# Definitions.thy

```
theory Definitions
imports
  Main
begin

section {* DFS framework *}

text {* Nested DFS is built on top of a generic framework for DFS, i.e. a generic
        algorithm parametrized over functions that are instantiated differently
        for the blue and red DFSs. *}

text {* The generic DFS framework  *}

text {* The framework uses the following data structure: *}

record ('S, 'n) dfs_sws =
  stack :: "'n list"
    -- "the dfs stack"
  state :: "'S"
    -- "additional state for implementation purposes"
  start :: "'n"
    -- "the starting node of the complete DFS"
  discovered :: "'n set"
    -- "the set of all nodes discovered by the DFS so far "

text {* Further, the set of all nodes the DFS has backtracked from is given
        implicitly as: *}

definition "finished s ≡ discovered s - set (stack s)"

text {* The topmost function of the framework is dfs_algo,
        which is parametrized by
           dfs_cond  (condition to be satisfied by the state to continue
                      searching from it)
        and calls the functions
           dfs_alg_start (code to be executed initially)
           dfs_algo_body (code for the search from the state). *}

definition dfs_algo :: "'n ⇒ ('S, 'n) dfs_sws nres"  where
  "dfs_algo x₀ ≡ do {
     let s₀ = dfs_alg_start x₀;
     if ¬ dfs_cond dfs (state s₀) then RETURN s₀
     else dfs_algo_body s₀ x₀
   }"

text {* dfs_alg_start is parametrized by dfs_start, the starting state *}

definition dfs_alg_start :: "'n ⇒ ('S, 'n) dfs_sws"
where "dfs_alg_start x₀ ≡
   (| stack = [], state = dfs_start dfs x₀, start = x₀, discovered = {} |)"

text {* dfs_algo_body is also parametrized by dfs_cond, and calls
```

```
         dfs_rem_step  (code to be executed when the state has already been
                        previously discovered)
         dfs_disc_step (code to be executed when the state is new)
         dfs_fin_step  (code to be executed after backtracking from the state)
*}

definition dfs_algo_body :: "('S, 'n) dfs_sws ⇒ 'n ⇒ ('S, 'n) dfs_sws nres"
where
  "dfs_algo_body s₀ x₀ ≡ REC_T (λD (s, x). do {
           if x ∈ discovered s then dfs_rem_step x s
           else
              (dfs_disc_step x s >>=
              FOREACH_C (succs x) (λs. dfs_cond dfs (state s))
                        (λ x s. D (s, x))) >>=
              (λ s'. dfs_fin_step x s')
        }) (s₀, x₀)"

text {* dfs_rem_step is parametrized by dfs_remove *}

definition dfs_rem_step :: "'n ⇒ ('S, 'n) dfs_sws ⇒ ('S, 'n) dfs_sws nres"
where
  "dfs_rem_step x s ≡ do {
      S' ← dfs_remove dfs s x;
      RETURN (s(| state := S' |))
  }"

text {* dfs_disc_step is parametrized by
        dfs_visit (code to be executed BEFORE inspecting the successors of
                   the node) *}

definition dfs_disc_step :: "'n ⇒ ('S, 'n) dfs_sws ⇒ ('S, 'n) dfs_sws nres"
where
  "dfs_disc_step x s ≡ do {
       S ← dfs_visit dfs s x;
       RETURN (disc_step_upd x s S)
  }"

definition disc_step_upd :: "['n, ('S, 'n) dfs_sws, 'S] ⇒ ('S, 'n) dfs_sws"
where
  "disc_step_upd x s S ≡ s(|stack := x # stack s, discovered := insert x (discovered
s), state := S|)"

text {* dfs_fin_step is parametrized by
        dfs_post (code to be executed AFTER inspecting the successors of
                  the node) *}

definition dfs_fin_step :: "'n ⇒ ('S, 'n) dfs_sws ⇒ ('S, 'n) dfs_sws nres"
where
  "dfs_fin_step x s ≡ if dfs_cond dfs (state s) then do {
      S ← dfs_post dfs s x;
      RETURN (fin_step_upd x s S)
    } else RETURN s"

definition fin_step_upd :: "['n, ('S, 'n) dfs_sws, 'S] ⇒ ('S, 'n) dfs_sws"
```

```
where
  "fin_step_upd x s S ≡ s(|stack := tl (stack s), state := S|)"

text {* Here is the record of all parameters mentioned above *}

record ('S, 'n) dfs_algorithm =
  dfs_cond :: "'S ⇒ bool"
  dfs_visit :: "('S,'n) dfs_sws ⇒ 'n ⇒ 'S nres"
  dfs_post :: "('S, 'n) dfs_sws ⇒ 'n ⇒ 'S nres"
  dfs_remove :: "('S, 'n) dfs_sws ⇒ 'n ⇒ 'S nres"
  dfs_start :: "'n ⇒ 'S"

section {* Nested DFS *}

text {* Now we give the code for Nested DFS
        The topmost function is nested_dfs, which calls dfs_algo with the
        record of parameters set to the values for the blue DFS given below,
        and converts some internal state into a more usable result *}

definition nested_dfs :: "'n ⇒ 'n nested_res nres" where
  "nested_dfs x = do {
                s ← dfs_algo x;
                case (state s) of
                   CYCLE_FIN y v ys vs cyc ⇒
                          if y ∈ A then
                             RETURN LASSO y ys (vs ⊕ cyc)
                          else
                             RETURN LASSO v (ys ⊕ vs) (cyc ⊕ vs)
                 | _ ⇒ RETURN NO_LASSO }"

subsection {* Blue DFS *}

text {* Blue DFS is the outer part of the Nested DFS.
