GSTE to “STE with tag invariants”

introduction

Nowadays the microprocessor industry is still booming, leading to industrial demand for the hardware verification. It is crucial to ensure the correctness and error-tolerance of safety-critical application. Formal method is used to tackle those problems in a mathematical sense, verifying or proving the soundness and completeness of the application. Formal method can be divided to two main parts: model checking and theorem proving.
STE
Generalized Symbolic Trajectory Evaluation was developed by Intel and has been successfully applied into verification of industrial-level hardware system.

term-level GSTE theory

Constants, Variable, Expression and Formula

- A constant is either a natural number or boolean value, which also have corresponding symbolic version. Natural number in our theory consists of two parts: its value and its bit length, while a symbolic natural number consists of its name and its bit length. Each boolean constant carries a truth-value and each boolean symbolic constant just carries a name, which can be identified through Ocaml type constructor.

- there are two kinds of variable: (1) simple identifier, denoted by a string and the type of it ; (2) element of an array, denoted by a variable followed by an expression representing index.

  - A simple expression is either a variable `v` or a constant `c`, in addition we allow using `IteForm`to construct a compound expression, which has the form of `f?e1: e2`.

  - A formula can be atomic formula or a compound formula. An atomic formula can be a constant `Chaos` or an equivalence formula `e1=e2`. Compound formula can be constructed from formulas using logic connectives, including `And, Or, Neg,Imply`.

$$
\begin{array}{llllllll}
c & ::= & \mid n & \mid true & \mid false & \mid symbN & \mid symbB & \\
v & := & \mid a & \mid a[i] & & & & \\
e & := & \mid c & \mid v & \mid f?e_1 : e_2 & & & \\
f & := & \mid Chaos & \mid e_1 = e_2 & \mid !f_1 & \mid f_1 \ \&\& \ f_2 & \mid f_1 \Vert f_2 & \mid f_1 \rightarrow f_2
\end{array}
$$

term-level assertion graph definition

An assertion graph definition:

```
(*types.ml *)
type scalar =
  | IntC of int * int (* value * size *)
  | BoolC of bool
  | SymbIntC of string * int
  | SymbBoolC of string
```

```
type sort =
  | Int of int (* size *)
  | Bool
  | Array of int * sort

type var =
  | Ident of string * sort
  | Para of var * expression
and expression =
  | IVar of var
  | Const of scalar
  | Uif of string * expression list
  | IteForm of formula * expression * expression
and formula =
  | Eqn of expression * expression
  | AndForm of formula * formula
  | Neg of formula
  | OrForm of formula * formula
  | ImplyForm of formula * formula
  | Chaos
```

term-level tag invariant definition

Tag invariant definition:
Define manually the tag invariants of a assertion graph.

transfomr term-level into boolean-level

boolean-level assertion graph in Ocaml

Because boolean theory is a subset of first-order theory, so it is feasible to define bit-level assertion graph in term-level. That is we keep the assertion graph skeleton the same as before and define the assertion graph formula just using following constructors:

```
BoolC
BoolV
AndForm
Neg
IteForm
```

boolean-level tag invariant in Ocaml

For one term-level tag invariant, use SMT/Z3 to obtain all the possible boolean-level tag invariants corresponding with it's term-level form. (undone)
difficulty: plus of an expression(IVar) and a number from read-memory operation and equation between two expression

symbolic trajectory evaluation formula in Ocaml

Define the STE formula in Ocaml, and then it’s convenient to transform boolean-level assertion graph formula into symbolic trajectory evaluation formula in Ocaml and pave the way for transforming AG in Ocaml to AG in Forte.

```
(* trajectory.ml *)
type bVar = Bvariable of string
type bExpr =
    | EVar of bVar
    | EAnd of bExpr * bExpr
    | EOr of bExpr * bExpr
    | ENeg of bExpr

let rec bExpr2FLbExprList be =
    match be with
    | EVar (Bvariable name)-> Printf.sprintf "bvariable \"%s\"" name
    | EAnd (be1, be2) -> "(" ^ (bExpr2FLbExprList be1)^ "bAND" ^(bExpr2FLbExprList be2)^ ")"
    | EOr (be1, be2) -> "(" ^ (bExpr2FLbExprList be1)^ "bOR" ^(bExpr2FLbExprList be2) ^ ")"
    | ENeg be0 -> "(bNOT (" ^ (bExpr2FLbExprList be0) ^"))"

type trajNode = Tnode of string

type trajForm =
    | Is1 of trajNode
    | Is0 of trajNode
    | Next of trajForm
    | Guard of bExpr * trajForm
    | TAndList of trajForm list
    | TChaos

let isb p tnode = TAndList [Guard (p, Is1 tnode); Guard ((ENeg p), Is0 tnode)]
```

