to link-list Conversion

In this case we have specification in term of sequences and implementation in term of link-list. Link list imposes over head for verification because of the nature of the data structure as it is more difficult to write abstraction invariant and hence implementation as we calculated in our metric in chapter 2. Below are some general guidelines for implementation conversion.

- Collecting Meta data of the current implementation by taking input from the user
- Declaring abstract node to start the link list
- Replacing sequence concatenation operator with the new node
- Replacing loops with the following properties for every loop in the program
    - Declare a new node and points it to head
    - Replace loop with condition ‘new node is not equal to null’
    - Provide invariant based on the loop
    - Provide termination of the loop
    - Provide increment on the list

Providing the abstraction invariant and above guidelines a semi verified program can be generated based on the above guidelines. The program will be semi verified because right now it cannot be judged fully about loop invariants, pre and post conditions from the old implementation of the sequence.

4.4 Specification Reuser

We discussed our ideal design of the Specification Reuser in the previous chapter and we discussed the limitation that we have for implementing our ideal system. In this section we are designing and implementing our proof of concept tool that demonstrate the concept that we are trying to achieve from this work. The development of the tool is using the software engineering principles.

4.4.1 System Specifications

A Specification Reuser system that has the capability to generate multiple implementations from one single specification.

4.4.2 System Requirements

The basic requirement is to get the abstract function, existing implementation and Meta data from the user and generates multiple implementations. The tool should provide proper error handling of input to wrong abstraction function and provides users with information with non convertible implementation. Prompt user to get Meta data of the current implementation. Apart from the core functionality tool should be user friendly and can be used by non expert users.

4.4.3 System Design

The system consists of different modules. The graphical user interface where user input the Meta data information input abstract function and selects the existing implementation file. Abstraction invariant, Meta data and file goes as input to the validity checker where system checks the abstraction invariant with respect to existing implementation file. After validity check Mea data, current implementation and abstraction invariant goes as input to code generation module where it applies the rules and generate the new file with same specifications. **Figure 4.1** shows the use case diagram that is describing the user interaction with the system followed by the block diagram in **Figure 4.2**

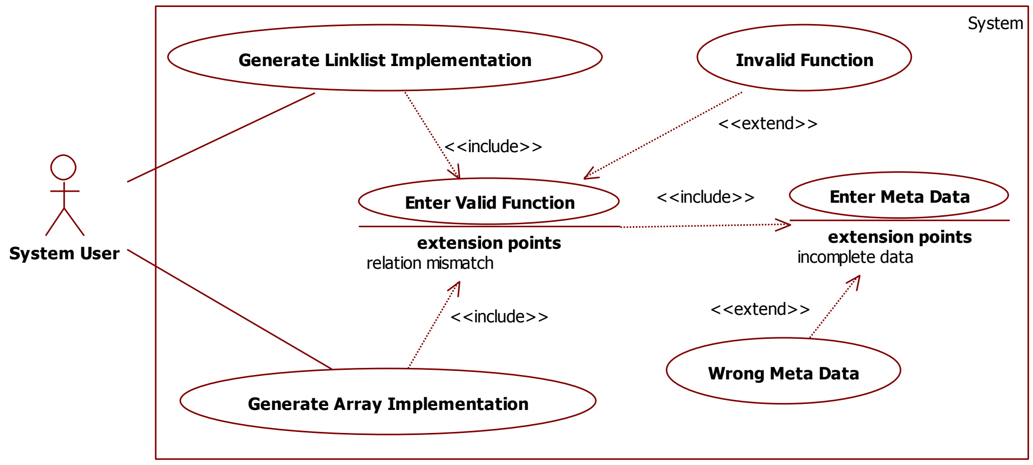


Figure 4.1: Specification Reuser Use Case Diagram

In block diagram existing implementation and abstraction function goes as input to validity checker. The task of the validity checker is to check the abstraction function's validity with existing and new implementation. For existing implementation it checks that specifications are same and for new implementation it checks that the new implementation is compatible and generateable from existing specifications

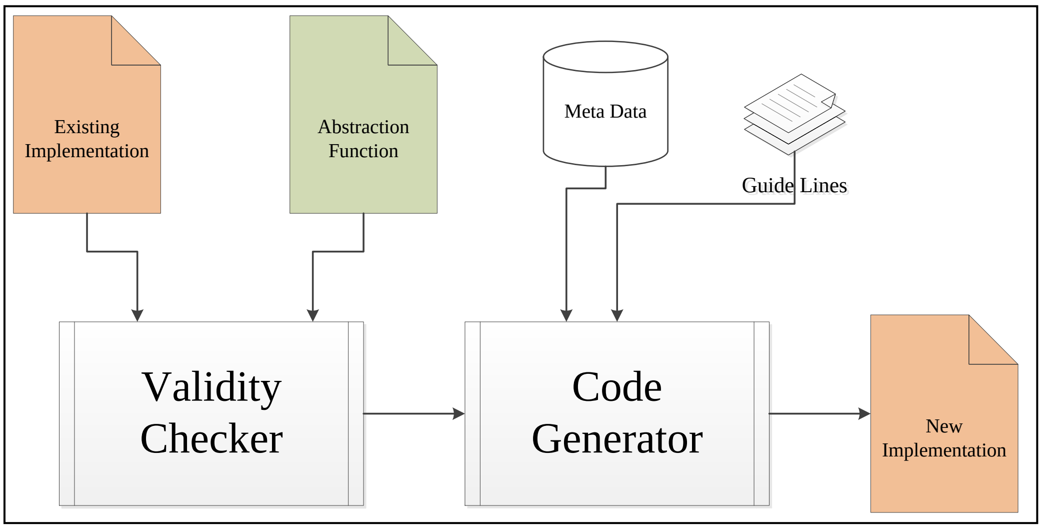


Figure 4.2 : Block diagram of Specification Reuser

The validity checker output becomes the input of the code generator which generates the code by taking the guidelines and Meta data into account and resulting new semi verified code. Class diagram of the system is present in appendix A.2.

4.4.4 Implementation

The table below describes the classes and their operations

Class Name	Description	Methods	Description
code_generator	Class for handling main GUI operations	seq_array_CheckedChanged	Handling check box for sequence to array conversion
		seq_linklist_CheckedChanged	Handling check box for sequence to link list conversion
		SequenceArrayProcessing	Handles conversion from sequence to array
		SequenceToLinkListProcessing s	Handles conversion from sequence to link list
logicAndArrayGener ation	Handling main logic of the tool and generating array implementation	storeGhostData	Storing ghost data of the current implementation
		storeConcreteData	Storing concrete data of the current implementation
		removeConcreteVariable	Removing concrete variables from the current implementation
		changeParamsandPrePostCond	Handling pre and post conditions for array implementation
		changeInitialization	Handling Initialization part of the current implementation
		seqToArrayAdditio	Converting sequence addition to array insertion
		addIndexestoParametersSTAddi ton	Adding indexes to function parameters for array implementation
		storeMethodsData	Handling and storing

			methods information such as method name, opening and closing braces position
		readBytesOfFile	Reading all file data into bytes
		getByteofFile	Getting all bytes of the file
		readNumberAndLengthofLines	Reading number of lines and length of each line from the file
		getFileStorage	Getting the array for number of lines and length of each line in the file
		getNumberofLines	Returning current number of lines in the file
		getReadingPath	Getting the reading path of the file
		getWrittingPath	Getting the writing path of the modified file
		getWrittingPathForLinkList	Getting the writing path for link list implementation
		readNumberAndLengthofLines FromStringBuilder	Reading current number of lines along with length of each line from changed file in memory
		chaingingLoops	Handling the loop conversion from sequence to array
		handlingValidFunctionConditio ns	Adding indexes to valid function for verification of array implementation
metaDataCollector	Collecting metadata of the current file by taking input from the user	record_btn_Click	Adding meta data objects into storeMetaData class
		finish_btn_Click	Closing the current window of meta data collector
		opertionDesctipion_Load	Reading all the methods from the file and populating it as a drop down list
storeMetaData	Responsible for storing meta data information which is collected from front end metadata collector	setOpName	Recording the method name
		setOpDesc	Recording the method description such as “INT for initiation”, “FD for searching” and “AD for addition”
		getOpName	Returning method name
		getOpDesc	Returning its meta information
methodsDataHolder	Responsible for storing methods information such as method name, its opening and closing braces line numbers	setMethodName	Recording methods name
		setMethodLineNumber	Recording the current line number of the method
		setOpeningBraceLineNumber	Recording the opening brace number of the method
		setClosingBraceLineNumber	Recording the closing brace number of the method
		getMethodName	Returning method name
		getMethodLineNumber	Returning method current line number
		getOpeningBraceLineNumber	Returning method opening brace line number
		getClosingBraceLineNumber	Returning method closing brace line number
dataHolder	Responsible for storing file data such as storing	setAbstractNameType	Setting abstract variable name type as ghost
		setConcreteNameType	Setting concrete variable

	ghost and concrete variable information for data type, variable name and whether its ghost or non ghost		name type as non-ghost
		setAbstractDataType	Setting abstract variable data type
		setConcreteDataType	Setting concrete variable data type
		setAbstractVariableName	Setting ghost variable name
		setConcreteVariableName	Setting non-ghost variable name
		getAbstractNameType	Returning abstract variable type
		getAbstractDataType	Returning abstract variable data type
		getAbstractVariableName	Returning abstract variable name
		getConcreteNameType	Returning concrete variable type
		getConcreteDataType	Returning concrete variable data type
		getConcreteVariableName	Returning concrete variable name
validFunction	Responsible for handling the valid function validity	btnClear_Click	Clearing the contents from the GUI
		btnEnter_Click	Collecting the valid function entered by the user and send it to validity checker
		fillValidFunction	Storing the valid function
		getValidFunction	Returning the valid function
validityChecker	Responsible for checking the validity of the valid function	checkValid	Returning true or false based on the valid function contents
linkListGeneration	Responsible for generating link list implementation	generateNewNodeVariable	Creating new Node variable for link list start up
		changeSpecificationDataType	Changing data type to node for some specification variables for verification purpose
		initializationChanging	Changing initiation from sequence to link list
		commentInitlizationCode	Commenting out unnecessary code from initialization part
		changiningAdditonOperation	Handling addition operation from sequence to link list
		changiningSignatureOfFinding Operations	Handling signatures for the methods based on the meta data information
		changiningInsideFindingOperati ons	Changing loops and other variables such as in IF statements inside methods who have meta data information such as "FD for searching operations"

4.5 Conclusion

In this chapter we presented our tool implementation based on Dafny for data refinement. The tool is implemented using C#. In the next chapter we will demonstrate our tool based on the birth day book case study.

