Dynamic Frames and Automated Verification

IOANNIS T. KASSIOS
DEPT. INFORMATICS - ETH ZURICH

SECOND COST ACTION IC0701 TRAINING SCHOOL
LIMERICK, 21/6/2011

Scope

- The small history of Dynamic Frames
  - Formalism to deal with the Frame Problem, in modular programs with rich heap data structures and encapsulation
  - Started off as theoretical work (presenter's PhD thesis – 2006)
  - Automation proved to be a challenge
    - mainly due to the higher order nature of the original theory
- In this tutorial, we show:
  - The problem
  - The original solution and its key points
  - Two major variants for the automation of the original theory
    - major idea, dealing with key points, advantages and shortcomings

Purpose

- This is interesting for…
  - Researchers in the area of Dynamic Frames and the related technologies
    - prospective contributors to the theory, its automation, or one of its implementations
  - Researchers in the area of Framing
    - who may be working on, or looking for "competing" methodologies
  - Prospective "clients" of the presented methodologies
    - researchers in the area of Specification and Verification, especially tool builders
    - users of Dafny, Regional Logic, VeriCool, Chalice, …
  - People interested in the design of formal theories
    - still researchers ☺

Overview

- Basics of Modular Specification
  - + specialisation for imperative programs with rich heap structures
- Dynamic Frames
- Automated Verification
- Automating Dynamic Frames
  - Ghost Fields for Dynamic Frames: the Dafny Solution
  - Explicit Folding and Unfolding: the Implicit Dynamic Frames Solution
  - Further Thoughts on the Verification of Dynamic Frames
- Pointers to Related Work
- Conclusion

Basics of Modular Specification

Modular Software Engineering

- *Modularisation*: Breaking a programming task into small manageable ones
- Isolation of errors and modifications
- Ideal: changes and corrections in one module should not affect other modules
  - (in practice: “should have small effect”)
- Almost every programming language has one or more modularisation construct(s)

Our Focus

- We are interested in *Imperative Programming with Heaps*
  - *Convention*: we choose "class" to be our module
    - but we stay "object-based"
- We are interested in *Full Functional Verification*
  - "Full" is a bit of a misnomer: we want to deal with as many desired properties of the program as possible
    - however, we stay "partially correct"
  - We want to specify the desired properties in an unambiguous, concise, and totally formal mathematical language
  - We want to prove compliance formally

Modular S&V: The Rules

- Each module *C* has a specification *S*
- The proof that *C* complies to *S* is *independent* of any other proof
  - change of implementation → no other proof is affected
  - change of environment → verification of *C* is not affected
- A client *C'* of *C* must rely only on *S* in its proof of correctness

Procedural Abstraction


```
// correct class

class C
{
  var x:int;

  method i()
   ensures x>old(x);
  { x:=x+1; }
}
```

```
// correct client
method m0(c:C)
  ensures c.x>2*old(c.x);
{ c.x:=2*c.x; c.i(); }


// incorrect client
method m1(c:C)
  ensures c.x=2*old(c.x)+1;
{ c.x:=2*c.x; c.i(); }
```

Data Abstraction

- The implementer wishes to hide the *data representation* from the client
- The client is not allowed to refer to the implementation fields anymore
- There should be an abstract vocabulary that both the client and the implementer can understand
  - the client reasons using the abstract vocabulary
  - the implementer connects the abstract vocabulary to the concrete fields
- Here: abstract vocabulary = *pure functions*

The List Example - I


```
class Node
{ var x:int; var n:Node; }

class List
{
  var c:Node;

  function len():int
  { len_aux(c) }

  function len_aux(p:Node):int
  { p=null ? 0 : 1+len_aux(p.n) }
```

The List Example - II


```
function get(i:int):int
  requires 0≤i<len();
{ get_aux(i,c) }

function get_aux(i:int,p:Node):int
  requires 0≤i<len_aux(p);
{ p=null ? 0 : 1+get_aux(i-1,p.n) }
```