        It instantiates the DFS framework as follows:  *}

definition blue_dfs :: "(('n, 'n set) blue_res, 'n) dfs_algorithm" where
  "blue_dfs = (| dfs_cond = λs. (case s of CYCLE_NO _ ⇒ True | _ ⇒ False),
                dfs_visit = λ s _. RETURN (state s),
                dfs_post = blue_dfs_post,
                dfs_remove = λ s _. RETURN (state s),
                dfs_start = λ_. CYCLE_NO {} |)"

text {* The function blue_dfs_post is parametrized over a function M
        and a set A.
        The function M returns the set of nodes to look for a cycle.
        The set A is the set of accepting nodes. *}

definition blue_dfs_post :: "'n blue_sws ⇒ 'n ⇒ ('n, 'n set) blue_res nres"
where
  "blue_dfs_post s x ≡ if x ∈ A then (case state s of CYCLE_NO R ⇒
                          do {
                             S' ← run_red_dfs R {y. M s R y} x;
                             case S' of
                                CYCLE_PRE v vs ⇒ do {
```

```
                                let rs = rev (stack s);
                                let cyc = mk_cycle rs v;
                                RETURN (CYCLE_FIN x v rs vs cyc) }
                            | _ ⇒ RETURN S'
                        }
                      | _ ⇒ RETURN (state s))
                      else RETURN (state s)"

definition run_red_dfs :: "'n set ⇒ 'n set ⇒ 'n ⇒ ('n, 'n set) blue_res
nres"
where
  "run_red_dfs R ss x ≡ do {
              red ← red_dfs_algo R ss x;
              case state red of
                None ⇒ RETURN (CYCLE_NO (finished red ∪ R))
              | Some (v,vs) ⇒ RETURN (CYCLE_PRE v vs)}"

primrec mk_cycle :: "'n list ⇒ 'n ⇒ 'n list" where
  "mk_cycle (x#xs) v = (if x = v then x#xs else mk_cycle xs v)"

subsection {* Red DFS *}

text {* The function red_dfs_algo implements the inner DFS of
        the Nested DFS algorithm.
        It instantiates the DFS framework as follows: *}

definition red_dfs :: "('n red_cycle, 'n) dfs_algorithm" where
  "red_dfs = (| dfs_cond = λs. s = None,
                dfs_visit = red_dfs_visit,
                dfs_post = λ s _. RETURN (state s),
                dfs_remove = red_dfs_remove,
                dfs_start = λ_. None |)"

text {* These definitions are parametrized over a set ss,
        which is the set of nodes to look up for cycle detection. This differs
        depending on the Nested DFS algorithm (e.g. only the top of the stack
        for Courcoubetis et al. or the complete stack for Holzmann et al.).