5 :-: Proof of Concept

In this chapter we present a case study using our proof of concept tool. We are taking the same example of Birthday Book which we have described in chapter 2 for data refinement. We are taking the implementation and specification in term of sequence as input and our tool converts it into an array and a link list implementation of birthday book while preserving the specifications.

5.1 Case Study

A system which can record the birthdays of different peoples and can issue the remainder to people who have birthdays on the same date and can find birth date for any particular person.

We have implemented the birthday book case study first by doing one to one mapping from sequence to sequence. As we are using Dafny language, it is possible to use sequence for both specification and implementation. Below is the code and description of this mapping. We will use this mapping as a basis to generate other implementations that differ in their data structure.

5.1.1 Seq to Seq

This example consists of a specification which is in term of sequences and implementation in term of sequence as well. We are connecting this example back to our data refinement example in chapter 2 in order to maintain the symmetry. Initially we have the following birthday book

```
class Data<NAME,DATE> {
var name: NAME;
var date : DATE;
}
ghost var names_a: seq<NAME>;
ghost var dates_a: seq<DATE>;
ghost var Repr: set<object>;
var namesseq_c: seq<Data<NAME,DATE>>;
ghost var elems: seq<Data<NAME,DATE>>;
```

Class ‘Data’ contains two member variables ‘name’ and ‘date’ where ghost variables names_a, dates_a, and elems are representing the abstract data types. These variables are corresponding to set of names and dates and a function from NAME$\rightarrow$DATE in the data refinement example of birthday book in chapter 2. Repr is for dynamic framing (see appendix B) and namesseq_c represents the concrete sequences to be used as underlying data structures. In Dafny abstraction invariant is referred to as validity function which we usually checked before and after the method where possible state change in the system can occur. In this case our validity function is as below

```
method Init()
modifies this;
ensures Valid() && fresh(Repr - {this});
{
names_a := []; dates_a := []; Repr := {this};
namesseq_c := []; elems := []; }
```

```
function Valid(): bool
 reads this, Repr;
{
this in Repr &&
|names_a| == |dates_a| && |elems| == |names_a| &&
|namesseq_c| == |elems| &&
(forall i :: 0 <= i && i < |names_a| ==>
elems[i] != null &&
elems[i] in Repr &&
elems[i].name == names_a[i] &&
elems[i].date == dates_a[i] &&
elems[i] == namesseq_c[i]) }
```

This validity function is describing the abstraction invariant that the concrete sequence which is referred to as ‘namesseq_c’ and abstract sequence as an ‘elems’ have equal length and their elements are also same. The ‘elems’ sequence is also related to abstract sequence ‘names_a’ and ‘names_a’ is related to ‘dates_a’. The’ elems’ sequence is also checking that his data elements such as ‘name’ and ‘date’ are also same with other elements of abstract sequence. Init function is initializing all the abstract and concrete variables to length zero so that the validity function can hold.

```
method Add(name: NAME, date: DATE)
requires Valid();
modifies this, Repr;
ensures Valid() && fresh(Repr - old(Repr));
{
 var h : = new Data<NAME,DATE>;
h.name : = name;
h.date : = date;
names_a : = [name] + names_a;
 dates_a : = [date] + dates_a;
 namesseq_c : = [h] + namesseq_c;
elems : = [h] + elems;
 Repr : = Repr + {h}; }
```

The 'Add' method is adding a name and date pair to the system by creating the new object of data by filling it with input parameters 'name' and 'date' and adding it to the sequence 'namesseq_c'. Sequences 'names_a' and 'dates_a' are also being updated in order to maintain the state of the system for consistency. Dafny does not provide the automatic updation of ghost variables that makes this language more flexible to support data refinement as compare to Spec#. Specification constructs 'require' and 'ensures' are showing that pre and post condition of the function for which valid holds. Following is the find birthday method

```
method FindBB(name: NAME) returns (date :
DATE,ghost n: int)
requires Valid();
modifies this;
{
n : = 0;
var i : = 0;
while (i < |namesseq_c| && namesseq_c[i] != null)
{
if (namesseq_c[i].name == name) {
date : = namesseq_c[i].date; return;
} else {  n : = n + 1;   i : = i + 1;  }}}
```

The 'FindBB' method is searching the birthday for any given name. We are not checking the post condition as validity function because the state of the system is not changing as no updation is being made to the concrete sequence 'nameseq_c'. Dafny supports the return variable as a part of the program; in this example we are using ghost variable "n" which is containing the index of the return name. Moreover Dafny has a feature to return multiple values. Following is the remind birthday method

```
method remindBB(date: DATE) returns (ghost n: int , res
: seq<Data<NAME,DATE>>)
requires Valid();
{
n : = 0;
var i : = 0;
while (i < |namesseq_c| && namesseq_c[i] != null)
{
if (namesseq_c[i].date == date) {
res : = [namesseq_c[i]] + res;  i : = i +1 ;  n : = n +1 ;
} else { i : = i +1;  n : = n + 1;  } }
```

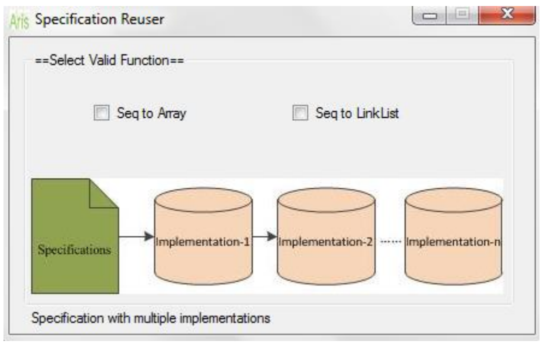
The method 'remindBB' is returning the sequence of names and date pair for a specific date. The value return by the method is the names of sequence which contains all the names that have birthday on the specific 'date'. Both 'FindBB' and 'remindBB' functions are containing 'while' loop and both are iterating over the sequence 'namesseq_c' bounds. Now we use our tool to generate the next implementation.

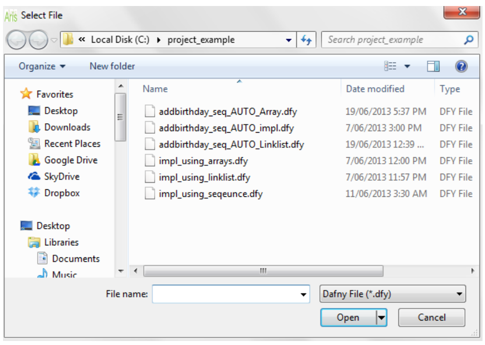
5.1.2 Tool Demonstration

In this section we are demonstrating our tool. We are generating the two implementations of birth day book system one which is using array and other is using link list as underlying data structure by following below steps.

Step-1

Select the implementation you want to generate and then select a file




Select File

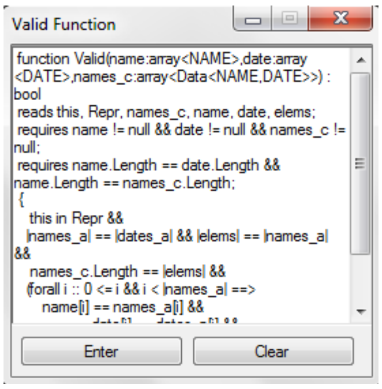
Local Disk (C:) ▸ project_example

Name	Date modified	Type
addbirthday_seq_AUTO_Array.dfy	19/06/2013 5:37 PM	DFY File
addbirthday_seq_AUTO_impl.dfy	7/06/2013 3:00 PM	DFY File
addbirthday_seq_AUTO_Linklist.dfy	19/06/2013 12:39 ...	DFY File
impl_using_arrays.dfy	7/06/2013 12:00 PM	DFY File
impl_using_linklist.dfy	7/06/2013 11:57 PM	DFY File
impl_using_seqeunce.dfy	11/06/2013 3:30 AM	DFY File

File name: Dafny File (*.dfy)

Step-2

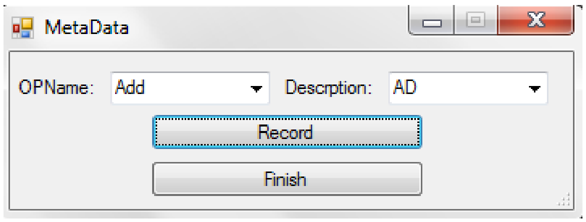
Enter the valid function after selecting


Valid Function

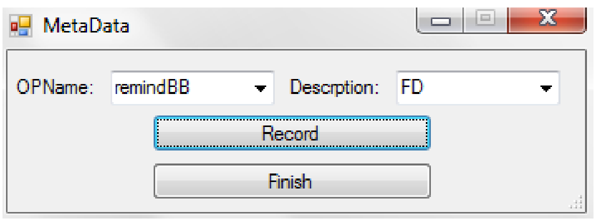
```
function Valid(name:array<NAME>,date:array
<DATE>,names_c:array<Data<NAME,DATE>>) :
bool
 reads this, Repr, names_c, name, date, elems;
 requires name != null && date != null && names_c !=
null;
 requires name.Length == date.Length &&
name.Length == names_c.Length;
 {
   this in Repr &&
   |names_a| == |dates_a| && |elems| == |names_a|
&&
   names_c.Length == |elems| &&
   (forall i :: 0 <= i && i < |names_a| ==>
      name[i] == names_a[i] &&
```

Step-3

Enter Meta data information


MetaData

OPName: Add Descrption: AD

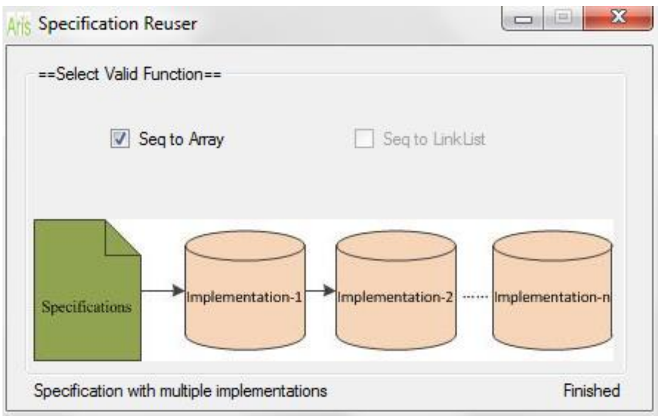

MetaData

OPName: remindBB Descrption: FD

Where "AD and "FD" are representing addition and searching. Repeat this process for every method listed in the OpName drop down menu.