STE in Forte

transfrom the assertion graph (AG) defined in Ocaml into complete STE verification program in Forte;
In the meantime, transform tag invariants defined in (STE formula in Ocaml) into Forte-accepted language.

```
(** gsteSpec to forte input AG *)
let toFL gs model_name binNodes=
    let rec trans trajf =
        match trajf with
        | TChaos -> []
```

```
        | Is1 (Tnode name) -> [Printf.sprintf "Is1 \"%s\"" name]
        | Is0 (Tnode name) -> [Printf.sprintf "Is0 \"%s\"" name]
        | Guard (be, tf) -> (
            match (trans tf) with
            |[] -> [Printf.sprintf "Guard (%s) (%s)" (bExpr2FLbExprList be) "Chaos"]
            | f::[] -> [Printf.sprintf "Guard (%s) (%s)" (bExpr2FLbExprList be) f]
            | fs -> [Printf.sprintf "Guard (%s) (%s)" (bExpr2FLbExprList be) "TAndList [" ^ (String.concat "," fs) ^ "]"]
        )
        | TAndList ts ->  List.flatten (List.map (fun f -> trans f) ts)
        | _ -> raise (Invalid_argument "this is for boolean level trajectory formula")
    in
    let main_assertion_graph init_node node_set edge_set =
        match init_node with (Vertex inum) -> Printf.fprintf stdout "let vertexI = Vertex %d;\n" inum ;
        Printf.fprintf stdout "%s" ("let vertexL = [" ^ (String.concat "," (List.map (fun (Vertex i) -> Printf.sprintf "Vertex %d" i) node_set)) ^ "];\n" );
        Printf.fprintf stdout "%s" ("let edgeL = [" ^ (String.concat "," (List.map (fun (Edge ((Vertex f),(Vertex t))) -> Printf.sprintf "Edge (Vertex %d) (Vertex %d)" f t) edge_set)) ^ "];\n");
    in
    let ant_function init_node node_set edge_set ants =
        let ants_traj e =
            let term_f = ants e in
            let bit_f = termForm2bitForm term_f in
            let traj_f = bitForm2trajForm bit_f in
            let add_myclk fs =
                match fs with
                | [] -> "TAndList []"
                | s -> "TAndList ["^ (String.concat "," (s@[Printf.sprintf "Is0 \"%s\"" visCLKname ; Printf.sprintf "Next (Is1 \"%s\")" visCLKname])) ^"]"
            in
            add_myclk (trans traj_f)
        in
        Printf.fprintf stdout "%s" (
"let ant aEdge =
    val (Edge (Vertex from) (Vertex to)) = aEdge
    in
"^      (
            let items = List.map (fun e -> (
                                            match e with
                                            |Edge ((Vertex f),(Vertex t)) -> Printf.sprintf "((from = %d) AND (to = %d)) => %s " f t (ants_traj e)
                                        )
                            ) edge_set
            in
            let cases = String.concat "\n\t| " items in
```

```
            let body = cases ^ "\n\t| error \"In cons: missing case\"" in
            Printf.sprintf "\t%s\n;\n\n" body
        )
    )
    in
    let cons_function init_node node_set edge_set cons =
        let cons_traj e =
            let term_f = cons e in
            let bit_f = termForm2bitForm term_f in
            let traj_f = bitForm2trajForm bit_f in
            let add_tandlist ts =
                match ts with
                | [] -> "TAndList []"
                | s -> "TAndList ["^ (String.concat "," s) ^"]"
            in
            add_tandlist (trans traj_f)
        in
        Printf.fprintf stdout "%s" (
"let cons aEdge =
    val (Edge (Vertex from) (Vertex to)) = aEdge
    in
"^      (
            let items = List.map (fun e -> (
                                            match e with
                                            |Edge ((Vertex f),(Vertex t)) -> Printf.sprintf "((from = %d) AND (to = %d)) => %s " f t (cons_traj e)
                                        )
                            ) edge_set
            in
            let cases = String.concat "\n\t| " items in
            let body = cases ^ "\n\t| error \"In cons: missing case\"" in
            Printf.sprintf "\t%s\n;\n\n" body
        )
    )
    in
    let binNodesPart binNodes =
        let ivar2str iv =
            match iv with
            | IVar (Ident (str, Int n)) -> List.map (fun i -> formatMapVIS ~axis1:i str) (upt 0 (n-1))
            | IVar (Ident (str, Bool)) -> [formatMapVIS str]
            | _ -> raise (Invalid_argument "In toFL binNodesPart: sry la")
        in
        match binNodes with
        | []  -> "[]"
        | bns -> (
            let strOfbns = (List.flatten (List.map (fun nd -> ivar2str nd) bns)) in
            let addQuoteMark = List.map (fun sob -> "\""^sob^"\"") strOfbns in
            "[" ^ (String.concat "," addQuoteMark) ^"]"
        )
```

```
    in
    match gs with Graph (init_node , node_set, edge_set, ants, cons) -> (
        Printf.fprintf stdout
"
let ckt = load_exe \"%s.exe\";
load \"gsteSymReduce.fl\";
loadModel ckt;
" model_name;
        main_assertion_graph init_node node_set edge_set;
        ant_function init_node node_set edge_set ants;
        cons_function init_node node_set edge_set cons ;
        Printf.fprintf stdout
"
let mainGoal = Goal [] (TGraph (Graph vertexL vertexI edgeL (Edge2Form ant) (Edge2Form cons)));
let binNodes = %s;
lemma \"lemmaTMain\" mainGoal;
    by (gsteSymbSim binNodes);
done 0;
quit;
"  (binNodesPart binNodes)
    )
```