*}

definition red_dfs_visit :: "'n red_sws ⇒ 'n ⇒ 'n red_cycle nres" where
  "red_dfs_visit s x ≡ case state s of
                      Some x ⇒ RETURN (Some x)
                    | None ⇒ if (x ≠ start s ∧ x ∈ ss) then
                               RETURN (mk_ret x s)
                             else
                               RETURN None"

definition red_dfs_remove :: "'n red_sws ⇒ 'n ⇒ 'n red_cycle nres" where
  "red_dfs_remove s x ≡ case state s of
                        None ⇒ if (x = start s ∧ x ∈ ss) then
                                 RETURN (mk_ret x s)
                               else
                                 RETURN None
                      | Some x ⇒ RETURN (Some x)"
```

```
abbreviation mk_ret :: "'n ⇒ 'n red_sws ⇒ 'n red_cycle" where
  "mk_ret x s ≡ Some (x, rev (x#stack s))"



section {* Improved Nested DFS of Schwoon-Esparza *}

text {* The Nested DFS of Schwoon-Esparza is an extension of the well-known
        Nested DFS given above. It extends the Blue DFS with additional functionality:
*}

definition se_dfs where
  "se_dfs = blue_dfs(| dfs_remove := se_dfs_remove |)"

definition se_dfs_remove where
  "se_dfs_remove s x ≡ case state s of
              CYCLE_NO R ⇒ do {
                           let hs = hd (stack s);
                           if (x ∈ A ∨ hs ∈ A ) ∧ x ∈ set (stack s)
                           then do {
                             let rs = rev (stack s);
                             let cyc = mk_cycle rs x;
                             let p = [hs, x];
                             RETURN (CYCLE_FIN hs x rs p cyc) }
                           else RETURN (CYCLE_NO R) }
            | _ ⇒ RETURN (state s)"


section {* LTL-to-Buchi Translator *}

text {* The function, that computes a Buchi automaton for an LTL formula, is
        given by LTL_to_Buchi_elem and is divided into two parts: The calculation
        of the LGBA and the conversion of the LGBA to a Buchi automaton. *}

definition
  "LTL_to_Buchi_elem φ
   ≡ do { lgba ← LTL_to_LGBA (ltl_to_ltln (pushneg (ltlc_to_ltl φ)));
          buchi_elem ← LGBA_to_Buchi_elem lgba;
          RETURN buchi_elem }"

text {* The first part of the translation is based on the idea of
        Gerth et al. It requires, that the input formula is in negation normal
        form. *}

fun
  pushneg :: "'a ltl ⇒ 'a ltl"
  where
    "pushneg true = true"
  | "pushneg false = false"
  | "pushneg prop(q) = prop(q)"
  | "pushneg (not true) = false"
  | "pushneg (not false) = true"
  | "pushneg (not prop(q)) = not prop(q)"
```

```
  | "pushneg (not (not ψ)) = pushneg ψ"
  | "pushneg (not (ν and μ)) = pushneg (not ν) or pushneg (not μ)"
  | "pushneg (not (ν or μ)) = pushneg (not ν) and pushneg (not μ)"
  | "pushneg (not (X ψ)) = X pushneg (not ψ)"
  | "pushneg (not (ν U μ)) = pushneg (not ν) V pushneg (not μ)"
  | "pushneg (not (ν V μ)) = pushneg (not ν) U pushneg (not μ)"
  | "pushneg (φ and ψ) = (pushneg φ) and (pushneg ψ)"
  | "pushneg (φ or ψ) = (pushneg φ) or (pushneg ψ)"
  | "pushneg (X φ) = X (pushneg φ)"
  | "pushneg (φ U ψ) = (pushneg φ) U (pushneg ψ)"
  | "pushneg (φ V ψ) = (pushneg φ) V (pushneg ψ)"

text {* The nested recursive core function of the method by Gerth et al.
        is given by expand. *}

definition "expand_init ≡ 0"
definition "expand_new_name ≡ Suc"

abbreviation expand_body
where
  "expand_body ≡ (λexpand (n,ns).