Step-4

Check the new Implemented File.



5.1.3 Seq to link-list

Following birthday book system is generated from our tool except validity function using our proof of concept tool which is using link list as underlying data structure. Below are the details of the generated code. Following is the initial birthday book

```
class Node<NAME,DATE> {
var name: NAME;
var date: DATE;
var next: Node<NAME,DATE>; }
ghost var names_a: seq<NAME>;
ghost var dates_a: seq<DATE>;
ghost var Repr: set<object>;
var head: Node<NAME,DATE>;
ghost var nodes: seq<Node<NAME,DATE>>;
```

We are taking the Node class which has three data members name, date and next (which is the pointer to next node in the linked list). Specifications are same and are in term of sequences except "nameseq_c" is replaced by "head" of type Node, which is representing the first node in the link list. Following is the validity function supplied by the user

```
method Init()
modifies this;
ensures Valid() && fresh(Repr - {this});
ensures |names_a| == 0;
{  names_a := [];   dates_a := [];   Repr := {this};
head := null;    nodes := [null];  }
```

```
function Valid(): bool
 reads this, Repr;
{
this in Repr &&
|names_a| == |dates_a| && |nodes| == |names_a| + 1 &&
head == nodes[0] &&
(forall i :: 0 <= i && i < |names_a| ==> nodes[i] != null &&
nodes[i] in Repr && nodes[i].name == names_a[i] &&
nodes[i].date  ==  dates_a[i]  &&      nodes[i].next  ==
nodes[i+1])
&& nodes[|nodes|-1] == null }
```

Validity function has the following descriptions. 'head' is pointing to first node in the link-list, where it is a concrete data type and 'nodes' is the abstract data type. Moreover abstract data "names_a" and "dates_a" are being compared for length and elements validity with 'nodes' sequence where the next node of 'nodes' is also being validated. One thing is to notice here that "names_a" has length one greater than the "nodes" due to last additional "null" node. Following is the add birthday function.

```
method Add(name: NAME, date: DATE)
requires Valid();
modifies Repr;
 ensures Valid() && fresh(Repr - old(Repr));
{
var h := new Node<NAME,DATE>;
h.name := name;  h.date := date;  h.next := head;
head := h;
names_a  :=  [name]  +  names_a;    dates_a  :=  [date]  +
dates_a;
nodes := [h] + nodes;
Repr := Repr + {h};  }
```

The 'Add' method is adding a name and date pair to the system by declaring new object of a node class and filling its data members and assigning head to current node and keep. Following is the find birthday method

```
method FindBB(name: NAME) returns
(curr: Node<NAME,DATE>, ghost n: int, prev:
Node<NAME,DATE>)
requires Valid();
{
n := 0;
prev := null;
curr := head;
while (curr != null)
invariant n <= |names_a| && curr == nodes[n];
decreases |names_a| - n;
{
if (curr.name == name) {
return;
} else {
n := n + 1;  prev := curr; curr := curr.next;   }  }  }
```

The 'FindBB' method is searching the birthday for any given name. The searching in link list is entirely different from searches in sequences and arrays. In arrays and sequences we have lengths to bound our loop execution but in link list we have to supply the termination. Here it is being proved with the help of abstract sequence |names_a|. Method is returning current and previous node along with node number. Following is the remind birthday function.

```
method remindBB(date: DATE) returns
(curr : Node<NAME,DATE> , ghost n: int , prev :
Node<NAME,DATE>, res :
seq<Node<NAME,DATE>>)
requires Valid();
{
 n := 0;
 prev := null;
 curr := head;
 while (curr != null)
invariant n <= |dates_a|&& curr == nodes[n];
decreases |names_a| - n; {
```

```
{
if (curr.date == date) {
res := [curr] + res;  n := n+1;  prev := curr;  curr :=
curr.next;
} else {
 n := n + 1;  prev := curr;  curr := curr.next;   }  } }
```

The method 'remindBB' is returning the current and previous node numbers along with sequences of nodes containing name and date pair. Below is the second implementation which generated from our tool by following steps described in **section 5.1.2**.

5.1.4 Seq to Array

Following birthday book system is generated from our tool except validity function using our proof of concept tool which is using array as underlying data structure. Below are the details of the generated code. Following is the initial birthday book

```
ghost var names_a: seq<NAME>;
ghost var dates_a: seq<DATE>;
```

```
ghost var Repr: set<object>;
ghost var elems: seq<Data<NAME,DATE>>;
```

In this implementation specifications are same except it operates on concrete array. The arrays are allocated and sent as function arguments. Following is the initialize and validity function.

```
method Init(name: array<NAME>, date: array<DATE>,
name_date: array<Data<NAME,DATE>>)
requires name_date != null && name != null && date !=
null;
requires name_date.Length >= 0 && name.Length ==
date.Length && name_date.Length == name.Length;
modifies this;
ensures fresh(Repr - {this});
ensures Valid(name,date,name_date);
ensures |names_a| >= 0;
{
var i := 0;   names_a := name[..];   dates_a := date[..];
Repr := {this};    elems := name_date[..];    }

function Valid(name: array<NAME>,
```

```
date: array<DATE>,
names_c: array<Data<NAME,DATE>>) : bool
reads this, Repr, names_c, name, date, elems;
requires name != null && date != null && names_c !=
null;
 requires name.Length == date.Length && name.Length
== names_c.Length;
{
this in Repr &&
|names_a| == |dates_a| && |elems| == |names_a| &&
names_c.Length == |elems| &&
(forall i :: 0 <= i && i < |names_a| ==>
name[i] == names_a[i] &&
date[i] == dates_a[i] &&
elems[i] == names_c[i]) }
```

Initialize function is initializing the abstract sequences from the arrays (feature supported by Dafny) so that we can check the length of the arrays and sequences after any state change. The validity function is also checking the length and elements for input arrays. As arrays are passed by reference in Dafny so we can use arrays as function arguments to check the validity of the system because they remain same throughout the program. Following is the add birthday function.

```
method Add(name: NAME, date: DATE,
namearr: array<NAME>,datearr: array<DATE>,names_
c: array<Data<NAME,DATE>>, index: int)
requires names_c != null && namearr != null && datearr
!= null;
requires names_c.Length >= 0 && namearr.Length ==
datearr.Length && names_c.Length ==
namearr.Length;
requires Valid(namearr,datearr,names_c);
```

```
requires 0<= index < names_c.Length;
modifies this, Repr, names_c;
ensures Valid(namearr,datearr,names_c);
{
var h := new Data<NAME,DATE>;
h.name := name;  h.date := date;  names_a := [name] +
names_a;  dates_a := [date] + dates_a;
elems := [h] + elems;  names_c[index] := h;
Repr := Repr + {h};  }
```

The 'Add' method is adding a name and date pair to the system by declaring an object of Data class and filling the "names_c" array with the data object by using index from methods parameters. As Dafny requires the bounds to be valid for any array as a pre condition so we need to specify the index range as the pre condition of the method. Add function is updating both concrete array and abstract sequences for maintaining the validity of the system. Following is the find birth day method.

```
method FindBB(name: NAME,
namearr: array<NAME>,date: array<DATE>,names_c: a
rray<Data<NAME,DATE>>) returns (curr:
```

```
Data<NAME,DATE>, ghost n: int)
requires names_c != null && namearr != null && date !=
null;
```

```
requires names_c.Length >= 0 && namearr.Length ==
date.Length && names_c.Length == namearr.Length;
 requires Valid(namearr,date,names_c);
{
n : = 0;
var i : = 0;
while (i < names_c.Length && names_c[i] != null)
{
if (names_c[i].name == name) {
curr : = names_c[i];   return;
} else {
i : = i +1;  n : = n + 1;  }
}
}
```

The 'FindBB' method is searching the birthday for any given name. This method is operating on the concrete array that is coming as input parameter to the method and returning the object of Data class with which input variable 'names' matches. Array bound is being checked with the built-in array length function. Following is the remind birthday function

```
method remindBB(date:  DATE,
name: array<NAME>,datearr: array<DATE>,names_c: a
rray<Data<NAME,DATE>>) returns  (ghost n: int , res :
seq<Data<NAME,DATE>>)
 requires names_c != null && name != null && datearr !=
null;
 requires names_c.Length >= 0 && name.Length ==
datearr.Length && names_c.Length == name.Length;
 requires Valid(name,datearr,names_c);  {
n : = 0;
var i : = 0;
while (i < names_c.Length && names_c[i] != null)
{
if (names_c[i].date == date) {
res : = [names_c[i]] + res;
n : = n +1 ;
return;
} else {     i : = i +1;    n : = n + 1;   }  } }
```

The method 'remindBB' is using a concrete array for searching. Method is returning the sequence containing all names those have birthdays on the supplied input date.

5.2 Conclusion

In this chapter we demonstrate our tool using Birthday book system case study and explained the generated code. In next chapter we evaluate our solution and discuss the validation of our work along with weakness that still needs to be address.

6 :-: Evaluation

In this chapter we evaluate and validate our work. We present the graphical comparison between automatic and manual implementation. We discuss the validation of our work in the form of verification for the generated implementation. We criticize our work and analyze it with related work.