The List Example - III


```
  List()
    ensures len()=0;
  { c:=null; }

  method insert(x:int)
    ensures len()=old(len())+1 ∧ get(0)=x;
    ensures ∀i:1..len()-1·get(i)=old(get(i-1));
  {
    var p:Node; p:=new Node;
    p.v:=x; p.n:=c; c:=p;
  }
}
```

Inconsistency in Recursive Definitions

- Pure functions are not so pure…

```
function f():bool { ¬f() }
```

- In our example, `len()` is inconsistent for cyclic lists
- We must always provide a measure of termination for pure functions used in specifications
  - but here, it is not clear how to bound `len()`
- We revisit this issue later

Dynamic Frames

The Frame Problem-I


```
// failing client
var A,B:List;
A:=new List();
  // here: A.len()=0
B:=new List();
  // here: B.len()=0
  // A.len=0
assert A.len()=0;  // fails
```

The Frame Problem – II

- The specification of `List()` ensures that `B.len()=0` but makes no other promise about the state!
  - e.g., the specification allows `List()` to change `A.c`
- Frame Problem: how do we specify *non-change*?
- Modularity in the way
  - "no client variable is affected": the specifier does not know the variables of the client
  - "only `this.c` is affected": the client does not know `this.c`
  - reasoning in the abstract level: "only the pure functions of `this` are affected": unsound!
    - "abstract aliasing"

Footprints and Dynamic Frames - I

- The *footprint* of a method invocation is the set of concrete fields that the invocation modifies
- The *footprint* of a function evaluation is the set the set of concrete fields that the returned value depends on

- Main idea of the DF theory: *make footprints part of the abstract vocabulary*
- Abstract aliasing = disjointness of footprints
  - abstractly expressible!

Footprints and Dynamic Frames - II

- Region = set of fields
- Footprints of methods
  - `modifies` *F*;
  - *F* is region-valued
  - *F* is evaluated in the pre-state
  - allocation of new state is allowed
- Footprints of functions
  - `reads` *F*;
- Dynamic frame = pure function of type **`reg`** (region)

Annotating Lists with Dynamic Frames - I


```
class List
{
  var c:Node;

  // representation region
  function rep():reg
  { {this.c} ∪ rep_aux(c) }

  function rep_aux(p:Node):reg
  { p=null ? ∅ : {p.v,p.n} ∪ rep_aux(p.n) }
```

Annotating Lists with Dynamic Frames - II


```
function len():int
  reads rep(); …
function len_aux(p:Node):int
  reads rep_aux(p); …
function get(i:int):int
  reads rep(); …
function get_aux(i:int,p:Node):int
  reads rep_aux(p); …
```

Annotating Lists with Dynamic Frames - III


```
  List()
    …
    modifies ∅; … // can be omitted

  method insert(x:int)
    …
    modifies rep(); …
}
```

New Members: Acyclicity Invariant

- An invariant that ensures acyclicity

```
function inv():bool
  reads rep();
{ inv_aux(c) }
function inv_aux(p:Node):bool
  reads rep_aux(p);
{ p=null ∨
    ({p.v,p.n}∩rep_aux(p.n)=∅ ∧ inv_aux(p.n)) }
```

- We conjoin the invariant as a precondition to all methods and functions (except the constructor)
- We conjoin the invariant as a postcondition to all methods

New Members: A Prepending Method


```
method prepend(p:List)
  requires p≠null ∧ rep()∩p.rep() = ∅ ∧ inv() ∧ p.inv();
  modifies rep() ∪ p.rep();
  ensures len()=old(len()+p.len());
  ensures ∀i:0..old(p.len())-1·get(i)=old(p.get(i));
  ensures ∀i:old(p.len())..len()-1·
                                  get(i)=old(get(i-p.len()));
  ensures inv();
{
  var q:Node;
  if (p.c≠null)
  {
    q:=p.c; while (q.n ≠ null) { q:=q.n; }
    q.n:=c; c:=p.c;
  }
}
```