      if new n = {} then
        ( if (∃n'∈ns. old n' = old n ∧ next n' = next n) then RETURN (name
n, upd_incoming n ns)
          else expand (⦇ name=expand_new_name (name n), incoming={name n},
new=next n, old={}, next={} ⦈, {n} ∪ ns) )
      else do {
        φ ← SPEC (λx. x∈(new n));
        let n = n⦇ new := new n - {φ} ⦈;
        if (∃q. φ = prop_n(q) ∨ φ = nprop_n(q)) then
          (if (frml_neg φ)∈old n then RETURN (name n, ns)
           else expand (n⦇ old := {φ} ∪ old n ⦈, ns))
        else if φ = true_n then expand (n⦇ old := {φ} ∪ old n ⦈, ns)
        else if φ = false_n then RETURN (name n, ns)
        else if (∃ν μ. (φ = ν and_n μ) ∨ (φ = X_n ν)) then
               expand (n⦇ new := new1 φ ∪ new n, old := {φ} ∪ old n, next
:= next1 φ ∪ next n ⦈, ns)
        else do {
          (nm, nds) ← expand (n⦇ new := new1 φ ∪ new n, old := {φ} ∪ old
n, next := next1 φ ∪ next n ⦈, ns);
          expand (n⦇ name := nm, new := new2 φ ∪ new n, old := {φ} ∪ old
n ⦈, nds)
        }
      }
     )"

definition expand :: "('a node × ('a node set)) ⇒ (node_name × 'a node set)
nres" where
  "expand ≡ REC expand_body"

text {* If through recursive calls a state is computed, that is already present,
        then the incoming edges of the present state are updated and the recursion
        stops.  *}
```

```
abbreviation
  "upd_incoming_f n ≡ (λn'. if (old n' = old n ∧ next n' = next n) then n'(|
incoming := incoming n ∪ incoming n' |) else n')"

definition
 "upd_incoming n ns ≡ ((upd_incoming_f n) ‘ ns)"

text {* The function create_graph starts the recursive computation of the automaton
        with appropriate start values. The output is a set of nodes, that can
        be interpreted as an LGBA. *}

definition create_graph :: "'a frml ⇒ 'a node set nres"
where
  "create_graph φ ≡
    do { (_, nds) ← expand (| name = expand_new_name expand_init, incoming
= {expand_init}, new = {φ}, old = {}, next = {} |)::'a node, {}::'a node set);
         RETURN nds }"

text {* The interpretation of the LGBA is done by create_lgba. *}

fun subfrmlsn :: "'a ltln ⇒ 'a ltln set"
where
  "subfrmlsn (μ and_n ψ) = {μ and_n ψ} ∪ subfrmlsn μ ∪ subfrmlsn ψ"
| "subfrmlsn (X_n μ) = {X_n μ} ∪ subfrmlsn μ"
| "subfrmlsn (μ U_n ψ) = {μ U_n ψ} ∪ subfrmlsn μ ∪ subfrmlsn ψ"
| "subfrmlsn (μ V_n ψ) = {μ V_n ψ} ∪ subfrmlsn μ ∪ subfrmlsn ψ"
| "subfrmlsn (μ or_n ψ) = {μ or_n ψ} ∪ subfrmlsn μ ∪ subfrmlsn ψ"
| "subfrmlsn x = {x}"

definition create_gba :: "['a frml, 'a node set] ⇒ ('a node) GBA"
where
  "create_gba φ qs ≡ (GBA_build
                        qs
                        {(q, q'). q∈qs ∧ q'∈qs ∧ name q∈incoming q'}
                        {q∈qs. expand_init∈incoming q}
                        {{q∈qs. μ U_n η∈old q ⟶ η∈old q}|μ η. μ U_n η
∈ subfrmlsn φ})"

definition create_lgba :: "['a frml, 'a node set] ⇒ ('a node, 'a prop set)
LGBA"
where
  "create_lgba φ qs ≡
    (| 𝒢ℬ𝒜 = create_gba φ qs,
      ℒ = λq l. {p. prop_n(p)∈old q}⊆l ∧ {p. nprop_n(p)∈old q} ∩ l={} |)"

text {* The states in above definitions are enriched with additional information,
        that are only necessary for the computation of the LGBA. Moreover each
        node can by identified by his name. As a consequence each state can
        be reduced to its name. *}

definition GBuchi_rename_states :: "['q ⇒ 'fq, ('q,'a) GBuchi] ⇒ ('fq,'a)
GBuchi"
where
  "GBuchi_rename_states f 𝒜 ≡
```

```
    SemiAutomaton_rename_states_ext
      (λf _. (| F = (λF. f ` F) ` F(A) |))
      (OmegaAutomaton_to_SemiAutomaton A) f"

abbreviation "GBA_rename_states f (A::'q GBA) ≡ GBuchi_rename_states f A"

definition LGBA_rename_states :: "['q ⇒ 'fq, ('q, 'l) LGBA] ⇒ ('fq, 'l)
LGBA"
where
  "LGBA_rename_states f A
   = (| GBA = GBA_rename_states f (GBA A),
       L = (λfq l. (if fq∈f`(Q_GBA A) then (L A (the_inv_into (Q_GBA
A) f fq) l) else False) ) |)"

definition create_name_lgba :: "['a frml, 'a node set] ⇒ (node_name, 'a prop
set) LGBA"
where
  "create_name_lgba φ nds ≡ LGBA_rename_states name (create_lgba φ nds)"

definition
  "LTL_to_LGBA φ
   ≡ do { nds ← create_graph φ;
          RETURN (create_name_lgba φ nds) }"


text {* The second part of the LTL-to-Bchi translation consists of the well-known
        counter construction, in order to reduce a generalized Buchi automaton
        to an ordinary one.