6.1 Evaluation and Validation

Our main idea behind this project is to generate automatic implementation using data structure based refinement so that we can reuse our specifications. We are evaluating our work based on the automation we have achieved so far. Below are the tables for different implementations that we are using for evaluation and validation

Evaluation

Specifications	Data Structure	Generation	Comments
Sequence	Arrays	Automatic	Specification is in term of a sequence and implementation is in term of arrays
Sequence	Link List	Automatic	Specifications is in term of a sequence and implementation is in term of link list

Validation

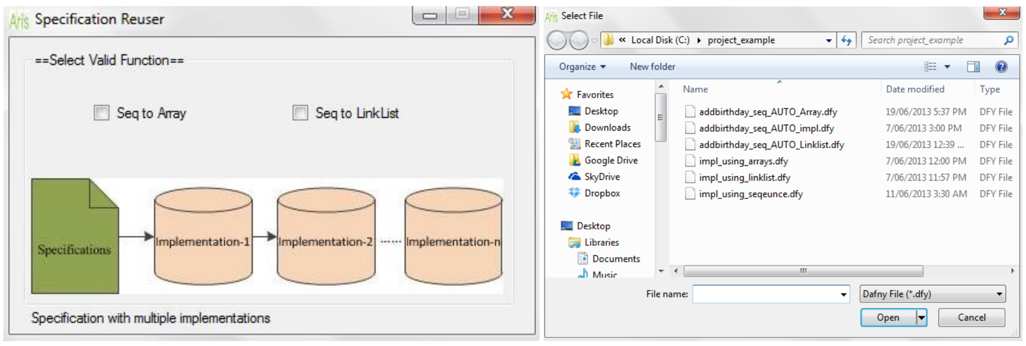
Specifications	Data Structure	Generation	Verification
Sequence	Arrays	Automatic	Semi
Sequence	Link List	Automatic	Semi but in our case study its fully verified

6.2 Experimental setup

We now use our developed tool for evaluating our results.

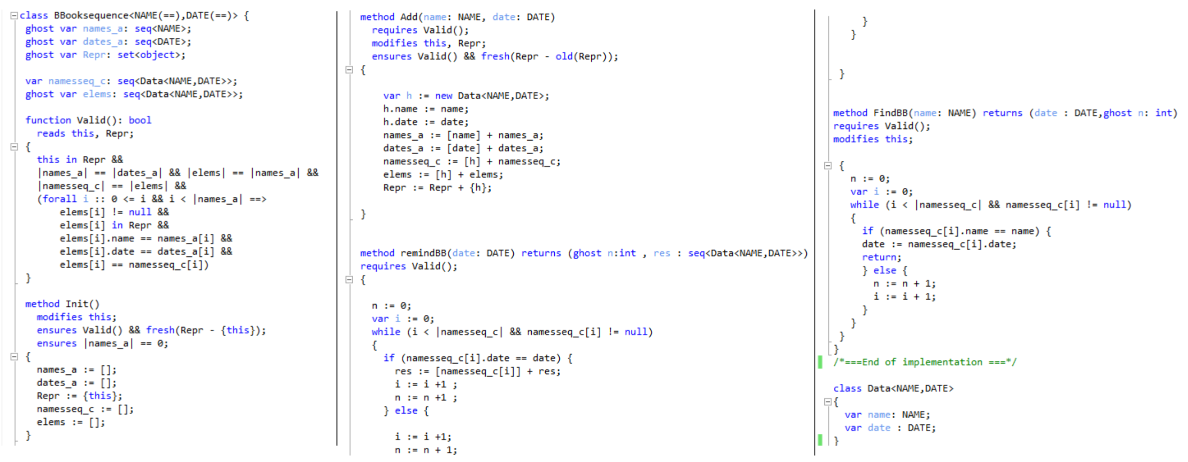
Evaluation

Step-1
Select the implementation you want to generate and then select the file



```
Name                               Date modified         Type
addbirthday_seq_AUTO_Array.dfy     19/06/2013 5:37 PM    DFY File
addbirthday_seq_AUTO_impl.dfy      7/06/2013 3:00 PM     DFY File
addbirthday_seq_AUTO_Linklist.dfy  19/06/2013 12:39 ...  DFY File
impl_using_arrays.dfy              7/06/2013 12:00 PM    DFY File
impl_using_linklist.dfy            7/06/2013 11:57 PM    DFY File
impl_using_seqeunce.dfy            11/06/2013 3:30 AM    DFY File
```

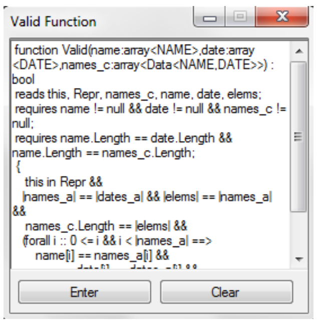
This is the existing implementation in term of sequence



```
class BBooksequence<NAME(==),DATE(==)> {
 ghost var names_a: seq<NAME>;
 ghost var dates_a: seq<DATE>;
 ghost var Repr: set<object>;

 var namesseq_c: seq<Data<NAME,DATE>>;
 ghost var elems: seq<Data<NAME,DATE>>;

 function Valid(): bool
   reads this, Repr;
 {
   this in Repr &&
   |names_a| == |dates_a| && |elems| == |names_a| &&
   |namesseq_c| == |elems| &&
   (forall i :: 0 <= i && i < |names_a| ==>
       elems[i] != null &&
       elems[i] in Repr &&
       elems[i].name == names_a[i] &&
       elems[i].date == dates_a[i] &&
       elems[i] == namesseq_c[i])
 }

 method Init()
   modifies this;
   ensures Valid() && fresh(Repr - {this});
   ensures |names_a| == 0;
 {
   names_a := [];
   dates_a := [];
   Repr := {this};
   namesseq_c := [];
   elems := [];
 }

 method Add(name: NAME, date: DATE)
   requires Valid();
   modifies this, Repr;
   ensures Valid() && fresh(Repr - old(Repr));
 {

     var h := new Data<NAME,DATE>;
     h.name := name;
     h.date := date;
     names_a := [name] + names_a;
     dates_a := [date] + dates_a;
     namesseq_c := [h] + namesseq_c;
     elems := [h] + elems;
     Repr := Repr + {h};

 }


 method remindBB(date: DATE) returns (ghost n:int , res : seq<Data<NAME,DATE>>)
 requires Valid();
 {

   n := 0;
   var i := 0;
   while (i < |namesseq_c| && namesseq_c[i] != null)
   {
     if (namesseq_c[i].date == date) {
       res := [namesseq_c[i]] + res;
       i := i +1 ;
       n := n +1 ;
     } else {

       i := i +1;
       n := n + 1;
       }
     }


 }


 method FindBB(name: NAME) returns (date : DATE,ghost n: int)
 requires Valid();
 modifies this;

 {
   n := 0;
   var i := 0;
   while (i < |namesseq_c| && namesseq_c[i] != null)
   {
     if (namesseq_c[i].name == name) {
     date := namesseq_c[i].date;
     return;
     } else {
       n := n + 1;
       i := i + 1;
     }
   }
 }
}
 /*===End of implementation ===*/

 class Data<NAME,DATE>
{
  var name: NAME;
  var date : DATE;
}
```

Step-2

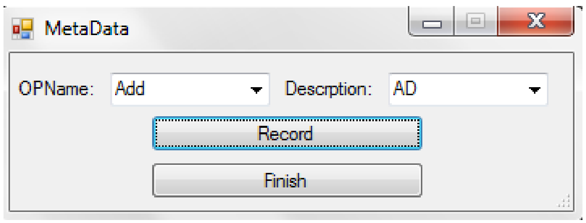
Enter the valid function for new implementations



```
function Valid(name:array<NAME>,date:array
<DATE>,names_c:array<Data<NAME,DATE>>) :
bool
 reads this, Repr, names_c, name, date, elems;
 requires name != null && date != null && names_c !=
null;
 requires name.Length == date.Length &&
name.Length == names_c.Length;
 {
   this in Repr &&
  |names_a| == |dates_a| && |elems| == |names_a|
&&
   names_c.Length == |elems| &&
  (forall i :: 0 <= i && i < |names_a| ==>
     name[i] == names_a[i] &&
```

