

GSTE to “STE with tag invariants”

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`.

c	$::=$	$ n$	$ true$	$ false$	$ symbN$	$ symbB$
v	$::=$	$ a$	$ a[i]$			
e	$::=$	$ c$	$ v$	$ f?e_1 : e_2$		
f	$::=$	$ Chaos$	$ e_1 = e_2$	$!f_1$	$ f_1 \ \&\& \ f_2$	$ f_1 \ \ f_2 \quad f_1 \rightarrow f_2$

term-level assertion graph definition

```
1. (*b_types.ml *)
2. type scalar =
3.   | IntC of int * int (* value * size *)
4.   | BoolC of bool
5.   | SymbIntC of string * int
6.   | SymbBoolC of string
7.
8. type sort =
9.   | Int of int (* size *)
10.  | Bool
11.  | Array of int * sort
12.
13. type var =
14.   | Ident of string * sort
15.   | Para of var * expression
16. and expression =
17.   | IVar of var
18.   | Const of scalar
19.   | IteForm of formula * expression * expression
```

```

20. and formula =
21.   | Eqn of expression * expression
22.   | AndForm of formula * formula
23.   | Neg of formula
24.   | OrForm of formula * formula
25.   | ImplyForm of formula * formula
26.   | Chaos

```

term-level tag invariant definition (non auto-generated)

Define manually the tag invariants of a assertion graph.

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:

```

1. BoolC
2. BoolV
3. AndForm
4. Neg

```

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.

```

1. (* trajectory.ml *)
2. type bVar = Bvariable of string
3. type bExpr =
4.   | EVar of bVar
5.   | EAnd of bExpr * bExpr
6.   | EOr of bExpr * bExpr
7.   | ENeg of bExpr
8.
9. let rec bExpr2FLbExprList be =
10.   match be with
11.   | EVar (Bvariable name)-> Printf.sprintf "bvariable \"%s\"" name

```

```

12.     | EAnd (be1, be2) -> "(" ^ (bExpr2FLbExprList be1)^ "bAND" ^ (bExpr2FLbExprList be2)^ ")"
13.     | EOr (be1, be2) -> "(" ^ (bExpr2FLbExprList be1)^ "bOR" ^ (bExpr2FLbExprList be2) ^ ")"
14.     | ENeg be0 -> "(bNOT (" ^ (bExpr2FLbExprList be0) ^"))"
15.
16. type trajNode = Tnode of string
17.
18. type trajForm =
19.     | Is1 of trajNode
20.     | Is0 of trajNode
21.     | Next of trajForm
22.     | Guard of bExpr * trajForm
23.     | TAndList of trajForm list
24.     | TChaos
25.
26. let isb p tnode = TAndList [Guard (p, Is1 tnode); Guard ((ENeg p), Is0 tnode)]

```

STE in Forte

transform 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.

```

1. (** gsteSpec to forte input AG *)
2. let toFL gs model_name binNodes=
3.     let rec trans trajf =
4.         match trajf with
5.         | TChaos -> []
6.         | Is1 (Tnode name) -> [Printf.sprintf "Is1 \"%s\" \"\" name]
7.         | Is0 (Tnode name) -> [Printf.sprintf "Is0 \"%s\" \"\" name]
8.         | Guard (be, tf) -> (
9.             match (trans tf) with
10.            | [] -> [Printf.sprintf "Guard (%s) (%s)" (bExpr2FLbExprList be)
"Chaos"]
11.            | f::[] -> [Printf.sprintf "Guard (%s) (%s)" (bExpr2FLbExprList be) f]
12.            | fs -> [Printf.sprintf "Guard (%s) (%s)" (bExpr2FLbExprList be)
"TAndList [" ^ (String.concat "," fs) ^ "]" ]
13.            )
14.         | TAndList ts -> List.flatten (List.map (fun f -> trans f) ts)
15.         | _ -> raise (Invalid_argument "this is for boolean level trajectory formula")
16.     in
17.     let main_assertion_graph init_node node_set edge_set =
18.         match init_node with (Vertex inum) -> Printf.fprintf stdout "let vertexI = Vertex %d;\n" inum ;