GSTE to “STE with tag-invariants”

toFL : term-level (assertion graph & tag-invariants) to Forte-accepted input language:
At term-level: using SMT to solve the limited equation to get all possible instantiated invariants
then transform tag-invariants sets (instantiated) into boolean-level invariants sets,
then transform boolean-level invariants set into Forte-accepted tag invariants (defined in gsteSymReduce.fl)

STE with tag-invariants verification algorithm

```
## main function
result = true;
for edge in edgeL:
    result = result and check(edge)
return result
```

```
## check function
# input : edge
# output : boolean
val (Edge from to) = edge
val (Vertex n1) = from
val (Vertex n2) = to
ants_set = [ TAndList ((ant edge) : x) for x in (tag from) ]
cons_set = [ TAndList ((cons edge): (map(Next, x))) for x in (tag to)]
```

```
    return check_next_ant(ants_set, cons_set)
```

```
## check_next_ant function
# input : antss: an antecedent set , contss: a consequent set
# ouput : boolean
for ant in antss:
    if check_helper(conss, ant):
        continue
    else:
        return false
```

```
## check_helper function
# input : conss: a consequent set, ant: an antecedent
# output : boolean
result = false
for con in conss:
    if steSymbSimGoalfDirect(ant, con):
        result = true
        break
return result
```

case analysis

ring buffer fifo

Considering a ring buffer fifo circuit with concrete parameters, the concrete parameters are following:

```
let depth = 4  (* fifo depth *)
let last = depth - 1 (* fifo maximum index value*)
let data_length = 2 (* element width of fifo *)
let index_length = 2 (* the variable (indicating index) 's width  *)
```

