DynAlloy: Upgrading Alloy with Actions

Nazareno Aguirre

Departamento de Computación

Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Universidad Nacional de Río Cuarto

`naguirre(at)dc.exa.unrc.edu.ar`

(joint work with M. Frias, J. P. Galeotti and C. López Pombo)

1 The Alloy Specification Language

- Simple formal semantics, based on relations
- Simple syntax, with few constructs with intuitive meaning
- OO-like style of specifications
- Designed with fully automated analysis in mind
- (Almost) no built in types
- Specifications are written by defining data domains and operations between these domains

2 Alloy’s Features

- Suitable for the specification of static structural properties
- Specifications are automatically analysable, via SAT solving based mechanisms
  - Alloy’s underlying logic is not decidable (proper extension of FOL), so the analysis is a validation mechanism rather than a verification procedure
  - Alloy counts on a tool for validating properties of specifications, the Alloy Analyzer
- Not appropriate for the specification of *dynamic* properties of systems, i.e., properties regarding executions

3 Alloy's Analysis Mechanism

Alloy's analysis mechanism is essentially a counterexample extraction procedure based on SAT solving:

> Given a system specification and a statement about it, a counterexample of the statement (under the assumptions of the system specification) is exhaustively searched for.

BUT

> Alloy's logic is a proper extension of FOL

So, this search is **exhaustively searched for up to a bound** k on the cardinality of the domains of interpretations.

According to D. Jackson, this is useful in practice due to the fact that, if a statement is false, then it usually has a counterexample of small size.

4 Alloy's Syntax

Alloy's syntax has only a few constructs:

- *signatures*, which can be used to denote data domains and modules
- *predicates*, which can be used to characterise special states or operations of modules
- *facts*, which represent the properties of a specification (i.e., *axioms*)
- *assertions*, which represent the *intended* properties of a specifications, i.e., those to be validated.

5 An Example

Let us consider a specification of memories with cache. We can use basic signatures denote data domains:

```
sig Addr { }            sig Data { }
```

Also, more complex signatures can be used to characterise modules:

```
sig Memory {
    addrs: set Addr
    map: addrs ->! Data
}
```

`Memory` denotes a set of atoms, `addrs` a subset of $\texttt{Memory} \times \mathcal{P}(\texttt{Addr})$ and `map` a subset of $\texttt{Memory} \times \texttt{Data}^{\texttt{addrs}}$

Signatures can be specialised via signature extension:

```
sig MainMemory extends Memory { }

sig Cache extends Memory {
    dirty: set addrs
}
```

and more complex signatures can be defined in terms of simpler ones:

```
sig System {
    cache: Cache
    main: MainMemory
}
```

5.1 Alloy’s Predicates

Predicates can be used to characterise special states of a system:

```
pred DirtyInv(s: System) {
    all a: s.cache.addrs - s.cache.dirty |
        s.cache.map[a] = s.main.map[a]
}
```

as well as operations of the modules of a specification (in the style of Z):

```
pred Write(m, m’: Memory, d: Data, a: Addr) {
    m’.map = m.map ++ (a -> d)
}
```

```
pred Flush(s, s’: System) {
    some x: set s.cache.addrs |
        s’.cache.map = s.cache.map - (x -> Data)
        s’.cache.dirty = s.cache.dirty - x
        s’.main.map = s.main.map ++
            {a: x, d: Data | d = s.cache.map[a]}

}

pred SysWrite(s, s’: System, d: Data, a: Addr) {
    Write(s.cache, s’.cache, d, a)
    s’.cache.dirty = s.cache.dirty + a
    s’.main = s.main
}
```

5.2 Facts and Assertions

A system specification can be complemented with facts, which represent properties of the system:

```
fact {no (MainMemory & Cache) }

fact {all s: System | s.cache.addrs in s.main.addrs }
```

Properties to be validated are specified as *assertions*:

```
assert {
    all s: System |
        Dirtyinv(s) && no s.cache.dirty =>
            s.cache.map in s.main.map
}
```

6 Validating Properties of Executions: The Alloy Style

Although Alloy is not an appropriate language for specifying properties of executions, a method for specifying execution traces and their properties in Alloy has been proposed. It consists of:

- extending a system specification with further domains (signatures) for clock ticks and traces,
- defining how traces are built out of the operations (predicates) of the system specification.