Preserving Disjointness


```
// failing client

method client(A,B:List)
 requires A.rep() ∩ B.rep() = ∅;
 requires B.len()=1;
{
  // here: A.rep() ∩ B.rep() = ∅
  A.insert(10);
  assert B.len()=1;  // succeeds
  // A.rep() ∩ B.rep() = ∅
  A.insert(10);
  assert B.len()=1;  // fails!
  // how does A.insert(10) change A.rep()?
  // what about B.rep()?
}
```

Swinging Pivots-I

- Let *S* be the set of all the dynamic frames that are included in the footprint of a method *m*(..)
- The invocation of *m*(..) should allow any dynamic frame in *S* to be increased only with:
  - initially unallocated locations
  - locations that belong initially to a dynamic frame in *S*
- Policy ensures that the dynamic frames do not “step on unknown territory”

Swinging Pivots-II

- The two-state specification

  fresh E

means "every location in E is allocated in the post-state and unallocated in the pre-state"

- Swinging pivots for dynamic frames in S

  fresh $(\cup_{f \in S} f()$-**old**$(\cup_{f \in S} f()))$

Swinging Pivots for the List Example


```
class List
{
  ...
  List()
    ...
    ensures fresh(rep()); ...

  method insert(x:int)
    ...
    ensures fresh(rep()-old(rep())); ...

  method prepend(p:Node)
    ...
    ensures fresh(rep()-old(rep() ∪ p.rep())) ; ...
}
```

Preserving Disjointness (revisited)


```
// failing client

method client(A,B:List)
 requires A.rep() ∩ B.rep() = ∅;
 requires B.len()=1;
{
  // here: A.rep() ∩ B.rep() = ∅
  A.insert(10);
  assert B.len()=1;  // succeeds
  // here: A.rep() ∩ old(B.rep()) = ∅
  // A.rep() ∩ B.rep() = ∅
  A.insert(10);
  assert B.len()=1;  // fails!
  // how does A.insert(10) change B.rep()?
}
```

Self-framing

- If an operation does not touch fields within a dynamic frame, that dynamic frame *should not change its value*

- All dynamic frames should be *self-framing*

Self-framing for the List Example


```
function rep():reg
  reads rep();
{ {this.c} ∪ rep_aux(c) }

function rep_aux(p:Node):reg
  reads rep_aux(p);
{ p=null ? ∅ : {p.v,p.n} ∪ rep_aux(p.n) }
```

Preserving Disjointness (final)


```
// the client now works!

method client(A,B:List)
 requires A.rep() ∩ B.rep() = ∅;
 requires B.len()=1;
{
  // here: A.rep() ∩ B.rep() = ∅
  A.insert(10);
  assert B.len()=1;  // succeeds
  // A.rep() ∩ old(B.rep())=∅ ∧ B.rep()=old(B.rep())
  // A.rep() ∩ B.rep() = ∅
  A.insert(10);
  assert B.len()=1;  // succeeds
}
```

Disjointness in a more complex Client


```
var A,B,C:List;
A:=new List(); B:=new List();
C:=new List();

A.insert(1); B.insert(2); C.insert(3);
A.insert(4); B.insert(5); C.insert(6);
A.prepend(B);

assert A=[5,2,4,1] ∧ C=[6,3];
assert A.rep() ∩ C.rep() = ∅;
```

Automated Verification

Two Popular Verification Methodologies

- Verification Condition Generation (VCG)
  - Use a program calculus (such as a wp-calculus) to construct a big logical formula (*verification condition*) out of a module and its specification
  - Then feed the verification condition to a *prover*
  - Popular VCG tools: Boogie, Why
- Symbolic Execution (SE)
  - Execute all possible branches with *symbolic* values gathering logical information in a *path condition*
  - From time to time direct questions to a *prover* providing the path condition as an axiom
  - Popular SE tools: KeY, Smallfoot, VeriFast

SMT Solvers

- Common denominator of verification methodologies: the back-end prover
- State-of-the-art automation: SMT (*Satisfiability Modulo Theories*) solvers
  - Popular Tools: Z3, AltErgo, Simplify
- First-order logic extended with custom decision procedures for theories
  - example theory: linear arithmetic

