A verified VCGen based on Dynamic Logic: an exercise in meta-verification with Why3

Maria João Frade[1], Jorge Sousa Pinto[2]

Logics and Calculi for All
Dedicated to Luís Soares Barbosa on the occasion of his 60th Birthday

HASLab/INESC TEC & Universidade do Minho, Portugal

Preliminaries

Context and Motivation

- General interest: the study of verification condition (VC) generation
- in the context of deductive verifiers based on program logics or calculi
- following the typical architecture:

  VCGen + automated prover/solver for FOL
- Aspects of VC generation have practical impact:

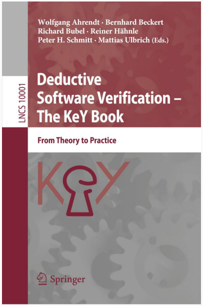
  forward/backward strategy; size of VCs; SA form; ...
- Dynamic logic: a program logic

The KeY Project and Tool

Quoting `key-project.org`:

> *The core feature of KeY is a theorem prover for Java Dynamic Logic based on a sequent calculus*

Does not follow the "typical architecture" ...



KeY's DL in a nutshell (1)

Program-carrying modalities

$[C]\phi$: "every terminating execution of C results in a state that satisfies ϕ"

$$[x := e]\phi \; = \; \phi[e/x] \qquad\qquad [C_1; C_2]\phi \; = \; [C_1][C_2]\phi$$

$$[\mathit{if}\ b\ \mathit{then}\ C_1\ \mathit{else}\ C_2]\phi \; = \; (b \rightarrow [C_1]\phi) \wedge (\neg b \rightarrow [C_2]\phi)$$

$$\frac{\theta \wedge b \; \rightarrow \; [C]\theta}{\theta \; \rightarrow \; [\mathit{while}\ b\ \mathit{do}\ C](\theta \wedge \neg b)}$$

...extremely familiar from the standpoint of WP calculus and Hoare logic

KeY's DL in a nutshell (2)

State Updates

Programs of a special form, essentially *parallel assignments*
$[x_1 := e_1 \mid\mid \ldots \mid\mid x_n := e_n]$

- may be *applied* to expressions or formulas
- application is "rightmost wins" parallel variable substitution
- *simplification rules* required to handle formulas like $[\mathcal{U}]([\mathcal{U}'] \psi)$, e.g.

$$[\mathcal{U}]\,([x_1 := e_1 \mid\mid \ldots \mid\mid x_n := e_n]\,\psi) \;\rightsquigarrow\; [\mathcal{U} \mid\mid x_1 := \mathcal{U}\,(e_1) \mid\mid \ldots \mid\mid x_n := \mathcal{U}\,(e_n)]\,\psi$$

Updates were introduced as a device to handle object aliasing. Their simplification is a forward propagation process resembling a strongest postcondition computation

(But: free of existential quantifiers and not requiring SA form)

KeY's DL in a nutshell (3)

State Updates in modern KeY

- are seen as separate entities, no longer as programs
- inference rules and update simplification promote symbolic execution

$$\frac{\phi \;\wedge \{\mathcal{U}\} b \;\rightarrow\; \{\mathcal{U}\}[C_1\,;\,C]\psi \qquad\qquad \phi \;\wedge \{\mathcal{U}\}(\neg b) \;\rightarrow\; \{\mathcal{U}\}[C_2\,;\,C]\psi}{\phi \;\rightarrow\; \{\mathcal{U}\}[\mathit{if}\;\; b\; \mathit{then}\; C_1\; \mathit{else}\; C_2\,;\, C]\psi}$$

$$\begin{aligned}
&\phi \rightarrow [x := 2 * x\,;\, y := x]\psi\\
&\quad \phi \rightarrow \{x := 2 * x\}\;[y := x]\psi\\
&\qquad \phi \rightarrow \{x := 2 * x\}\;(\{y := x\}\;\psi)\\
&\qquad \phi \rightarrow \{x := 2 * x \;||\; \{x := 2 * x\}\; y := x\}\;\psi)\\
&\qquad \phi \rightarrow \{x := 2 * x \;||\; y := 2 * x\}\;\psi)
\end{aligned}$$

Formalization of a DL-based Verifier

- Initial idea: to explore the use of Why3 to formalize a simple program logic and prove its properties
- Then wrote a VCGen for the logic. It includes a strategy for update simplification and produces FOL proof obligations
- Verifying the VCGen: an exercise in meta-verification.
  Quis custodiet ipsos custodes?
- Shows how dynamic logic with updates can serve as the basis for an alternative verifier following the "typical architecture"
- It highlights distinctive aspects of Why3, in particular the rich relationship between its logic and programming languages.

