Validating the Translation of an Industrial Optimizing Compiler

Raya Leviathan

Joint work with:
Prof. Amir Pnueli
Ira gordin
Weizmann Institute of Science
IST – Project SafeAir-II
Minerva Center for Verification of Reactive systems
NFS grant CCR-0205571

Outline

- Translation Validation of Optimized Code
- Synchronous Programs
- Theoretical framework
- MCVT - Machine code Validation tool
- Experimental results
- Our contribution

Guarantee the Correct Functioning of a Compiler

The importance of formally proving the correctness of safety critical software system is well recognized.

It should also be verified that the correctness of the high level implementation is not impaired by the compiler.

In safety critical applications standards and regulations require that every compiler be certified.

Or translation is manually checked and verified.

Translation Validation

~~Formally verifying full fledge compiler~~

Translation Validation

Evolution of the translator is not frozen after verification.

Significantly cheaper.

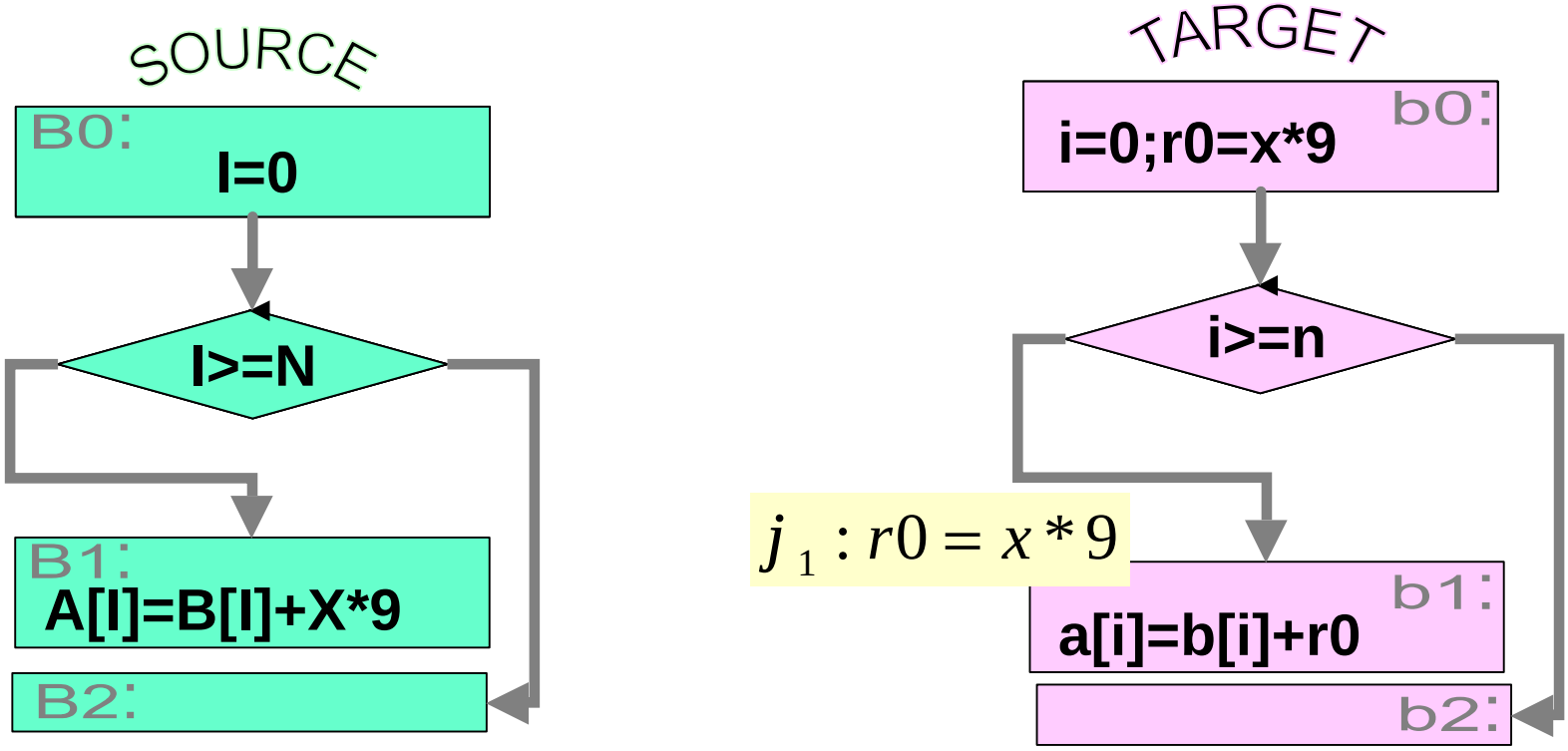
Provides an independent cross check.