```

```

19.         Printf.fprintf stdout "%s" ("let vertexL = [" ^ (String.concat ","
(List.map (fun (Vertex i) -> Printf.sprintf "Vertex %d" i) node_set)) ^
"];\\n" );
20.         Printf.fprintf stdout "%s" ("let edgeL = [" ^ (String.concat "," (Li
st.map (fun (Edge ((Vertex f),(Vertex t))) -> Printf.sprintf "Edge (Vertex %
d) (Vertex %d)" f t) edge_set)) ^ "];\\n");
21.     in
22.     let ant_function init_node node_set edge_set ants =
23.         let ants_traj e =
24.             let term_f = ants e in
25.             let bit_f = termForm2bitForm term_f in
26.             let traj_f = bitForm2trajForm bit_f in
27.             let add_myclk fs =
28.                 match fs with
29.                 | [] -> "TAndList []"
30.                 | s -> "TAndList ["^ (String.concat "," (s@[Printf.sprintf
"Is0 \\\"%s\\\"" visCLKname ; Printf.sprintf "Next (Is1 \\\"%s\\\"" visCLKname)))
^"]"
31.             in
32.             add_myclk (trans traj_f)
33.         in
34.         Printf.fprintf stdout "%s" (
35. "let ant aEdge =
36.     val (Edge (Vertex from) (Vertex to)) = aEdge
37.     in
38.     "^
39.         let items = List.map (fun e -> (
40.                                     match e with
41.                                     | Edge ((Vertex f),(Vertex t)) ->
Printf.sprintf "((from = %d) AND (to = %d)) => %s " f t (ants_traj e)
42.                                     )
43.             ) edge_set
44.         in
45.             let cases = String.concat "\\n\\t| " items in
46.             let body = cases ^ "\\n\\t| error \\\"In cons: missing case\\\"" in
47.             Printf.sprintf "\\t%s\\n;\\n\\n" body
48.         )
49.     )
50.     in
51.     let cons_function init_node node_set edge_set cons =
52.         let cons_traj e =
53.             let term_f = cons e in
54.             let bit_f = termForm2bitForm term_f in
55.             let traj_f = bitForm2trajForm bit_f in
56.             let add_tandlist ts =
57.                 match ts with
58.                 | [] -> "TAndList []"
59.                 | s -> "TAndList ["^ (String.concat "," s) ^"]"
60.             in
61.             add_tandlist (trans traj_f)
62.         in
63.         Printf.fprintf stdout "%s" (

```

```

64. "let cons aEdge =
65.     val (Edge (Vertex from) (Vertex to)) = aEdge
66.     in
67.     "^      (
68.         let items = List.map (fun e -> (
69.             match e with
70.             | Edge ((Vertex f),(Vertex t)) ->
        Printf.sprintf "((from = %d) AND (to = %d)) => %s " f t (cons_traj e)
71.             )
72.         ) edge_set
73.         in
74.         let cases = String.concat "\n\t| " items in
75.         let body = cases ^ "\n\t| error \"In cons: missing case\"" in
76.         Printf.sprintf "\t%s\n;\n\n" body
77.     )
78. )
79. in
80. let binNodesPart binNodes =
81.     let ivar2str iv =
82.         match iv with
83.         | IVar (Ident (str, Int n)) -> List.map (fun i -> formatMapVIS ~
axis1:i str) (upt 0 (n-1))
84.         | IVar (Ident (str, Bool)) -> [formatMapVIS str]
85.         | _ -> raise (Invalid_argument "In toFL binNodesPart: sry la")
86.     in
87.     match binNodes with
88.     | [] -> "[]"
89.     | bns -> (
90.         let strOfbns = (List.flatten (List.map (fun nd -> ivar2str nd) b
ns)) in
91.         let addQuoteMark = List.map (fun sob -> "\""^sob^"") strOfbns
92.         in
93.         "[" ^ (String.concat "," addQuoteMark) ^ "]"
94.     )
95. in
96. match gs with Graph (init_node , node_set, edge_set, ants, cons) -> (
97.     Printf.fprintf stdout
98.     "
99.     let ckt = load_exe \"%s.exe\";
100.     load \"gsteSymReduce.fl\";
101.     loadModel ckt;
102.     " model_name;
103.     main_assertion_graph init_node node_set edge_set;
104.     ant_function init_node node_set edge_set ants;
105.     cons_function init_node node_set edge_set cons ;
106.     Printf.fprintf stdout
107.     "
108.     let mainGoal = Goal [] (TGraph (Graph vertexL vertexI edgeL (Edge2Form ant)
(Edge2Form cons)));
109.     let binNodes = %s;
110.     lemma \"lemmaTMain\" mainGoal;
111.     by (gsteSymbSim binNodes);

```

```

111.  done 0;
112.  quit;
113.  " (binNodesPart binNodes)
114.  )

```

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

```

1.  ## main function
2.  result = true;
3.  for edge in edgeL:
4.      result = result and check(edge)
5.  return result

```

```

1.  ## check function
2.  # input : edge
3.  # output : boolean
4.  val (Edge from to) = edge
5.  val (Vertex n1) = from
6.  val (Vertex n2) = to
7.  ants_set = [ TAndList ((ant edge) : x) for x in (tag from) ]
8.  cons_set = [ TAndList ((cons edge): (map(Next, x))) for x in (tag to)]
9.  return check_next_ant(ants_set, cons_set)

```

```

1.  ## check_next_ant function
2.  # input : antss: an antecedent set , contss: a consequent set
3.  # ouput : boolean
4.  for ant in antss:
5.      if check_helper(conss, ant):
6.          continue
7.      else:
8.          return false