Formalization of a DL-based Verifier

- We define a fragment of JavaDL for While programs that we call WhileDL
- We formalize its syntax and semantics in Why3
- We formalize an inference system for WhileDL and mechanically prove its soundness and (a notion of) completeness
- We introduce a VCGen that produces FOL proof obligations, and prove its soundness
- Our proofs use (just a few) proof transformations and (mostly) external SMT solvers
- The verified VCGen can be extracted as an OCaml program

Why3 in a Nutshell

- A logic language: FOL; algebraic types; inductive predicates; rich logic library
- WhyML programming language: functional with mutability
- Pure program functions may exist in both namespaces
- Proof manager: external tool interaction; proof sessions; transformations; smoke detection; hypotheses bissection
- Verification based on contracts and clonable modules (refinement VCs)

Why3 Example: (functional) Insertion Sort

```
module InsertionSort
  use int.Int, list.List, list.Permut, list.SortedInt

  val function insert (i: int) (l: list int) : list int
    requires { sorted l }
    ensures { sorted result }
    ensures { permut result (Cons i l) }

  let rec function iSort (l: list int) : list int
    ensures { sorted result }
    ensures { permut result l }
  = match l with
    | Nil -> Nil
    | Cons h t -> insert h (iSort t )
    end
end
```

Why3 Example: (functional) Insertion Sort

```
module InsertionSortRfn
  use  ...

  let rec function insert (i: int) (l: list int) : list int
    requires { sorted l }
    ensures { sorted result }
    ensures { permut result (Cons i l) }
  = match l with
    | Nil -> Cons i Nil
    | Cons h t -> if i <= h then Cons i l else Cons h (insert i t)
    end

  clone InsertionSort with val insert      (* will generate VCs! *)

  goal itSorts : forall l :list int. let ls = iSort l in sorted ls /\ permut ls l
end
```

The WhileDL Dynamic Logic

Semantics

The interpretation of an update $\mathcal{U} \in \textbf{Upd}$ in a given state is a state transformer function $[\![\mathcal{U}]\!] : \Sigma \to (\Sigma \to \Sigma)$:

$$
\begin{array}{lcl}
[\![\texttt{skip}]\!](s)(s') & = & s' \\
[\![x := a]\!](s)(s') & = & s'[x \mapsto [\![a]\!](s)] \\
[\![\mathcal{U}_1 \| \mathcal{U}_2]\!](s)(s') & = & [\![\mathcal{U}_2]\!](s)([\![\mathcal{U}_1]\!](s)(s')) \\
[\![\{\mathcal{U}_1\}\mathcal{U}_2]\!](s)(s') & = & [\![\mathcal{U}_2]\!]([\![\mathcal{U}_1]\!](s)(s))(s')
\end{array}
$$

Expressions are interpreted in the usual way. For update aplications:

$$
[\![\{\mathcal{U}\}\, a]\!](s) \quad = \quad [\![a]\!]([\![\mathcal{U}]\!](s)(s))
$$

The usual interpretation of first-order formulas is extended with the two following cases:

$$
\begin{array}{lll}
[\![\{\mathcal{U}\}\,\phi]\!](s) = \mathbf{T} & \text{iff} & [\![\phi]\!]([\![\mathcal{U}]\!](s)(s)) = \mathbf{T} \\
[\![[C]\,\phi]\!](s) = \mathbf{T} & \text{iff} & [\![\phi]\!](s') = \mathbf{T} \ \text{ for } s' \text{ such that } \ \langle C, s\rangle \Downarrow s'
\end{array}
$$

Semantics in Why3

We call a formula of the form $\phi \to \{\mathcal{U}\}\,[C]\,\psi$ an update triple

```
predicate satisfies (s:state) (p:fmla) =
match p with | ...
             | Fand p1 p2 -> (satisfies s p1) /\ (satisfies s p2)
             | Fsqb c p ->  forall s' :state. big_step s c s' -> satisfies s' p
             | Fupd u p ->  satisfies (eval_upd s u s) p
end

predicate valid_fmla (p:fmla) = forall s:state. satisfies s p

predicate validUT (p:fmla)(u:upd)(c:stmt)(q:fmla) =
            valid_fmla (Fimplies p (Fupd u (Fsqb c q)))
```