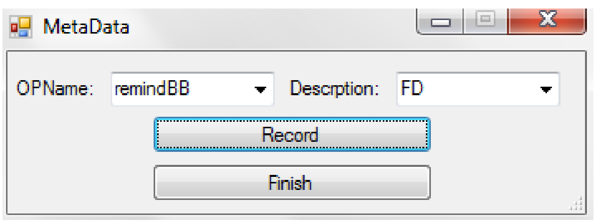
Step-3

Enter Meta data information



MetaData

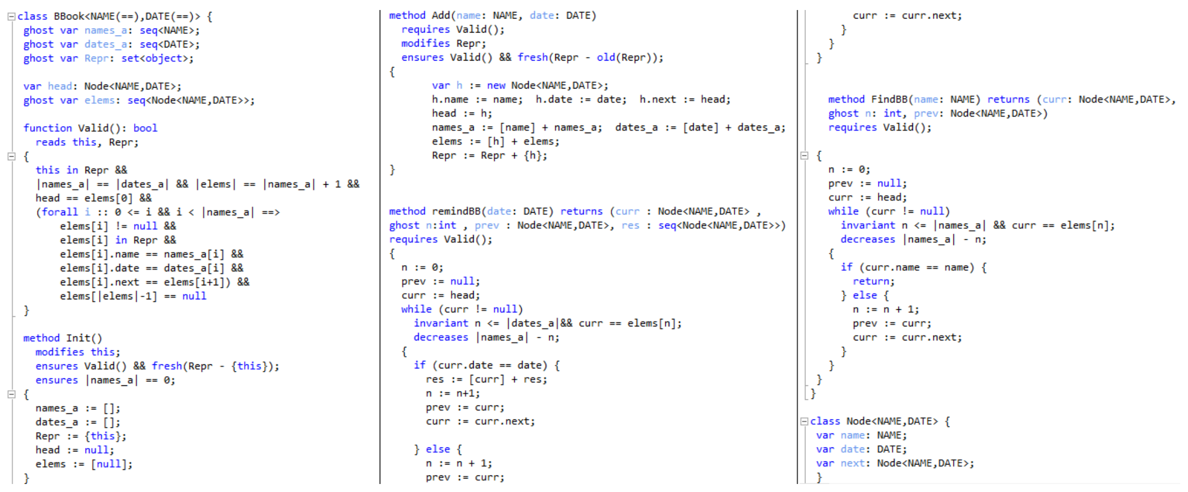
OPName: Add Descrption: AD



MetaData

OPName: remindBB Descrption: FD

Press finish to generate the implementation. Below is the generated implementation which is using link list as underlying data structure



```
class BBook<NAME(==),DATE(==)> {
 ghost var names_a: seq<NAME>;
 ghost var dates_a: seq<DATE>;
 ghost var Repr: set<object>;

 var head: Node<NAME,DATE>;
 ghost var elems: seq<Node<NAME,DATE>>;

 function Valid(): bool
   reads this, Repr;
 {
   this in Repr &&
   |names_a| == |dates_a| && |elems| == |names_a| + 1 &&
   head == elems[0] &&
   (forall i :: 0 <= i && i < |names_a| ==>
       elems[i] != null &&
       elems[i] in Repr &&
       elems[i].name == names_a[i] &&
       elems[i].date == dates_a[i] &&
       elems[i].next == elems[i+1]) &&
       elems[|elems|-1] == null
 }

 method Init()
   modifies this;
   ensures Valid() && fresh(Repr - {this});
   ensures |names_a| == 0;
 {
   names_a := [];
   dates_a := [];
   Repr := {this};
   head := null;
   elems := [null];
 }

method Add(name: NAME, date: DATE)
  requires Valid();
  modifies Repr;
  ensures Valid() && fresh(Repr - old(Repr));
{
       var h := new Node<NAME,DATE>;
       h.name := name;  h.date := date;  h.next := head;
       head := h;
       names_a := [name] + names_a;  dates_a := [date] + dates_a;
       elems := [h] + elems;
       Repr := Repr + {h};
}


method remindBB(date: DATE) returns (curr : Node<NAME,DATE> ,
ghost n:int , prev : Node<NAME,DATE>, res : seq<Node<NAME,DATE>>)
requires Valid();
{
  n := 0;
  prev := null;
  curr := head;
  while (curr != null)
    invariant n <= |dates_a|&& curr == elems[n];
    decreases |names_a| - n;
  {
    if (curr.date == date) {
      res := [curr] + res;
      n := n+1;
      prev := curr;
      curr := curr.next;

    } else {
      n := n + 1;
      prev := curr;
      curr := curr.next;
    }
  }
 }


  method FindBB(name: NAME) returns (curr: Node<NAME,DATE>,
  ghost n: int, prev: Node<NAME,DATE>)
  requires Valid();

 {
  n := 0;
  prev := null;
  curr := head;
  while (curr != null)
    invariant n <= |names_a| && curr == elems[n];
    decreases |names_a| - n;
  {
    if (curr.name == name) {
      return;
    } else {
      n := n + 1;
      prev := curr;
      curr := curr.next;
    }
  }
 }
}

class Node<NAME,DATE> {
 var name: NAME;
 var date: DATE;
 var next: Node<NAME,DATE>;
 }
```

<u>Validation</u>

We are using Dafny extension for visual studio for verification of programs. The above screen shot is taken from the visual studio which is showing the absence of any verification error. So the generated implementation of link list is fully verified.

6.3 Results

We have calculated the programmer overhead by using metric in chapter 2. The metric assigns different weights to different data structure according to their complexity. We now analyze our results while keeping in mind the effort required for manual implementation. Tables below describes the statistics that we have discussed earlier

Effort with high Programming and Verification skills with automatic verification

Specification	Data Structures	KLOC	Implementation and Verification effort in term of DT
Sequence/Set	Sequence	500 => 0.5	12 days
Sequence/Set	Link-list	500 => 0.5	20 days
Sequence/Set	Arrays	500 => 0.5	15 days
Sequence/Set	Tree	500=> 0.5	30 days

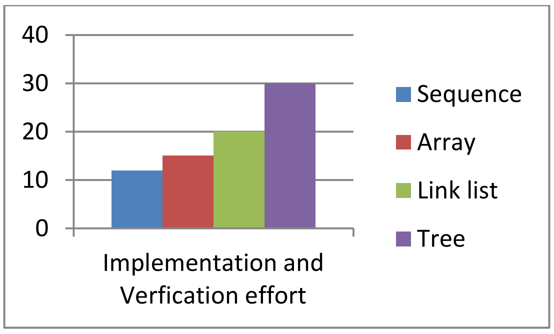
Effort with low Programming and Verification skills with manual verification

Specification	Data Structures	KLOC	Implementation and Verification effort in term of DT
Sequence/Set	Sequence	500 => 0.5	33 days
Sequence/Set	Link-list	500 => 0.5	56 days
Sequence/Set	Arrays	500 => 0.5	43 days
Sequence/Set	Tree	500=> 0.5	85 days

By using Specification Resuer we can see significant difference in implementation and verification effort

Specification	Data Structure	KLOC	Implementation effort	Verification effort	Programming Skills	Verification Skills
Sequence	Arrays	Doesn't matter	Low	Moderate	Low	Moderate
Sequence	Link-list	Doesn't matter	Low	Moderate	Low	Moderate

Below are the graphical representations of the results which are showing effort without using the 'Specification Resuer'.



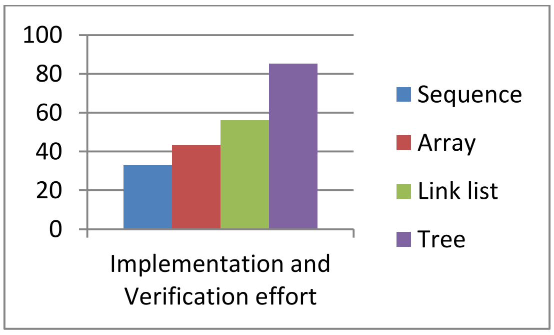


Figure 6.1: Statistics based on cost drivers

Below graph is representing the effort required with using the 'Specification Resuer'.

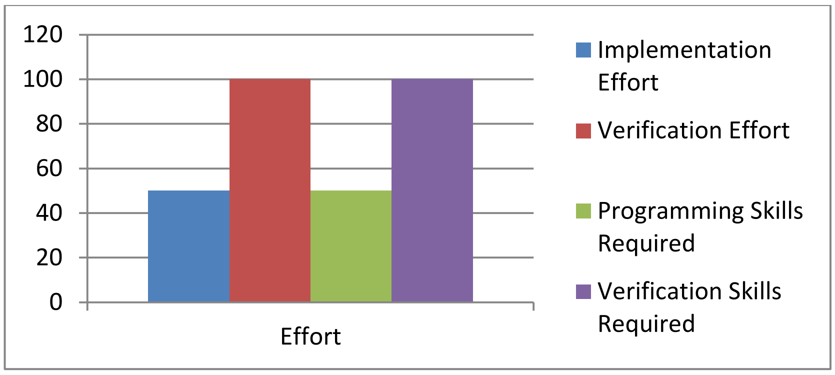


Figure 6.2: Implementation effort using Specification Reuser

Effort is categorized as Low, Moderate and High with values of 50, 100 and 150. As we can see in the first two graphs which are not using Specification Reuser, effort is high in man days while in the second graph there is a significant difference in the effort.

6.4 Critical Analysis

Effort Metric

The metric we have proposed in chapter 2 is based on Cost Constructive Model [23] where we have changed the cost drivers to reflect the verification and implementation effort required to verify the program. The metric is only for rough estimation and designed by keeping in mind the verification and implementation efforts using specification based programming language.

Genericity

Table below is describing the automation we have achieved and issues that are preventing the automation to be fully automatic

Measure	Percentage	Description
Genericity	40%	We can't say right now that our tool is not more than forty percent generic that can transform any implementation written in terms of sequence to array and linked list or to any data structure. There are limitations and reasons which we are summarizing in table below

Issues and Limitations

Below are some limitations and issues with their descriptions which are preventing us to achieve fully automatic implementation or conversion to any data structure

Issues	Description
Lack of Operational knowledge	Operational knowledge or metadata is necessary in order to achieve full automation. Meta data may include knowledge about operations, variables and logic of the program. For example two different methods may have two different sets of parameters as per their operational requirements and new implementation may require some additional different parameters for each method according to the logic of the method it is performing. Right now there is no any clear technique to collect the operational knowledge of the implementation.
Abstraction Invariant	User input is required for supplying the abstraction invariant
Language Dependency	Tool is language dependent and not capable to generate implementation other than Dafny
Limited Reusability	If the logic of the program will be change than the tool might achieve less than forty percent of genericity because of the lack of operational knowledge for new implemented logic

Validation
For Validation we are taking into account the verification of the generated code. Table below is summarizing the verification achieved so far

Verification Status	Percentage	Description
Semi Verified	50%	Based on our genericity statistics that we have achieved, the percentage of the verification code is almost fifty percent. That's mean programmers can have almost half of the code verified automatically and they can start rest of the coding for full verification of the code.

6.4.1 Related Work

In this section we analyze our work with relation to related work that we have presented in chapter 2.

New Approach
Usually data refinement is used to transform high level mathematical specifications into executable programs by changing the data types that differs from specifications.