```

```

1.  ## check_helper function
2.  # input : conss: a consequent set, ant: an antecedent
3.  # output : boolean

```

```

4.     result = false
5.     for con in conss:
6.         if steSymbSimGoalFDirect(ant, con):
7.             result = true
8.             break
9.     return result

```

case analysis

ring buffer fifo

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

```

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

```

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

```

1. (** main assertion graph *)
2. let vertexI = Vertex 0
3. let vertexL = vertexI :: ((Vertex 1):: (List.map (fun i -> Vertex i) (upt 3
    (2*last+4))))
4. (** odd-vertex selfloop edge, odd-vertex bidirection edge, odd-vertex to eve
    n-vertex edge, even-vertex backward edge , Vertex 4 to Vertex 1 edge*)
5. let edgeL = [Edge (vertexI,(Vertex 1))] @
6.     (List.map (fun i -> Edge (Vertex (2*i+1), Vertex (2*i+1))) (upt
    0 depth))@
7.     (List.map (fun i -> Edge (Vertex (2*i-1), Vertex (2*i+1))) (upt
    1 depth))@
8.     (List.map (fun i -> Edge (Vertex (2*i+1), Vertex (2*i-1))) (dwt
    depth 1))@
9.     (List.map (fun i -> Edge (Vertex (2*i-1), Vertex (2*i+2))) (upt
    1 depth))@
10.    (List.map (fun i -> Edge (Vertex (2*i+2), Vertex (2*i))) (dwt
    depth 2))@
11.    [Edge ((Vertex 4), (Vertex 1))]

```

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

```

1. (** actions of assertion graph *)
2. let rst      : expression = IVar (Ident ("rst", Bool))
3. let push     : expression = IVar (Ident ("push", Bool))
4. let pop      : expression = IVar (Ident ("pop", Bool))

```

```

5. let empty      : expression = IVar (Ident ("empty", Bool))
6. let full       : expression = IVar (Ident ("full", Bool))
7. let dataIn     : expression = IVar (Ident ("dataIn", Int data_length))
8. let dataOut    : expression = IVar (Ident ("dataOut", Int data_length))
9. let low        : expression = Const (BoolC false)
10. let high       : expression = Const (BoolC true)
11. let readDataIn : expression = Const (IntC (1, data_length))
12. let symbolDataOut : expression = Const (SymbIntC ("dout", data_length))
13.
14. let rstFormula = Eqn (rst, high)
15. let nrstFormula = Eqn (rst, low)
16. let pushFormula = AndForm (nrstFormula, AndForm (Eqn (push, high), Eqn (pop, low)))
17. let popFormula = AndForm (nrstFormula, AndForm (Eqn (push, low), Eqn (pop, high)))
18. let noPopPushFormula = AndForm (nrstFormula, AndForm (Eqn (push, low), Eqn (pop, low)))
19. let fullFormula = Eqn (full, high)
20. let noFullFormula = Eqn (full, low)
21. let emptyFormula = Eqn (empty, high)
22. let noEmptyFormula = Eqn (empty, low)
23.
24. let pushData data = AndForm (pushFormula, Eqn (dataIn, data))
25. let popData data = Eqn (dataOut, data)
26.
27. let antOfRbFIFO aEdge =
28.   let f = nodeToInt (source aEdge) in
29.   let t = nodeToInt (sink aEdge)   in
30.   (
31.     if (f == 0) then rstFormula
32.     else (
33.       if (f mod 2 == 1) then (
34.         if (f = t) then noPopPushFormula
35.         else if ((f + 2) == t) then pushFormula
36.         else if (f == (t+2)) then popFormula
37.         else pushData readDataIn
38.       ) else popFormula
39.     )
40.   )
41.
42. let consOfRbFIFO aEdge =
43.   let f = nodeToInt (source aEdge) in
44.   let t = nodeToInt (sink aEdge)   in
45.   (
46.     if ((f mod 2 == 1) && (t mod 2 == 1)) then (
47.       if (f == 1) then AndForm (emptyFormula, noFullFormula)
48.       else if (f == (2*depth+1)) then AndForm (noEmptyFormula, fullFormula)
49.       else AndForm (noEmptyFormula, noFullFormula)
50.     ) else (
51.       if ((f == 4) && (t == 1)) then popData readDataIn

```



```

52.         else if (f == (2*depth+2)) then AndForm (noEmptyFormula, fullFo
        rmula)
53.         else if (f == 1) then AndForm (emptyFormula, noFullFormula)
54.         else if (f == 0) then Chaos
55.         else if (t == 4) then Chaos
56.         else AndForm (noEmptyFormula, noFullFormula)
57.     )
58. )

```

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

```

1. let rbfifoGsteSpec = Graph (vertexI, vertexL, edgeL, antOfRbFIFO, consOfRbFI
    FO)

```

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