The WhileDL Calculus

$$(\cdots)$$

$$\frac{\phi \rightarrow \{\mathcal{U} \| \{\mathcal{U}\}\, x := a\}\, [C]\, \phi}{\phi \rightarrow \{\mathcal{U}\}\, [x := a\,;\ C]\, \phi} \quad \text{(assign-seq)}$$

$$\frac{\phi \wedge \{\mathcal{U}\}\, b \rightarrow \{\mathcal{U}\}\, [C_1\,;\ C_3]\, \psi \quad \phi \wedge \{\mathcal{U}\}\, \neg b \rightarrow \{\mathcal{U}\}\, [C_2\,;\ C_3]\, \psi}{\phi \rightarrow \{\mathcal{U}\}\, [(\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2)\,;\ C_3]\, \psi} \quad \text{(if-seq)}$$

$$\frac{\phi \rightarrow \{\mathcal{U}\}\, \theta \quad \theta \wedge b \rightarrow \{\texttt{skip}\}\, [C_1]\, \theta \quad \theta \wedge \neg b \rightarrow \{\texttt{skip}\}\, [C_2]\, \psi}{\phi \rightarrow \{\mathcal{U}\}\, [(\texttt{while}\ b\ \texttt{do}\ \{\theta\}\ C_1)\,;\ C_2]\, \psi} \quad \text{(while-seq)}$$

The WhileDL Calculus in Why3

```
inductive infUT fmla upd stmt fmla =
(...)
| infUT_assignseq: forall p:fmla, q:fmla, x:ident, e:expr, c:stmt, u:upd.
        infUT p (Upar u (Uupd u (Uassign x e))) c q -> infUT p u (Sseq
        (Sassign x e) c) q

| infUT_ifseq: forall p q:fmla, c1 c2 c:stmt, b:bexpr, u:upd.
        infUT (Fand p (Fupd u (Fembed b))) u (Sseq c1 c) q ->
        infUT (Fand p (Fupd u (Fnot (Fembed b)))) u (Sseq c2 c) q ->
        infUT p u (Sseq (Sif b c1 c2) c) q

| infUT_whileseq: forall p q:fmla, c cc:stmt, b:bexpr, inv ainv :fmla, u:upd.
        valid_fmla (Fimplies p (Fupd u inv)) ->
        infUT (Fand inv (Fembed b)) Uskip c inv ->
        infUT (Fand inv (Fnot (Fembed b))) Uskip cc q ->
        infUT p u (Sseq (Swhile b ainv c) cc) q
```

WhileDL **Soundness and Completeness**

In Why3 inductive proofs can be written as lemma functions

```
let rec lemma infUT_sound_complete (c:stmt) =
    ensures { forall p q :fmla, u :upd. validUT p u c q <-> infUT p u c q  }
    variant { size c }
match c with
| Sskip -> ()
| Sassign _ _ -> ()
| Sif _ c1 c2 -> infUT_sound_complete c1 ; infUT_sound_complete c2
| Swhile _ _ c -> infUT_sound_complete c
| Sseq Sskip c -> infUT_sound_complete c
| Sseq (Sassign _ _) c -> infUT_sound_complete c
| Sseq (Sif _ c1 c2) c -> infUT_sound_complete (Sseq c1 c) ;
                          infUT_sound_complete (Sseq c2 c)
| Sseq (Swhile _ _ c1) c -> infUT_sound_complete c1 ; infUT_sound_complete c
| Sseq (Sseq c1 c2) c -> infUT_sound_complete (Sseq c1 (Sseq c2 c))
end
```