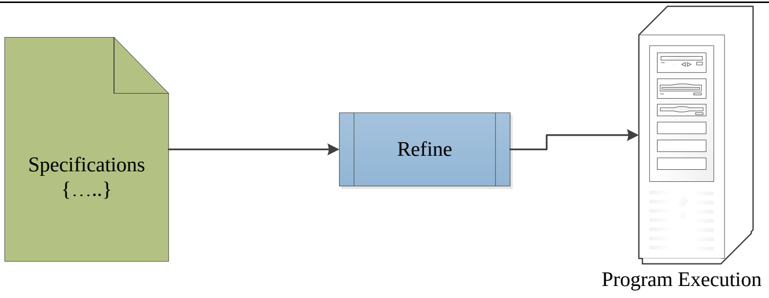


Figure 6.3 Traditional approach to refinement

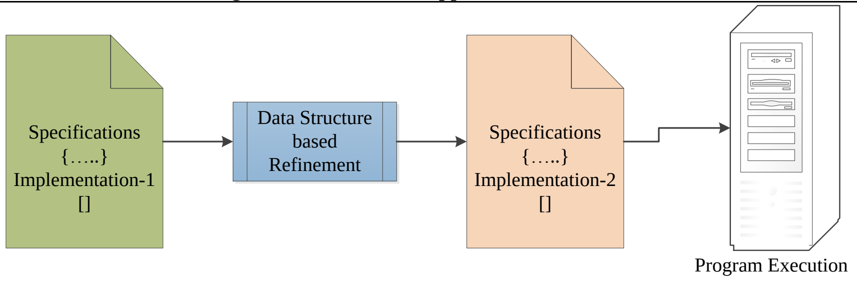


Figure 6.4 Approach based on data structure refinement

This approach is achieving the first step to support data refinement that is based on data structure where we have existing refined implementation and we want to generate another implementation that differs in data structures. The focus of this approach is on combining the data refinement and automatic program verification. Data structure based refinement approach is suitable for situations where system performance need to tunned based on data structure and when we have our specifications and design but we do not have data structure support in a tool recommended for implementation.

6.4.2 Spec# Refinement

Spec# supports both one class and two class approach to data refinement. The support for reusing specifications was achieved by using two class approach. Table below is summarizing our work with Spec# approach

Approaches	One Class	Two Class
Spec#	No support for specification reusing.	Support specification reusing but do not have support for automatic generation of new implementation. Programmer overhead is high because all abstraction invariants need to be redefined for new implementation
Specification Reuser	Support specification reusing in a automatic way using data refinement based on data structure	N/A

6.4.3 Event-B

Event-B support refinement but does not support data refinement. It supports automatic code generation based on Event-B models. Table below is summarizing our work with Event-B approach

Approaches	Refinement	Data Refinement	Code Generation
Event-B	Supports refinement of models where next model contains more detailed information about the system as compare to previous one	Not supported	Automatic code generation is supported from models but the correctness of generated code relies on the correctness of Event-B model. Also any extension to the generated code results unverified code
Specification Reuser	Support refinement based on data structure	Supported	Automatic code generation based on data structure refinement with facility to extend the code

6.5 Conclusions

In this chapter we evaluated and validated our work and provide the experimental setup. Moreover we criticize our work and analysed the threats to validity of our work. We discussed our new approach to data refinement and compare our approach to related work. In next chapter we will summarize our work and provides the future direction on the topic.

7 :-: Conclusions

In this chapter we summarize our work by visiting our problem statement again. We discuss what we have achieved so far and what are needed more to achieve our ideal system design by providing future directions.

7.1 Summary

We proposed a new technique to data refinement by focusing on data structure replacement so that we can reuse the specifications. This technique is different from the traditional technique in a way that it is focusing on data structure for refinement instead changing data types for refinement. We presented the related work where we discussed refinement in Spec# and Event-B and proposed the metric for calculating overhead based on Cost Constructive Model. We proposed the generic system design for generating multiple implementations with difference in data structure without any language and tool restrictions.

In chapter four we presented our current system design followed by implementation in chapter five. In chapter six we evaluated and validated our work based on automation and verification that we have achieved so far. We analyse the programmers overhead by comparing the results obtained from our tool with the results we have computed in chapter two by using our proposed metric. We criticized our work and discussed the validity, genericity and verifiability of our work followed by comparison with related work.

7.2 Achieving our Goals

As per our problem statement we have proposed a generic system design for data structure based data refinement. We provided the design of the system along with its class diagram. We provided the proof of concept tool for data structure based data refinement. We demonstrated our tool using a case study. Now we will provide the directions to extend the existing tool and future recommendations for developing a generic framework.

7.3 Future work

In this section we discuss future work that needed to implement our ideal system framework.

7.3.1 Library Development

The library that consists of multiple data structures which can be used in our generic framework and from where we can pick any data structure in which we want to implement our solution.

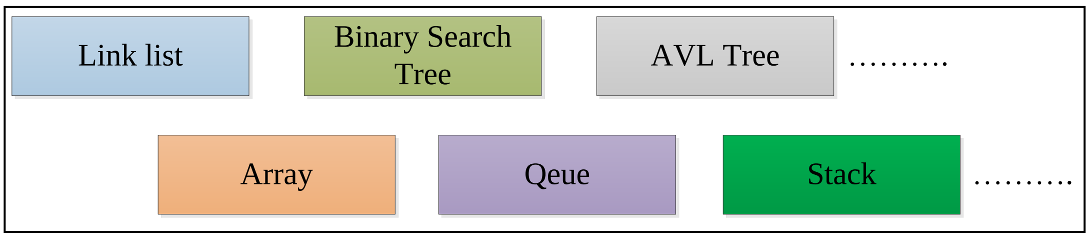


Figure 7.1: Library of data structures

Library should provide operations for every data structure. For instance if we want to implement a link list, the library should have supporting operations such as creating a node, inserting an element in the list and searching an element.

7.3.2 Language Development

Although we used Dafny language in our project as it provides greater support for data refinement but we still think that we need a more dedicated data refinement language that can treat specification, Abstraction Invariant and implementation in more easy way and that can be used as a part of our proposed system.

7.3.3 Meta Data Collection

As we discussed previously in our project that operational knowledge or Meta data is very important for providing data structure based refinement. We need to develop such a system that can provide us the operational knowledge of the system code such as methods, variables, decision statements an loops which can be used within our proposed system.

7.3.4 Improved GUI

As we presented the ideal graphical user interface for our generic framework in our solution design chapter, the new and improved graphical user interface need to develop where user can have facility to drag and drop the components and can generate implementations for their particular language.

7.3.5 Generic Framework Implementation

In order to achieve our generic system goal where we can replace any implementation for the same specifications without any underlying tool or implementation language restriction we need the development of generic framework with the components we have described above.

7.3.6 Proof obligations Reuse

In order to support data refinement from already verified program where verification required proof to be discharged we need to develop the system which can reuse proof obligations from existing implementation to new implementations.

7.3.7 Full Danfy tool Implementation

Currently our proof of concept tool is working only for refining data from sequence to array and sequence to link list with manual collection of Meta data. It can be extended to support data refinement for sets and multi-sets with automatic collection of Meta data and drag and drop facility for user interface.

In this chapter we summarized our work and provide the future directions in order to build a generic framework for data structure based refinement. In summary we have shown in this dissertation the first step towards achieving our generic refinement framework which will base on data structure change. Our future work will focus on full Dafny tool implementation followed by generic framework development.

Bibliography

[1] J. Woodcock, P. G. Larsen, J. Bicarregui and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM Computing Surveys (CSUR),* vol. 41, no. 4, pp. 1-40, 2009.

[2] K. R. M. Leino, “Automating Theorem Proving with SMT,” Microsoft Research, Redmond, WA, USA, May 2013.

[3] J.-R. Abrial, The B-book: assigning programs to meanings, Cambridge University Press New York, NY, USA ©1996 ISBN:0-521-49619-5, 1996.

[4] J. Barnes, High Integrity Software: The SPARK Approach to Safety and Security, New York: Addison Wesley, 2003.

[5] B. Meyer, “Applying Design by Contract,” *Computer,* vol. 25, no. 10, pp. 40-51, 1992.

[6] K. R. M. Leino, “This is Boogie 2,” Microsoft Research, One Microsoft Way Redmond, WA, 2008.

[7] K. Robinson, System Modelling & Design Using Event-B, University of New South Wales, 2007.

[8] R. Monahan, Data Refinement in Object-Oriented Verification, School of Computing, Dublin City Univeristy, Dublin, 2010.

[9] M. Barnett, K. R. M. Leino and W. Schulte, “The Spec# Programming System: An Overview,” in *CASSIS 2004 proceedings.*, Microsoft Research, Redmond, WA, USA, Manuscript KRML 136, 12 October 2004.

[10] D. Méry and N. K. Singh, “Automatic Code Generation from Event-B Models,” in *SoICT '11 Proceedings of the Second Symposium on Information and Communication Technology*, New York, NY, USA, 2011.

[11] N. Catano, K. R. M. Leino and V. Rivera, “The EventB2Dafny Rodin Plug-In,” in *2nd Workshop on Developing Tools as Plug-ins (TOPI)*, Zurich, 2012.

[12] K. R. M. Leino and P. Müller, “A verification methodology for model fields,” *P. Sestoft, editor, ESOP,* vol. 3924 of Lecture Notes in Computer Science, p. 115–130, 2006.

[13] J. M. Spivey, The Z Notation, Oxford: Prentice Hall International (UK) Ltd, 1992.

[14] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Abstraction and Reuse of Object-Oriented Design,” in *ECOOP '93 Proceedings of the 7th European Conference on Object-Oriented Programming*, London, 1993.

[15] OMG, Object Constraint Language (OCL), Version 2.3.1, OMG Document Number: formal/2012-01-01, 2012.

[16] M. Jastram, Rodin User's Handbook, 2012.

[17] J. Woodcock and A. Cavalcanti, "A Concurrent Language for Refinement," in *IW-FM'01 Proceedings of the 5th Irish conference on Formal Methods*, Swinton, 2001.

[18] A. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall (Pearson), 2005.

[19] Lamport, Leslie; Digital Equipment Corp., Palo Alto, CA, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 16, no. 3, pp. 872-923, 1994.

[20] ECMA, Eiffel: Analysis, Design and Programming Language, Geneva: Standard ECMA-367, 2006.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, "Extended Static Checking for Java," in *PLDI '02*, NY, 2002.

[22] A. B, Proof Obligations Reference Manual, CLEARSY System Engineerning.