        Additionally the labels move from nodes to edges. *}

definition
  "LGBA_to_Buchi_elem A ≡
    do { F1 ← SPEC(λx. F_GBA A = set x ∧ distinct x);
         RETURN (| Buchi_elem_Δ = λ(q, k) a.
                   if L A q a then
                     {q'. (q,q') ∈ Δ_GBA A}
                        × {if k<length F1 ∧ q∈F1!k then (k+1) mod length
F1 else k}
                   else {},
                 Buchi_elem_I = I_GBA A × {0},
                 Buchi_elem_F = λ(q, k). (k = 0) ∧ (length F1 = 0 ∨ q∈F1!0)
|) }"


text {* The product construction on Buchi automata expects the first input
        to be a system automaton, where all states are accepting. *}

definition product_system_buchi_elem where
  "product_system_buchi_elem A_S A_B
     = (| Buchi_elem_Δ = λ(qs,qb) l. Buchi_elem_Δ A_S qs l × Buchi_elem_Δ
A_B qb l,
          Buchi_elem_I = Buchi_elem_I A_S × Buchi_elem_I A_B,
          Buchi_elem_F = λ(qs,qb). Buchi_elem_F A_B qb |)"
```

```
text {* As the Nested DFS algorithm can only handle one initial state, but
        the LTL-to-Buchi transformation might return an automaton with multiple
        initial states, we need to convert the latter to the former.

        This is done by adding 'Some' to each existing state and label and
        adding a new initial state 'None' with transitions labelled 'None'
        to each former initial state. *}

definition Buchi_elem_one_initial :: "('q, 'a) Buchi_elem ⇒ ('q option, 'a
option) Buchi_elem"
where "Buchi_elem_one_initial 𝒜 ≡ (| Buchi_elem_Δ = Buchi_elem_one_initial_Δ
𝒜,
                                     Buchi_elem_ℐ = {None},
                                     Buchi_elem_ℱ = option_case False (Buchi_elem_ℱ
𝒜) |)"

fun Buchi_elem_one_initial_Δ :: "('q,'a) Buchi_elem ⇒ 'q option ⇒ 'a option
⇒ 'q option set" where
  "Buchi_elem_one_initial_Δ 𝒜 (Some q) (Some l) = Some ` Buchi_elem_Δ 𝒜
q l"
| "Buchi_elem_one_initial_Δ 𝒜 None None = Some ` Buchi_elem_ℐ 𝒜"
| "Buchi_elem_one_initial_Δ 𝒜 _ _ = {}"

section {* The Model Checker *}

text {* The main function cava relies on cava_graph to build a product graph
        (with one initial state) out of the boolean program and the ltl formula.
        The result is then passed on to nested_dfs. *}

definition cava :: "bprog × config ⇒ nat ltlc ⇒ buchiq nested_res nres"
where
  "cava bpc φ = cava_graph bpc φ >>= nested_dfs"

definition cava_graph :: "bprog × config ⇒ nat ltlc ⇒ (graphq, nat set
option) Buchi_elem nres" where
  "cava_graph bpc φ = cava_BA bpc φ >>= RETURN ∘ Buchi_elem_one_initial"

text {* cava_BA creates the product buchi automaton out of the LTL formula
        and the boolean program. It utilizes LTL_to_Buchi_elem to receive
        the Buchi automaton of the LTL and BP_to_SA_elem, which creates a
        Buchi automaton out of the boolean program. *}

definition cava_BA :: "(bprog × config) ⇒ nat ltlc ⇒ (prodq, nat set) Buchi_elem
nres"  where
  "cava_BA bpc φ = do {
                    𝒜_B ← LTL_to_Buchi_elem (not_c φ);
                    let 𝒜_S = BP_to_SA_elem bpc;
                    RETURN (product_system_buchi_elem 𝒜_S 𝒜_B) }"

text {* For the product of the automatons, the labels must match: The labels
        of the LTL-Buchi are sets of variables which are true. The function
        bool_list_to_prop_set converts the array of variables from the configuration
        of the boolean program into this format. *}
```

```
fun BP_to_SA_elem :: "bprog × config ⇒ (config, nat set) Buchi_elem"
  where "BP_to_SA_elem (bp, c) = (| Buchi_elem_Δ = λ(pc,s) l.
                                      if l = bool_list_to_prop_set s then
                                        set (BoolProgs.nexts bp (pc,s))
                                      else {} ,
                                    Buchi_elem_ℐ = {c},
                                    Buchi_elem_ℱ = λ_. True |)"

definition bool_list_to_prop_set where
  "bool_list_to_prop_set xs = {i ∈ {0..<length xs}. xs ! i}"
end
```