The VALIDATE Procedure
[Pnueli & al 00,01,02]

- Construct control mapping - k
- For each basic block B_i form an invariant j_i which holds at the beginning of the block.
- Construct a mapping a from source to target variables: $a : V_S \rightarrow Exp(V_T)$
- For each basic block B_i construct a verification condition:

$$C_i : j_i \wedge a \wedge a' \wedge r_i^T \rightarrow r_{k(i)}^S \wedge \left(\bigvee_{j \in succ(B_j)} (j_j \wedge pc' = B_j) \right)$$

Validating Translation of General C



CV_{b_1} :

$$pc = b1 \wedge pc' = b1 \wedge a' = a \;\; with : (a'[i] = b[i] + r0) \wedge i' = i + 1 \wedge$$

$$b' = b \wedge r0' = r0 \wedge x' = x \wedge A = a \wedge A' = a' \wedge B = b \wedge B' = b' \wedge i = I \wedge I' = i' \wedge$$

$$x = X \wedge ... j_1 \wedge 0 < i \wedge i' < n \longrightarrow$$

$$PC = b1 \wedge A' = A \;\; with : (A'[I] = B[I] + X * 9) \wedge B' = B \wedge I' = I + 1 \wedge I > 0 \wedge ... \wedge j_1'$$

Validated Optimizations

1. Constant folding.
2. Common sub-expression elimination.
3. Dead Code elimination.
4. Strength reduction.
5. Loop invariant code motion.
6. Loop unrolling.
7. Software Pipelining.

The Problem

Source Synchronous C Program

Optimizing
Compiler

Target Synchronous Program (executable)

Synchronous programs properties:

- Contains no loops, except one main external loop
- Always terminates

The Source Synchronous Programs

Produced from high level design language: SCADE.

A SCADE program is composed of nodes.

A node is a network of operators.

The observable variables of a node are defined by the node formal interface.

A program is a collection of nodes, which are activated by an external loop.

Each node is separately compiled into Synchronous C.

Synchronous C

- Definitions
- Assignment: v:=exp(V);
- If () {…} else {…}
- Compositions of statements

Example

```
int a;
int b;
void P_source( )
{
    if (a > 5)   a= 5;
    b=a+4;
}
```

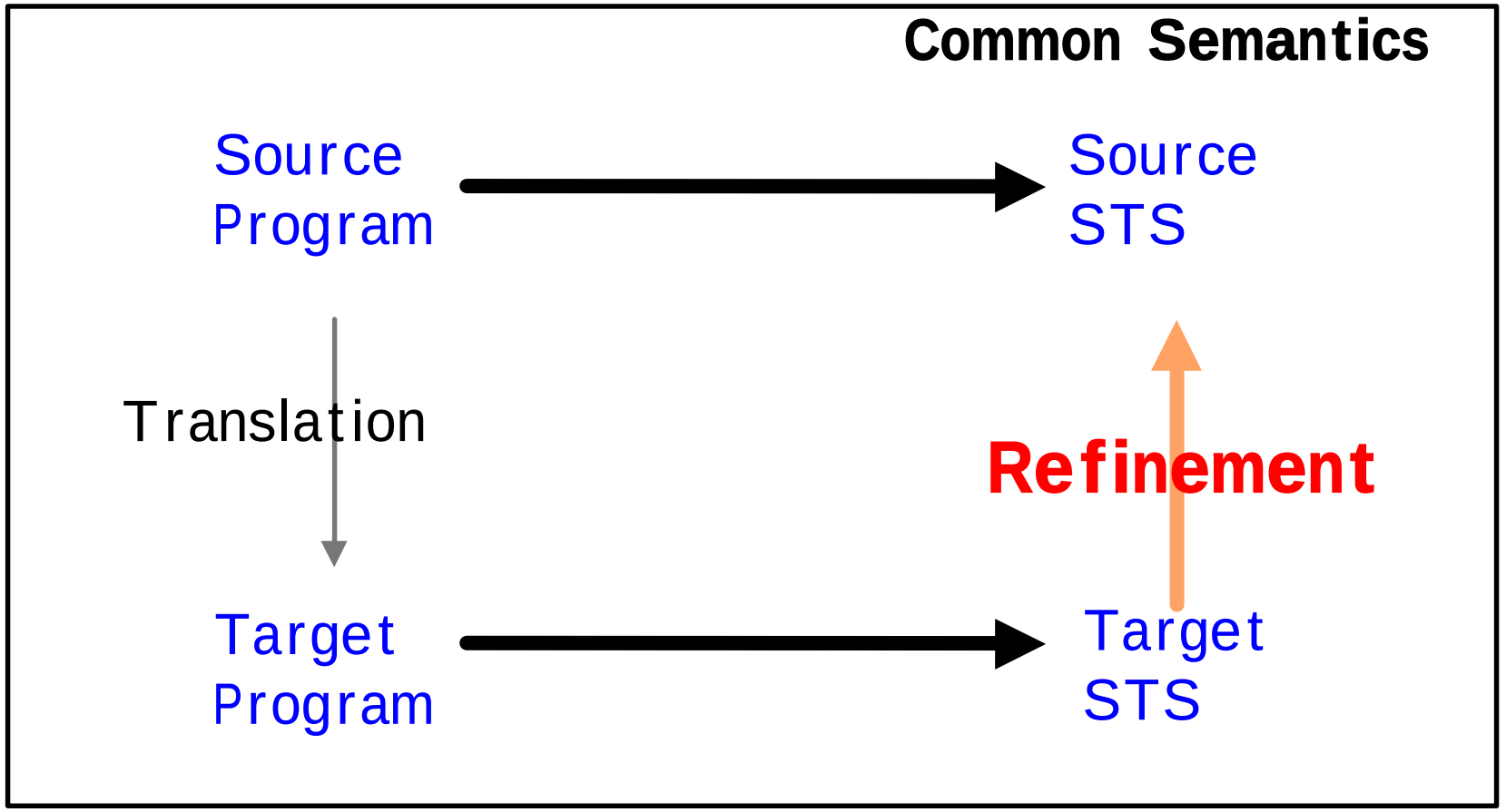
The Target Synchronous Program
PowerPC Machine Code