[23] J. Hale, A. Parish, B. Dixon and R. K. Smith, "Enhancing the Cocomo estimation models," *Software, IEEE,* vol. 17, no. 6, pp. 45-49, 2000.

[24] C. L. Goues, K. R. M. Leino and M. Moskal, The Boogie Verification Debugger, Microsoft Research, Redmond, WA, USA, 2011.

[25] I. T. Kassios, "Dynamic Frames and Automated Verification," 2011.

[26] P. Chalin, J. R. Kiniry, G. T. Leavens and E. Poll, "Advanced Specification and Verification with JML and ESC/Java2".

[27] L. Lamport, "What Good is Temoral Logic," in *Elsevier Science Publishers B.V*, North-Holland, 1983.

[28] K. R. M. Leino and R. Monahan, "Dafny meets the Verification Benchmarks Challenge," Microsoft Research, Redmond, WA, USA.

[29] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," Microsoft Research WA, USA.

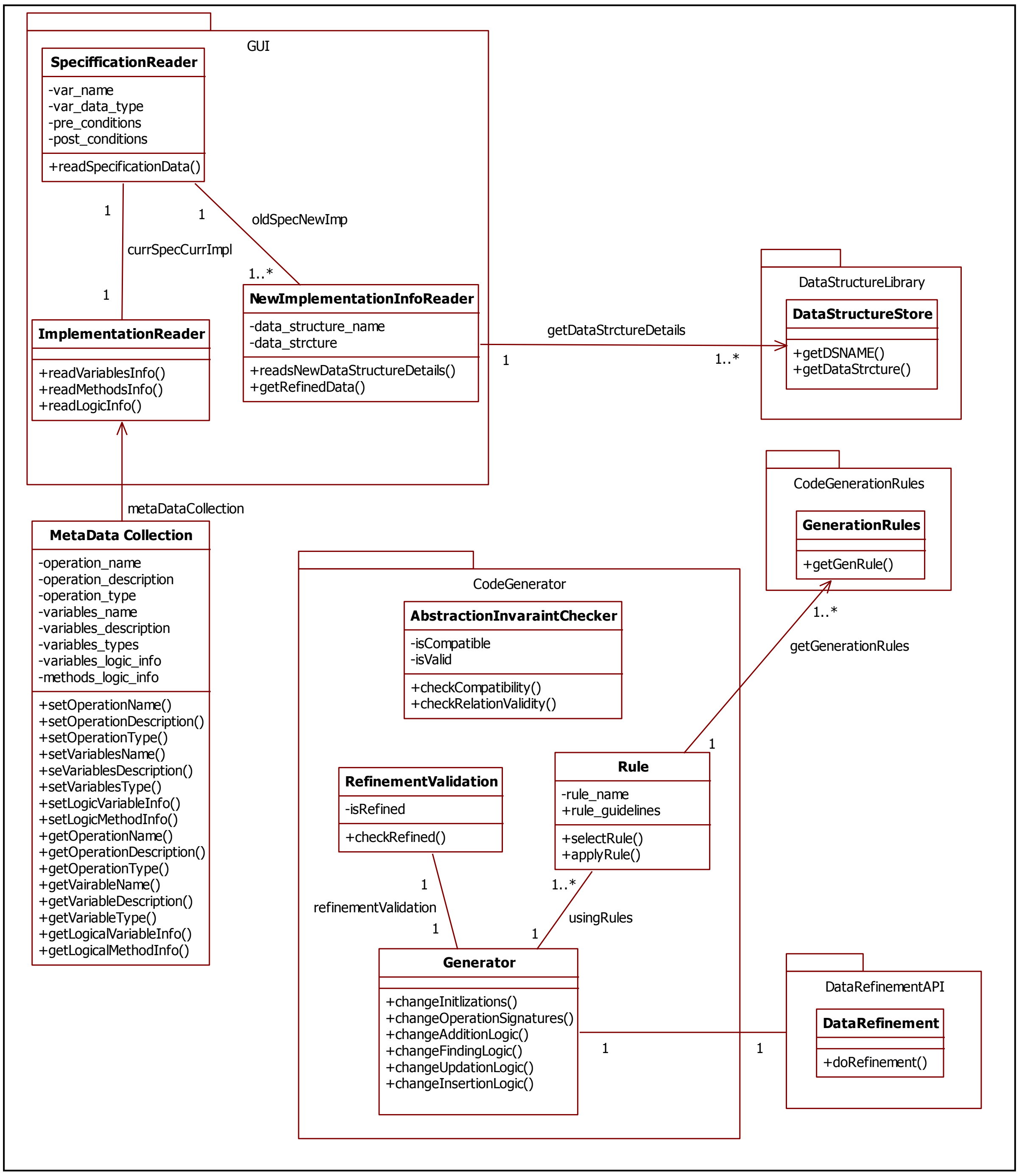
[30] L. Herbert, K. R. M. Leino and J. Quaresma, "Using Dafny, an Automatic Program Verifier," Technical University of Denmark & Microsoft Research.

[31] K. R. M. Leino, "Developing Verifi ed Programs with Dafny," Microsoft Research, Redmond, WA, USA.

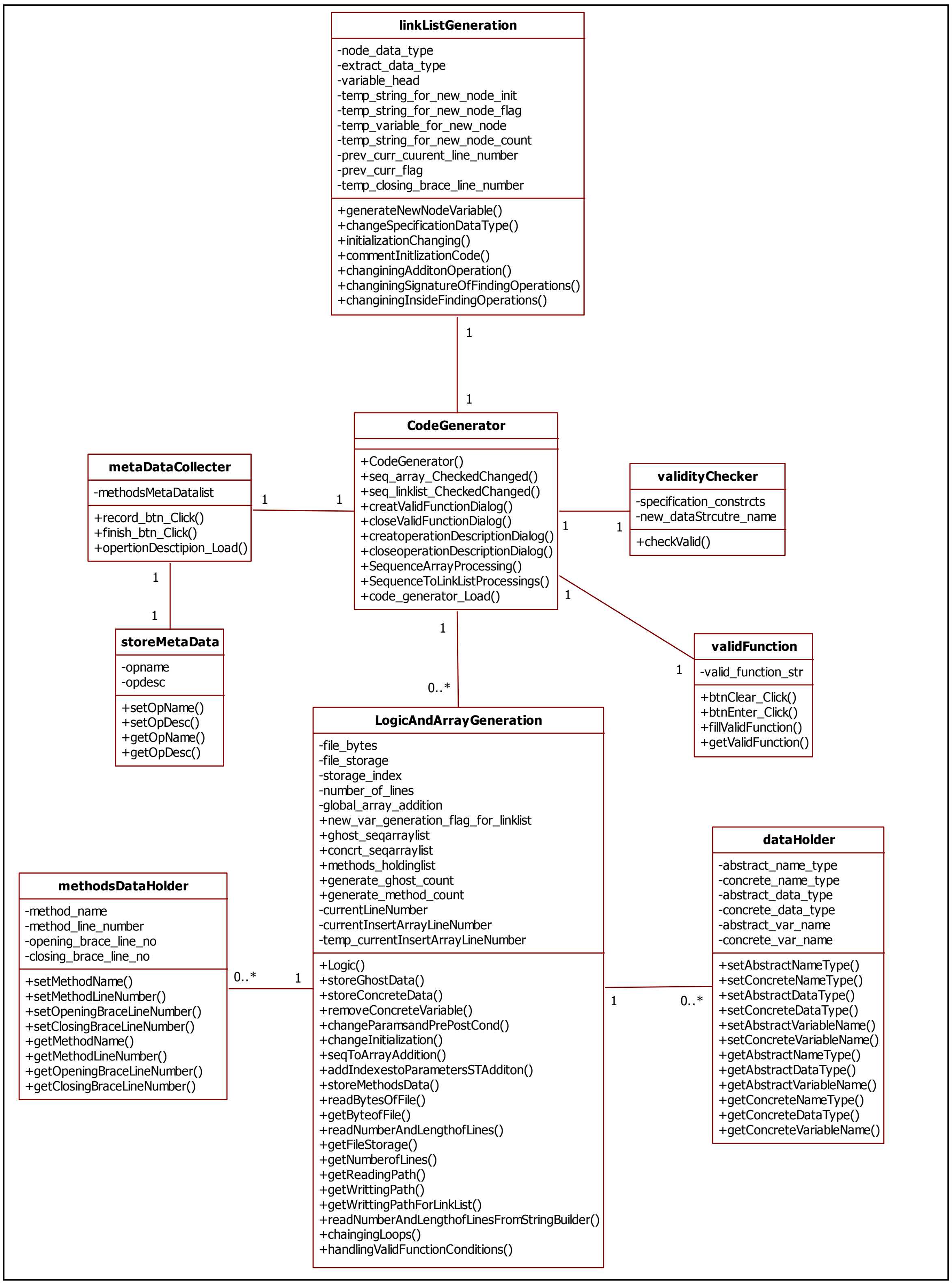
[32] K. R. M. Leino, "Specification and Verification of Object-Oriented Software," Microsoft Research, Redmond, WA, USA.