For instance, if one would need to check whether a system preserves property `DirtyInv` under sequences of “calls” to `Flush` and `SysWrite`, it would be necessary to:

- express that the initial state satisfies `DirtyInv`
- express that every pair of consecutive states are related either by `Flush` or `SysWrite`
- assert that the final state satisfies `DirtyInv`.

```
sig Tick { }

sig SystemTrace {
    ticks: set Tick
    first, last: Tick
    next: (ticks - last) !->! (ticks - first)
    state: ticks ->! System
}

fact {
    first.next* = ticks, DirtyInv(first.state)
    all t: ticks - last |
    some s = t.state, s' = t.next.state |
    Flush(s,s') || some d: Data, a: Addr
                  | SysWrite(s, s', d, a) }
```

6.1 Features of Alloy’s Style for the Specification of Executions

- It allows for fully automated validation of properties of traces
- It allows for the validation of a wide range of properties (essentially, anything expressible using quantification over ticks)
- It allows for the validation of invariants even if these are not *inductive*
- The definition of traces and ticks is *ad hoc*, and depends on the particular property to be validated,
- The mechanism does not favour incremental validation

7 DynAlloy

DynAlloy is an extension of Alloy to support the definition of actions and the specification of properties regarding execution traces.

Actions are meant to replace predicates as operation descriptions, and can be:

- Basic actions, assumed to be *atomic*, and described via partial correctness assertions:

```
{ true }
Write(m: Memory, d: Data, a: Addr)
{ m'.map = m.map ++ (a -> d) }
```

- Composite actions, which correspond to *programs*, and are built using nondeterministic choice, sequential composition, iteration, etc, using atomic actions as the base case:

```
(SysWriteDA(s) + FlushDA(s))*
```

7.1 Assertions regarding Execution Traces

Assertions regarding execution traces are specified using programs and partial correctness assertions:

```
{ DirtyInv(s) }
(SysWriteDA(s) + FlushDA(s))*
{ DirtyInv(s’) }

{ Init(s) }
(SysWriteDA(s) + FlushDA(s))*
{ FreshDir(s’) }
```

7.2 Features of DynAlloy's Style for the Specification of Executions

- Better separation of concerns (dynamic and static aspects are specified separately)
- The treatment of execution traces is done "behind the scenes"
- Improved analysability
- Possibility of performing incremental validation
- Restricted expressive power for the characterisation of properties of executions (only partial correctness assertions)

8 Analysis of DynAlloy Specifications

In order to automatically validate a DynAlloy assertion of type:

```
{ pre }
P
{ post }
```

we compute the *weakest liberal precondition* wlp[P,post] of P and post, and validate the formula:

```
pre => wlp[P,post]
```

BUT, wlp is traditionally defined as follows:

$$\textit{wlp}[g?, f] = g \Rightarrow f$$
$$\textit{wlp}[p1 + p2, f] = \textit{wlp}[p1, f] \wedge \textit{wlp}[p2, f]$$
$$\textit{wlp}[p1; p2, f] = \textit{wlp}[p1, \textit{wlp}[p2, f]]$$
$$\textit{wlp}[p*, f] = \textstyle\bigwedge_{i=0}^{\infty} \textit{wlp}[p^i, f]$$

which for the case of iteration does not necessarily yield a formula. So, for the case of iteration, we **impose a bound** k, and define *wlp* as:

$$\textit{wlp}[p*, f] = \bigwedge_{i=0}^{k} \textit{wlp}[p^i, f]$$

8.1 Analysing Choice within Iteration

A typical pattern of assertions to validate is the following:

```
{ pre }
(a1 + ... + an)*
{ post }
```

Using Alloy, with a bound of k for the size of traces, the number of traces is exponential with respect to k.

For a similar pattern, and a fixed number k of iterations, DynAlloy generates a formula whose size is *linear* with respect to k.

8.2 Some Experimental Results

Tr. length $\longrightarrow$	$\leq$ 2		$\leq$ 3		$\leq$ 4	
$\#$ elems $\downarrow$	Alloy	DAlloy	Alloy	DAlloy	Alloy	DAlloy
3	$0'01''$	$0'01''$	$0'04''$	$\mathbf{0'02''}$	$0'20''$	$\mathbf{0'07''}$
4	$0'13''$	$\mathbf{0'01''}$	$3'04''$	$\mathbf{0'11''}$	$45'25''$	$\mathbf{1'48''}$
5	$1'40''$	$\mathbf{0'03''}$	$34'40''$	$\mathbf{0'59''}$	$> 60'$	$\mathbf{31'17''}$
6	$5'52''$	$\mathbf{0'06''}$	$> 60'$	$\mathbf{2'24''}$	$> 60'$	$> 60'$

Table 1: Verification time for the assertion *DirtyInv* using the SAT solver *MChaff*.