Good Practices for SMT Solvers

- Restrict the specification language to first order logic
- Avoid quantifications (as much as possible)
- Avoid *matching loops*
  - Dangerous axiom:

    $\forall\, x \cdot f(x) = g(f(x))$ (A)
  - Attempt to prove $f(3)$

    instantiate (A) for $x=3$ → attempt to prove $g(f(3))$

    → instantiate (A) for $x=f(3)$ → attempt to prove $g(g(f(3)))$

    → …
  - (!) caused frequently by common recursive definitions

Automating Dynamic Frames

Verification of DF: the Problem

- Recursive definitions have many solutions
  - In reality, we want `rep()` to be the *least fix-point* of its definition
    - incidentally, we have the same issue with `len()`
- Least fixed-points (the induction principle) are *not expressible* in first order logic (eqv. *reachability* predicates)
- The need for least fixed points (reachability) is not a peculiarity of the list example
  - Interesting dynamic frames are non-trivial under-approximations of reachability
- How can we use SMT solvers to automate our framework?

Approaches

- Use of *ghost fields* to store the values of dynamic frames
  - Dafny, Regional Logic,…
- Use of explicit *folding/unfolding* of recursive definitions
  - Implicit Dynamic Frames: VeriCool, Chalice

Automating Dynamic Frames: Ghost Fields

Ghost Fields

- Ghost fields are “imaginary” fields
  - they don’t appear in the “real” program, e.g., the compiled version
  - they are used for specification purposes
  - they can be assigned to
  - they cannot influence the control flow of the program
- Use of a ghost field to store a value of interest
  - desired properties, incl. recursive definition appear in the invariant
  - the value must be updated explicitly by the programmer

The Node Class in Dafny


```
// rep and len are now ghost fields (in both List and Node)

class Node
{
  var v:int; var n:Node;
  ghost var rep:set<object>, len:int;

  function inv():bool
    reads this,rep;
    decreases len;
  {
      this∈rep ∧ null∉rep ∧ len>0
    ∧ (n=null ⇒ rep={this} ∧ len=1)
    ∧ (n≠null ⇒
         n∈rep ∧ n.rep⊆rep ∧ this∉rep ∧ len=1+n.len ∧ n.inv())
  }
}
```

The List Class in Dafny-I


```
class List
{
  var c:Node;
  ghost var rep:set<object>, len:int;

  function inv():bool ... // same as in the Node class

  function get(i:int):int
    requires inv() ∧ 0≤i<len;
    reads rep;
    decreases i+1;
  { get_aux(i,c) }

 function get_aux(i:int,p:Node):int
    requires p≠null ∧ p.inv() ∧ 0≤i<p.len;
    reads p.rep;
    decreases i;
  { if i=0 then p.v else get_aux(i-1,p.n) }
```

The List Class in Dafny-II


```
method Init()
 modifies this;
 ensures len=0 ∧ fresh(rep-{this}) ∧ inv();
{ c:=null; len:=0; rep:={this}; }

method insert(x:int) //omitting get-related specifications
  requires inv();
  modifies rep;
  ensures inv() ∧ len=old(len)+1  ∧ fresh(rep-old(rep));
{
  var p:Node;

  p:=new Node; p.v:=x; p.n:=c; c:=p;

  len:=len+1; c.len:=len; rep:=rep∪{p};
  if(c.n=null) { c.rep:={c}; } else { c.rep:={c} ∪ c.n.rep; }
}
```