Appendix A: UML Class Diagrams

A.1: Ideal System: UML Class Diagram



A.2: Implemented System: UML Class Diagram

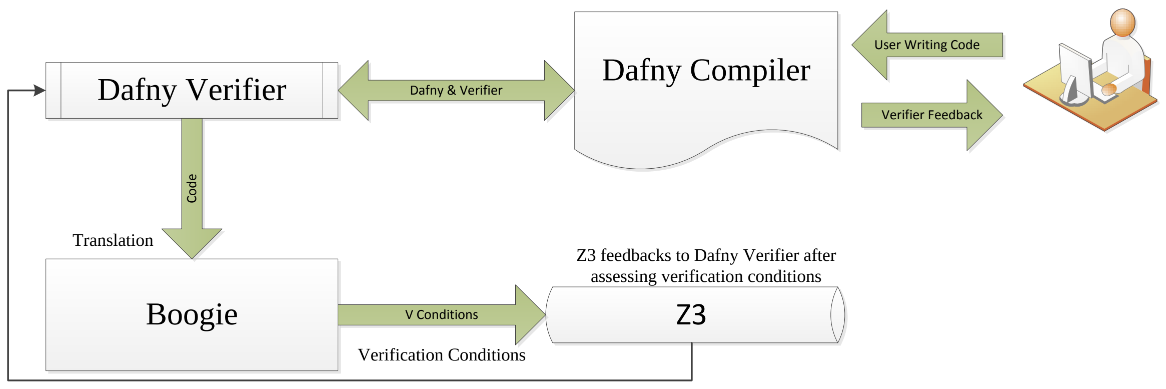


Appendix B: Dafny

Dafny Background

Dafny is an object oriented programming language designed to support static verification of the programs. The language was designed to support data refinement in fully manners and its supports generic classes, dynamic frame allocation and much useful specification constructs. Dafny allows user define algebraic data types and specification constructs which include pre and post conditions, frames handling (modifies and reads), loop invariants and termination metrics. Further in order to support the specification, language allows the updateable ghost variables and types like set and sequences. Ghost variables have a major advantage over the model variables that they are giving freedom to update them manually. Ghost specification constructs are used for the verification purpose only and the compiler does not generate the code for ghost variable for using at run time.

Dafny verifier runs as a part of the compiler same as syntactic checker and its interactive in the sense that it's always running in the background and promoting users for any failed verification. There is an integration of Dafny verifier with visual studio 2010 .By using this integration we can write the Dafny code in visual studio and verifier will run in the background constantly for checking the code. Dafny verifier translates its code to intermediate verification language Boogie [6] in such a way that the correctness of Boogie program implies the correctness of Dafny program. Boogie generates the first order verification conditions that are passed to the underlying SMT [2] solver. Any violation in verification conditions promotes back to Dafny. Sometimes it's hard to understand the error message produces by the SMT solver for which 'BVD' [24] Boogie verification debugger is available to debug the verification conditions. Below is the overall architecture of Dafny.



Dafny Architecture

Types

Dafny provides only Boolean and integer data types. The integer data type is also referred to as natural numbers "Nat". Other data types such as string and float are not supported by the language. Dafny provides user defined algebraic data types and do not allow sub classing. All the classes are subtypes of the class object. Moreover language provides the Generics facility and specification constructs such as set, sequence and multi-sets.

Pre and Post Conditions

Dafny provides the specification constructs for pre and post conditions. 'requires' clause is used for writing precondition and 'ensures' clause is used for expressing post conditions. Pre and post conditions are written immediately after the method declaration.

```
someMehtod(x: int)
requires x >=0;  //pre-condition
ensures x == 10; //post-condition
{/*method body goes here*/}
```

Methods

The methods provide the modular structure in Dafny. They are same like any other programming language but starts with the key word 'method'. The 'modifies' clause in methods provides the authority for which they can change the values of objects. Below is the general structure of the method.

```
method someMethod(x: T) returns (y: T)
requires ...       //pre-condition
modifies ...       // frame condition
ensures ...        //post-condition
{/*method body*/}
```

Where T represents the data type of x and returns indicates the return statement. One important feature that the language provides that the return variable value can be used in method code.

Functions

Functions are defined in Dafny to be used as specification constructs, where functions have the same structures as method except the return type syntax. Below is the general structure of the function

```
function someFunction(x: T): bool
requires ...       //pre-condition
reads ...     // frame condition
ensures ...        //post-condition
{/*function body*/}
```

Where 'reads' in a function indicates the frame condition. Functions are restricted to use only in specifications but if we declare the functions as "function method" then we can use it outside the specification as well.

Predicates

Predicates are same as functions but they don't have a return type. Predicates provide the behavior as boolean expressions. Below is the general structure of the predicate

```
predicate somePredicate(x:int)
requires ...       //pre-condition
{ if x > 0 then true else false}
```

Quantifiers

Quantifiers are used to iterate over arrays, sequences and sets. 'forall' is the only quantifier that is available in Dafny right now. Below is the general structure of the quantifier

```
forall k :: 0=< k < array.Length ==> …array[k]…;
```

Where 'forall' is the key word and statement can be interpreted as 'forall k such that k is greater than 0 and less than array.Length implies that check every element in the array on index k'. Quantifiers are very useful for writing pre and post conditions.

Sets

Sets are very useful specification constructs and a powerful mathematical property to express specifications. Sets are order less collection of elements which are distinct and no repetition is allowed in the set. Sets are immutable in Dafny that's mean they cannot be modified once created hence they can be used easily in annotations without involving heap. Dafny has 'set' key word for declaring sets.

```
var s1 : set<int>;   var s2 : set<int>;
s1:={1,2,3} , s2:={2,4}
assert s1 + s2 == {1,2,3,4} //set union
```

```
assert s1 * s2 == {2} //set intersection
assert s1 – s2 == {1,3} //set difference
```

Where s1 and s2 are the sets of integers and assert statement is ensuring the equality of the different operations such as union, difference and intersection. Sets are very useful for verification of the programs.

Sequences

Sequences are immutable objects and same as their counterpart set. Sequences contain ordered elements and without restriction to be distinct. Sequences have different operations then sets such as sets required union for adding an element whereas a sequence has concatenation for adding an element into sequence. Dafny has 'seq' keyword for declaring the sequence.

```
var s : seq<int>;
s := [1,2,3,4,5];
assert s[|s|-1] == 5;
assert s[..] == [1,2,3,4,5];
```

```
assert s[1..] == [2,3,4,5];
assert s[1,2,3,4,5] == s[1,2,3] + s[4,5];
assert forall k :: 0=< k < |s| ==> s == s[..k] + s[k..]
```

Where 's' is a sequence of integers in first statement. Second statement is describing the concept of slicing where we can slice the sequence and extract particular element at any given index. Concatenation operation is associative in sequence. The last assert statement is indicating the iteration over the sequence and holds the relationship that both sides are equal after implication.

Loop invariants

Loop invariants are very important for program verification. Dafny has no mechanism to determine in advance about the number of time loop will execute in cases where no bounds exist for the data structure such as link list. It is deterministic in case of iterating over arrays and sequences but un-deterministic in other cases. Loop invariants are used in order to guide the verifier about the current situation of the loop. Loop invariant should hold before entering the loop, during execution and after termination of the loop. Dafny has 'invariant' as a key word for expressing invariants in the loop. Following is an example of an invariant.

```
var i: = 0;
var n : = 10;
```

```
While ( i < n)
Invariant i >= 0;  //invariant remains true before, during and after execution of the loop
{i : = i +1 }
assert i == n;  //condition after loop termination
```

Termination

As termination is deterministic in case of arrays and sequences but un-deterministic in other cases such as link lists we have to provide some termination information about the loop otherwise verifier will complain about the loop that he cannot terminate it. Dafny has key word 'decreases' for handling loop termination. It is good practice to provide decreases clauses in the deterministic case as well because it helps verifier to verify the program easily.

```
while ( i < array.Length )
decreases array.Length - i;    //length will decrease on every iteration
{i : = i +1}
assert i == array.Length;  //condition after loop termination
```

Ghost Variable

Ghost variables are specification only variables that are needed only for verification purpose. The compiler does not generate code for ghost variables as these are not needed at the execution time. Ghost variable is same like physical variables and can be used inside the program freely except that the values of ghost variables cannot be floated into physical variables.

```
ghost var gv :  int;
ghost var gsq :  seq<int>;
ghost var gst :  set<int>;
```

Where 'ghost' is the key word used to declare the variables. Here 'gv' is a ghost variable of type integer and variables "gsq" and "gst" are ghost sequence and set.

Frame Handling

Frame problem can be stated as: 'when formally describing a change in a system, how do we specify what parts of the state of the system are not affected by the change' [25]. Frame conditions are used in methods and functions specifications which tell the verifier about the data or object that can be modified during the execution of the method or function. Methods are using 'modifies' clause where function are using "reads" clause to express frame conditions. Dafny handles these conditions in a very simple way by providing the set of objects that methods can modify and the function can read.

```
ghost var Repr :  set<object>
method someMethod1()
modifies this
{ /*method body */}
```

```
method someMethod2()
modifies Repr;
{/*method body*/}
```

'Repr' is a set of objects and declared as a ghost because this set do not require at run time. It contains all objects that method can modify such as this, obj1 and obj2 where obj1 and obj2 are some objects of the class. This technique is called dynamic framing in Dafny. Set is dynamic because it contains different objects at different states of the system.

Abstraction Function

Abstraction function is a function that relates specification data with concrete data. In Dafny this relation is defined in a method called 'Valid'. This is not reserved name of the function instead we can give any name to this method only just for simplicity we refers it to as Validity function. This method is used as pre and post condition to most of the methods and as a precondition to functions for marinating system consistency.

Dynamic Frames

Dafny handles frames problem with a dynamic frame technique where it uses a set of objects which contains different objects at different times that are available to method to modify. Below is an example of handling frames.

```
class handle_frames{
ghost var repr : set<object>

function Valid()
reads this, repr;
{this in repr && /*function body*/}

method Init()
modifies this;
ensures Valid() && fresh(repr – {this});
{/*body of the method*/
 repr := {this} + {any new object}}

method Update()
requires Valid();
modifies repr;
ensures Valid() && fresh(repr – {this});
{/*body of the method*/}
}

var obj = new handle_frames();  obj.Init();
```

Above class declares a 'representation' set named 'repr'' as a ghost variable. 'Valid' functions say that it allows reading of objects in the 'repr' along with 'this' object in order to maintain the consistent state. 'Init' specifies that he can modifies 'this' object and ensures that after execution of the method system remains in consistent state and also ensures that all newly allocated objects have been added to repr set except 'this' object. Keyword 'fresh' is used by Dafny to express these allocations where fresh(s) means that all non-null objects in set 's' will be allocated after execution of the method.

Update method showing that the system will remains in consistent states before and after method execution. Since this method is allowed to modify 'repr' its post condition is ensuring the consistency of the system and updating all the objects that have been modified during the execution of the method in 'repr' set. Now if we make an object of the class as shown in the example the 'obj.repr' is disjoint from any other object of the system and it has its own frame to operate.