Tr. length $\longrightarrow$	$\leq$ 2		$\leq$ 3		$\leq$ 4	
# elems $\downarrow$	Alloy	DAlloy	Alloy	DAlloy	Alloy	DAlloy
3	$\mathbf{0'02''}$	$0'06''$	$\mathbf{0'07''}$	$2'17''$	$\mathbf{0'24''}$	$31'53''$
4	$\mathbf{0'16''}$	$0'18''$	$6'48''$	$\mathbf{1'57''}$	$> 60'$	$> 60'$
5	$5'31''$	$\mathbf{0'40''}$	$> 60'$	$\mathbf{9'53''}$	$> 60'$	$> 60'$
6	$> 60'$	$\mathbf{0'58''}$	$> 60'$	$> 60'$	$> 60'$	$> 60'$

Table 2: Verification time for the assertion *DirtyInv* using the SAT solver *ZChaff*.

Tr. length $\longrightarrow$	$\leq$ 2		$\leq$ 3		$\leq$ 4	
# elems $\downarrow$	Alloy	DAlloy	Alloy	DAlloy	Alloy	DAlloy
3	$0'01''$	$0'01''$	$0'01'$	$0'01''$	$0'06''$	$\mathbf{0'01''}$
4	$0'05''$	$\mathbf{0'01''}$	$0'29''$	$\mathbf{0'02''}$	$4'35''$	$\mathbf{0'09''}$
5	$0'16''$	$\mathbf{0'01''}$	$2'01''$	$\mathbf{0'10''}$	$20'18''$	$\mathbf{0'47''}$
6	$0'55''$	$\mathbf{0'04''}$	$8'15''$	$\mathbf{0'32''}$	$> 60'$	$\mathbf{5'29''}$

Table 3: Verification time for the assertion *DirtyInv* using the SAT solver *Berkmin*.

	MChaff		Berkmin	
$Tr. length \downarrow$	Alloy	DAlloy	Alloy	DAlloy
3	$0'01''$	$0'01''$	$0'01''$	$0'01''$
4	$0'11''$	$\mathbf{0'02''}$	$0'05''$	$\mathbf{0'02''}$
5	$17'12''$	$\mathbf{0'43''}$	$\mathbf{0'05''}$	$0'19''$
6	$> 60'$	$\mathbf{2'55''}$	$10'30''$	$\mathbf{8'09''}$
7	$> 60'$	$\mathbf{59'18''}$	$> 60'$	$> 60'$

Table 4: Verification times for *FreshDir*.

9 Incremental Validation

DynAlloy also allows for incremental validation. Assume that we define a new signature:

```
sig ComplexSystem { s: System }
```

and some programs for it:

```
CSysWriteDA(c: ComplexSystem) = SysWriteDA(c.s)

BlockFlushDA(c: ComplexSystem) = (FlushDA(c.s))*
```

Now assume we want to validate the following assertion:

```
{ DirtyInv(c.s) }
(CSysWriteDA(c) + BlockFlushDA(c))*
{ DirtyInv(c'.s)}
```

To validate this assertion, we have at least two choices:

- unfold the definitions of the programs and validate the resulting assertion, or
- perform incremental validation, by:
  - first validating

    ```
    { DirtyInv(c.s) }
    BlockFlushDA(c)
    { DirtyInv(c'.s)}
    ```

  - and then, if the validation succeeded, validating the above program, BUT considering `BlockFlushDA(c)` as an **atomic action**, specified by the validated assertion.

9.1 The Impact of Incremental Validation

If we validate the above original assertion via unfolding of definitions, we get nested iterations. Thus, the lengths of the traces to explore goes up to k, where k is the bound on iteration (for Alloy) (similar effect in the size of the wlp formula for DynAlloy). This makes unfeasible the validation **even for small values of k**.

By performing incremental validation, we split the validation in two, leading us, intuitively, to the **exploration of longer computations** during the validation tasks.

10 Conclusions

DynAlloy is an extension of Alloy to better characterise operations and the specification of properties regarding execution traces. Its main advantages, over Alloy, are:

- Better separation of concerns
- Specification of dynamic properties at a high level of abstraction
- Improved analysability for a common class of properties
- Incremental validation is straightforward

11 Current Activities regarding (Dyn)Alloy

- Make (Dyn)Alloy Analysis *scalable*
  - via abstraction techniques
  - via combining deductive reasoning and SAT analysis
- Use of DynAlloy for the analysis of code
- Use of DynAlloy for the analysis of requirements specifications
- Use of DynAlloy for the analysis of dynamic software architectures

. . .