The List Class in Dafny-III


```
  method prepend(p:List)
    requires p≠null ∧ rep∩p.rep = ∅  ∧ inv() ∧ p.inv();
    modifies rep ∪ p.rep;
    ensures inv() ∧ len=old(len+p.len) ∧ fresh(rep-old(rep ∪ p.rep));
  {
    if(p.c ≠ null)
    {
      call prepend_aux(p.c); c:=p.c; len:=len+p.len; rep:={this}∪ p.c.rep;
    }
  }

  method prepend_aux(q:Node)
    requires q ≠ null ∧ rep∩q.rep = ∅ ∧ inv() ∧ q.inv();
    modifies q.rep;
    ensures q.inv() ∧ q.len=old(len+q.len);
    ensures old(q.rep) ∪ rep-{this} = q.rep;
  {
    q.rep:=q.rep ∪ rep-{this}; q.len:=q.len+len;
    if(q.n ≠ null) { call prepend_aux(q.n); } else { q.n:=c; }
  }
}
```

Dafny – Technicalities

- Definedness of recursively defined functions is ensured by keyword **`decreases`**
- Self-framing: *not required*!
  - dynamic frames have no definition
  - impose swinging pivots
  - irrelevant frames do not change values – they must be updated manually
- Dealing with matching loops: “limited functions”
  - Boogie-level technique
  - disallows the use of a recursive definition more than once

Dafny: Discussion

- Very flexible – very effective
  - especially with Rustan Leino in front of the screen
  - ghost variables provide very valuable guidance to the prover
- Verbose Specifications
  - problem inherited from Dynamic Frames
- Ghost assignments are hard
  - may introduce bugs themselves
  - they are frequently forgotten
  - they are an annotation burden

Automating Dynamic Frames: Explicit (Un)folding

Explicit (Un)folding of Recursions

- Suppose that we have a recursive definition

  $x=f(x)$

- We do not allow the prover to use this axiom, unless the user explicitly directs us to do so:
  - unfold x: allows the axiom to be used from left to right
  - fold x: allows the axiom to be used from right to left
  - no matching loops!
- Suppose these are boolean types

  x is proved

  unfold $x \rightarrow f(x)$ is proved

  unfold $x \rightarrow f(f(x))$ is proved…
- Effectively, we have induction / reachability

Linear Logic and (Un)folding

- Treat expressions as *resources*

  x is *held*

  unfold $x \rightarrow x$ is *given up*, $f(x)$ is *acquired*

  fold $x \rightarrow f(x)$ is *given up*, x is *acquired*

- The "frame rule"

  $A \wedge x$ is held

  unfold $x \rightarrow A \wedge f(x)$ is held

  fold $x \rightarrow A \wedge x$ is held

Implicit Dynamic Frames

- The *accessibility predicate* `acc(x.f)`
  - = the field `f` of object `x` is in the footprint
  - no modifies clause: it all goes to the precondition

```
requires acc(this.x) ∧ acc(this.y);
```

- There is only one resource `acc(x.f)` per field `x.f`

```
acc(x.f) ∧ acc(x.f)  ⇔  false
```

- Conjunction is now *separating*
  - non-separating conjunction is not expressible

Abstract Predicates

- Abstract predicate = pure function of type boolean, that possibly contains accessibility predicates
  - potentially recursive definitions
- Typical example: the list invariant

```
predicate inv
{ acc(this.v) ∧ acc(this.n) ∧
    (this.n≠null⇒this.n.inv)
}
```

  - offers access to all nodes and values stored (but access must be justified with explicit unfolds)
  - succinct heap separation: this is an acyclic list

The Node Class in Chalice


```
class Node
{
  var v:int; var n:Node;

  predicate inv
  { acc(v) ∧ acc(n) ∧ (n≠null⇒n.inv) }

  function len()
    requires inv;
  { unfolding inv in n≠null ? n.len()+1 : 1 }
}
```

The List Class in Chalice-I


```
class List // get functionality omitted
{
  var c:Node;

  predicate inv
  { acc(c) ∧ (c≠null⇒c.inv) }

  function len()
    requires inv;
  { unfolding inv in c≠null ? c.len() : 0 }
```

The List Class in Chalice-II


```
List()
  ensures inv ∧ len()=1;
{ c:=null; fold inv; }

method insert(x:int)
  requires inv;
  ensures inv ∧len()=old(len())+1;
{
  var p:Node; p:=new Node;
  unfold inv; p.v:=x; p.n:=c;
  fold p.inv; fold inv;
}
```