- Assignment
- Branch
- Conditional branch
- Composition of statements

Example

```
<0>:    lwz     r10,0(r3)
<4>:    lwz     r4,4(r3)
<8>:    cmpi    0,r10,5
<12>:   addi    r5,r0,3
<16>:   bc      4,1,<24>
<20>:   addi    r5,r4,5
<24>:   cmpi    0,r4,4
<28>:   bc      4,1,<40>
<32>:   stw     r5,4(r3)
<36>:   br      <44>
<40>:   stw     r5,0(r3)
<44>:   exit
```

Validation of a Translation



State Transformer Systems

A State transformer system (STS) $S : \langle V , O , F \rangle$

$V = \{v_0, \ldots, v_k\};$ Variables

$we\ use : \sum, s \in \sum, and\ v \in V\ , s[v]$ States

$O \subseteq V;\ \ s \Downarrow_O$ Observables

$F : \sum \mapsto \sum; \quad F : \langle F_i, \ldots, F_k \rangle$

$F_i : \sum \mapsto D_i; \quad i = 0, \ldots, k$ State Transformer

$s = (s_0, s_1), is\ a$ Computation $, where\ s_1 = F(s_0)$

$s \Downarrow_O = (o_0, o_1), is\ an$ Observation of a computation

$where\ \ o_0 = \ \ s_0 \Downarrow_O, \quad o_1 = s_1 \Downarrow_O, (s_0, s_1)\ is\ a\ computation.$

Transformer Composition

$$F_{s_1;s_2} = F^1 \circ F^2$$

$$F^1 \circ F^2 = lV \cdot F^2(F^1(V))$$

$$If\ F = F^1 \circ F^2\ then\ F_j(V) = F_j{}^2(F^1(V)),\ for\ j = 1,\ldots,k$$

For example:

$$F_{S_1} : (a, b+1)$$

$$F_{S_2} : (2*b, a+b+1)$$

$$F_{S_1;S_2} = F_{S_1} \circ F_{S_2} : (2*(b+1), a+b+2)$$

Refinement Between STS's

A correct translation needs only to ensure the preservation of the observable behavior.

$$S^S = \langle V^S, O^S, F^S \rangle, \quad S^T = \langle V^T, O^T, F^T \rangle$$

Let $R \subseteq \Sigma^S \Downarrow_O \times \Sigma^T \Downarrow_O$

be a relation between source and target system observations, then S^T is R-Refinement of S^S

If for every source and target states: s, t

$R(s \Downarrow_O, t \Downarrow_O)$ implies $R(F_O^S(s), F_O^T(t))$

Valid Translation

A translation is valid if there exists a refinement relation between the source and the target systems.

We restrict the refinement relation to be a conjunction of equalities.