The VC Generator

WhileDL **Update Simplification**

1. $\{\ldots \| x := a_1 \| \ldots \| x := a_2 \| \ldots\}\ t \rightsquigarrow \{\ldots \| \texttt{skip} \| \ldots \| x := a_2 \| \ldots\}\ t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
2. $\{\ldots \| x := a \| \ldots\}\ t \rightsquigarrow \{\ldots \| \texttt{skip} \| \ldots\}\ t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$ and $x \notin \mathsf{FV}(t)$
3. $\{\mathcal{U}_1\}\ \{\mathcal{U}_2\}\ t \rightsquigarrow \{\mathcal{U}_1 \| \{\mathcal{U}_1\}\ \mathcal{U}_2\}\ t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
4. $\{\mathcal{U} \| \texttt{skip}\}\ t \rightsquigarrow \{\mathcal{U}\}\ t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
5. $\{\texttt{skip} \| \mathcal{U}\}\ t \rightsquigarrow \{\mathcal{U}\}\ t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
6. $\{\texttt{skip}\}\ t \rightsquigarrow t$ where $t \in \mathbf{AExp} \cup \mathbf{BExp} \cup \mathbf{Form} \cup \mathbf{Upd}$
7. $\{\mathcal{U}\}\ t \rightsquigarrow t$ where $t \in \mathbf{Var} \cup \{\texttt{true}, \texttt{false}\} \cup \{\ldots, -1, 0, 1, \ldots\}$
8. $\{\mathcal{U}\}\ (a_1 \bullet a_2) \rightsquigarrow (\{\mathcal{U}\}\ a_1) \bullet (\{\mathcal{U}\}\ a_2)$ where $\bullet \in \{+, *, -, =, <, >, \leq, \geq\}$
9. $\{\mathcal{U}\}\ \neg b \rightsquigarrow \neg\{\mathcal{U}\}\ b$
10. $\{\mathcal{U}\}\ (b_1 \bullet b_2) \rightsquigarrow (\{\mathcal{U}\}\ b_1) \bullet (\{\mathcal{U}\}\ b_2)$ where $\bullet \in \{\wedge, \vee\}$
11. $\{\mathcal{U}\}\ \neg\phi \rightsquigarrow \neg\{\mathcal{U}\}\ \phi$
12. $\{\mathcal{U}\}\ (\phi_1 \bullet \phi_2) \rightsquigarrow (\{\mathcal{U}\}\ \phi_1) \bullet (\{\mathcal{U}\}\ \phi_2)$ where $\bullet \in \{\wedge, \vee, \rightarrow\}$
13. $\{\mathcal{U}\}\ \forall x.\ \phi \rightsquigarrow \forall x.\ \{\mathcal{U}\}\ \phi$ where $x \notin \mathsf{FV}(\mathcal{U})$
14. $\{\mathcal{U}\}\ \exists x.\ \phi \rightsquigarrow \exists x.\ \{\mathcal{U}\}\ \phi$ where $x \notin \mathsf{FV}(\mathcal{U})$
15. $\{\mathcal{U}\}\ (x := a) \rightsquigarrow x := \{\mathcal{U}\}\ a$
16. $\{\mathcal{U}\}\ \texttt{skip} \rightsquigarrow \texttt{skip}$
17. $\{\mathcal{U}\}\ (\mathcal{U}_1 \| \mathcal{U}_2) \rightsquigarrow (\{\mathcal{U}\}\ \mathcal{U}_1) \| (\{\mathcal{U}\}\ \mathcal{U}_2)$
18. $\{x := a\}\ x \rightsquigarrow a$

VCGen

Not a decision procedure for DL formulas in general! Takes an update triple $\phi \to \{\mathcal{U}\}\,[C]\,\psi$ subject to "well-formedness" restrictions: C does not contain expressions with updates, ϕ, ψ do not contain statements ...

```
let rec ghost function vcgen (p:fmla) (u:upd) (c:stmt) (q:fmla) : fset fmlaFOL
  requires { stmt_freeF p /\ upd_freeF p /\ parUpd u /\ progInv c /\ stmt_freeF q }
  ensures  { valid_fmlas result -> validUT p u c q }
  variant  { size c }
= match c with
  | (...)
  | (Sseq (Sassign x e) c) -> vcgen p (concat u (applyU u (Uassign x e))) c q

  | (Sseq (Sif b c1 c2) c) ->
      union (vcgen (Fand p (applyF u (Fembed b))) u (Sseq c1 c) q)
            (vcgen (Fand p (applyF u (Fnot (Fembed b)))) u (Sseq c2 c) q)

  | (Sseq (Swhile b inv c1) c) ->
      addFOL (Fimplies p (applyF u inv))
             (union (vcgen (Fand inv (Fembed b)) Uskip c1 inv)
                    (vcgen (Fand inv (Fnot (Fembed b))) Uskip c q))
  end
```

Wrapping Up

Conclusions

- Design of a VCGen producing first-order verification conditions – proof of concept of how a DL-based verifier can be constructed making used of standard first-order proof tools
- Non-trivial case study in program verification with Why3: a functional program (VCGen + simplifier), with a complex spec
- Online repository also contains an execution version of the VCGen, refining abstract type of finite sets by concrete mutable sets
- Extraction to OCaml code using Why3's program extraction facility results in an actual executable, correct-by-construction VCGen

Why3 module files, proof sessions, proof summaries available from `https://github.com/jspdium/dlKeY`.

(Buenos Aires, 2008)