The List Class in Chalice-III


```
 method prepend(p:List)
    requires p≠null ∧ inv ∧ p.inv;
    ensures inv;
  {
    unfold p.inv;
    if(p.c ≠ null)
    { unfold inv; call prepend_aux(p.c); c:=p.c; fold inv; }
  }

  method prepend_aux(q:Node)
    requires q ≠ null ∧ acc(c) ∧ (c≠null⇒c.inv) ∧ q.inv;
    ensures q.inv ∧ acc(c);
  {
    unfold q.inv;
    if(q.n ≠ null) { call prepend_aux(q.n); } else { q.n:=c; }
    fold q.inv;
  }
}
```

IDF – Technicalities-I

- Definedness of functions
  - besides various "regular" termination measures, unfolding also provides a definedness argument: in any state only finitely many folds may have happened, so the following is consistent:

```
function len()
  requires inv;
{ unfolding inv in n≠null ? n.len()+1 : 1 }
```

- Self-framing: *all predicates and specifications must contain access to the fields they are referring to*
  - self-framing: `acc(x) ∧ x=10` not self-framing: `x=10`
  - self-framing: `inv ∧ len()>0` not self-framing: `len()>10`
  - checking self-framing is syntactic!

IDF – Technicalities-II

- Swinging Pivots: always true!
  - an accessibility resource can be included in a predicate, only if the method has access to the resource
- Dealing with matching loops
  - not automatically solved: predicates do not introduce them, but recursive functions do
  - solution depends on the tool. Limited Functions are applied to Chalice-VCG
  - same idea as fold/unfold can be applied to functions, but this can get too much for the programmer

IDF: Discussion – I

- Very concise notation
  - close to Separation Logic
- Many things "happen" with minimal effort
  - e.g., self-framing, swinging pivots
- Linear logic is not that intuitive
  - specifications have side-effects to the ghost state
  - not trivial to get accessibility specifications right

IDF: Discussion – II

- Explicit folds/unfolds
  - sometimes too much pain
  - implicit tool-generated folds/unfolds: not recommended
- Less expressive than Dafny (?)
  - no non-separating conjunction
  - expressiveness close to automated parts of Separation Logic
- Less powerful than Dafny (?)
  - according to the presenter's experience
  - also: higher level

Automating Dynamic Frames: Some Further Notes

Call for More Approaches

- Maybe bite the bullet and use interactive higher order provers?
- Use of special reachability theories in SMT solvers?
- Use in conjunction with shape analysis (e.g. TVLA)?

Related Work

Related Work-I

- Problem known since at least 1998 (technical report by Leino and Nelson). Partial solution introduced in (Leino, Nelson TOPLAS 2002)

- Approaches based on Ownership (Clarke 2001) are used for the framing problem
  - For *full functional verification*: Universes (Müller 2002) evolved into Spec#
  - VCC also works with Ownership

Related Work-II

- Separation Logic (Reynolds LICS 2002) augmented with Abstract Predicates (Parkinson, Bierman POPL 2005) is mainstream
  - Very expressive specification language
  - Automated versions are based on Symbolic Execution (King 1976) and deal with a subset of the language – typically without non-separating conjunction
  - Relationship to IDF explored by (Parkinson, Summers ESOP 2010) – results premature
  - Automated tools: Smallfoot, VeriFast, jStar, …

Related Work - III

- More automated tools for DF
  - Naumann's Regional Logic (VCG)
  - Extension to the KeY system by Weiß (SE)
- Implicit Dynamic Frames
  - Introduced in (Smans, Jacobs, Piessens ECOOP 2009)
  - Tools: VeriCool and Chalice (Leino, Müller 2010)
  - Chalice extends IDF with Fractional Permissions (Boyland SA 2003), to deal with Concurrency (deadlock / data race freedom)
  - Initial Version of Chalice is VCG. A SE tool called Syxc has been recently developed (Schwerhoff 2011)

Conclusion