we can define the main assertion graph's skeleton as follow:

```
(** main assertion graph *)
let vertexI = Vertex 0
let vertexL = vertexI :: ((Vertex 1):: (List.map (fun i -> Vertex i) (upt 3 (2*last+4))))
(** odd-vertex selfloop edge, odd-vertex bidirection edge, odd-vertex to even-vertex edge, even-vertex backward edge , Vertex 4 to Vertex 1 edge*)
let edgeL = [Edge (vertexI,(Vertex 1))] @
```

```
             (List.map (fun i -> Edge (Vertex (2*i+1), Vertex (2*i+1)))  (upt 0 depth))@
             (List.map (fun i -> Edge (Vertex (2*i-1), Vertex (2*i+1)))  (upt 1 depth))@
             (List.map (fun i -> Edge (Vertex (2*i+1), Vertex (2*i-1)))  (dwt depth 1))@
             (List.map (fun i -> Edge (Vertex (2*i-1), Vertex (2*i+2)))  (upt 1 depth))@
             (List.map (fun i -> Edge (Vertex (2*i+2), Vertex (2*i)))    (dwt depth 2))@
             [Edge ((Vertex 4), (Vertex 1))]
```

and the antecedent formula set and the consequent formula set can be defined like following:

```
(** actions of assertion graph *)
let rst      : expression = IVar (Ident ("rst", Bool))
let push     : expression = IVar (Ident ("push", Bool))
let pop      : expression = IVar (Ident ("pop", Bool))
let empty    : expression = IVar (Ident ("empty", Bool))
let full     : expression = IVar (Ident ("full", Bool))
let dataIn   : expression = IVar (Ident ("dataIn", Int data_length))
let dataOut  : expression = IVar (Ident ("dataOut", Int data_length))
let low      : expression = Const (BoolC false)
let high     : expression = Const (BoolC true)
let readDataIn  :  expression = Const (IntC (1, data_length))
let symbolDataOut :  expression = Const (SymbIntC ("dout", data_length))

let rstFormula = Eqn (rst, high)
let nrstFormula = Eqn (rst, low)
let pushFormula =  AndForm (nrstFormula, AndForm (Eqn (push, high), Eqn (pop, low)))
let popFormula  = AndForm (nrstFormula, AndForm (Eqn (push, low), Eqn (pop, high)))
let noPopPushFormula = AndForm (nrstFormula, AndForm(Eqn (push,low), Eqn (pop, low)))
let fullFormula = Eqn (full, high)
let noFullFormula = Eqn (full, low)
let emptyFormula = Eqn (empty, high)
let noEmptyFormula = Eqn (empty, low)

let pushData data = AndForm (pushFormula, Eqn (dataIn, data))
let popData data = Eqn (dataOut, data)

let antOfRbFIFO aEdge =
  let f = nodeToInt (source aEdge) in
  let t = nodeToInt (sink aEdge)   in
  (
    if( f == 0) then rstFormula
    else (
        if (f mod 2 == 1 ) then (
```

```
            if (f = t) then noPopPushFormula
            else if ((f + 2) == t) then pushFormula
            else if ( f == (t+2)) then popFormula
            else pushData readDataIn
        )else popFormula
    )
  )

let consOfRbFIFO aEdge =
    let f = nodeToInt (source aEdge) in
    let t = nodeToInt (sink aEdge)   in
    (
        if ((f mod 2 == 1)&&(t mod 2 == 1)) then (
            if (f == 1) then AndForm (emptyFormula, noFullFormula)
            else if (f == (2*depth+1)) then AndForm (noEmptyFormula, fullFormula)
            else AndForm (noEmptyFormula, noFullFormula)
        )else (
            if ((f == 4) && (t == 1)) then popData readDataIn
            else if (f == (2*depth+2))  then AndForm (noEmptyFormula, fullFormula)
            else if (f == 1) then AndForm (emptyFormula, noFullFormula)
            else if (f == 0) then Chaos
            else if (t == 4) then Chaos
            else AndForm (noEmptyFormula, noFullFormula)
        )
    )
```

A typical GSTE Assertion graph can be constructed by using following statement:

```
let rbfifoGsteSpec = Graph (vertexI, vertexL, edgeL, antOfRbFIFO, consOfRbFIFO)
```

Next step it is about to construct the tag invariants set and transfrom the (GSTE assertion graph and tag invariants) into Forte-accpted input language by using `toFL` function

memory