The COMPARE Procedure

$$S^S = \langle V^S, O^S, F^S \rangle, S^T = \langle V^T, O^T, F^T \rangle$$

$$O^S = \{V_1, \ldots, V_k\}; \; O^T = \{v_1, \ldots, v_k\};$$

1. Construct a 1-1 mapping of source to target observable variable

$$a : V_1 = v_1 \wedge \ldots \wedge V_k = v_k$$

2. For Each source observable variable V_i, construct a verification condition

$$VC_i : \; a \Rightarrow F_i^S(V^S) = F_i^T(V^T)$$

3. Establish the validity of all verification conditions.

STS Semantics of Synchronous C

For a synchronous C program, with the variables: $V=\{v_1,..,v_k\}$

- the set of program variables.
- the global variables.

F^P :

$\mathrm{S}:\mathrm{v_i}=\mathrm{E};$	$F_S(V)=(v_1,\ldots,v_{i-1},E,v_{i+1},\ldots,v_k)$
$\mathrm{S}:\mathrm{if}\ (\mathrm{C})\ \mathrm{then}\ \mathrm{S}^1\ \mathrm{else}\ \mathrm{S}^2;$	$F_S(V)=(ite(C,F_1^1,F_1^2),\ldots,ite(C,F_k^1,F_k^2))$
$\mathrm{S}:\mathrm{S}^1,\ldots,\mathrm{S}^2;$	$F_S(V)=F_{S^1}\circ\cdots\circ F_{S^k}$

STS of Machine Code Programs: ANNOTATION Algorithm

We consider the machine code program:

$$P = inst_1, \ldots, inst_n$$

We apply Forward Analysis, by inductively constructing transformers: $F^1, \ldots, F^{n+1}$

That corresponds to program prefixes:

$$P_i = inst_1, \ldots, inst_{i-1}$$

and annotations: $Ann[1], \ldots, Ann[n]$

Which expresses the condition under which instruction $inst_i$ is executed.

The ANNOTATION Algorithm

1. Initially: $Ann[1] = true, Ann[2] = \ldots = Ann[n] = false, F^1 : lV.V$

2. For $i = 1, \ldots, n$ examine $inst_i$:

   a. If $inst_i$ is an assignment $v_i = E$, then do:

   $Ann[i+1] := Ann[i+1] \vee Ann[i]$

   $F^{i+1} := F^i \circ ?V.(v_1, \ldots, v_{j-1}, ite(Ann[i], E, v_j), v_{j+1}, \ldots, v_k)$

   b. If $inst_i$ is a conditinal branch *branch (C)* j, then do:

   $Ann[i+1] := Ann[i+1] \vee Ann[i] \wedge \neg C(F^i(V))$

   $Ann[j] := Ann[j] \vee Ann[i] \wedge C(F^i(V))$

   $F^{i+1} := F^i$

   c. If $inst_i$ is a branch instruction: *branch* j, then do:

   $Ann[j] := Ann[j] \vee Ann[i]$

   $F^{i+1} := F^i$

finally, F is taken to be F^{n+1}

Optimizing the construction of the target state transformer

- Compact expression representation
- Formula's simplifier
- Code pattern recognizer

Data Mapping

- Mapping is extracted from the standard debug information.
- Only observable variables are interesting.
- Local variables are not part of the transformer function produced for the observable variables.

One Verification Condition for Each Observable Variable

Example:

```
int A,B;
Main()
 { int i=A;
 A:=B+1;
if (i>5) B:=1 else B:=2; }
```

$$F^{P_S} : l(A, B) \cdot ((B + 1), ite(A > 5, 1, 2))$$

The Target_Machine Code

```
<0>:    lwz      r3,0(r13)
<4>:    lwz      r11,4(r13)
<8>:    cmpi      0,r3,5
<12>:   addi      r11,r11,1
<16>:   stw       r11,4(r13)
<20>:   bc        4,1,<32>
<24>:   addi      r12,r0,1
<28>:   b         <36>
<32>:   addi      r12,r0,2
<36>:   stw       r12,0(r13)
<40>:   addi      r3,r0,0
<44>:   bclr      20,0
```

Cont'

$$F^{P_S} : ?(A, B) \cdot ((B + 1), ite(A > 5, 1, 2))$$

$$F^{P_T} :$$

$$l(mem_0, mem_4) \cdot (mem_4 + 1, ite(\neg(mem_0 \leq 5), 1, 2))$$

Mapping: $a : A = mem_0 \wedge B = mem_4$

Verification conditions:

$$VC1 : A = mem_0 \wedge B = mem_4 \rightarrow mem_4 + 1 = B + 1$$

$$VC2 : A = mem_0 \wedge B = mem_4 \rightarrow ite(\neg(mem_0 \leq 5), 1, 2) = ite(A > 5, 1, 2)$$

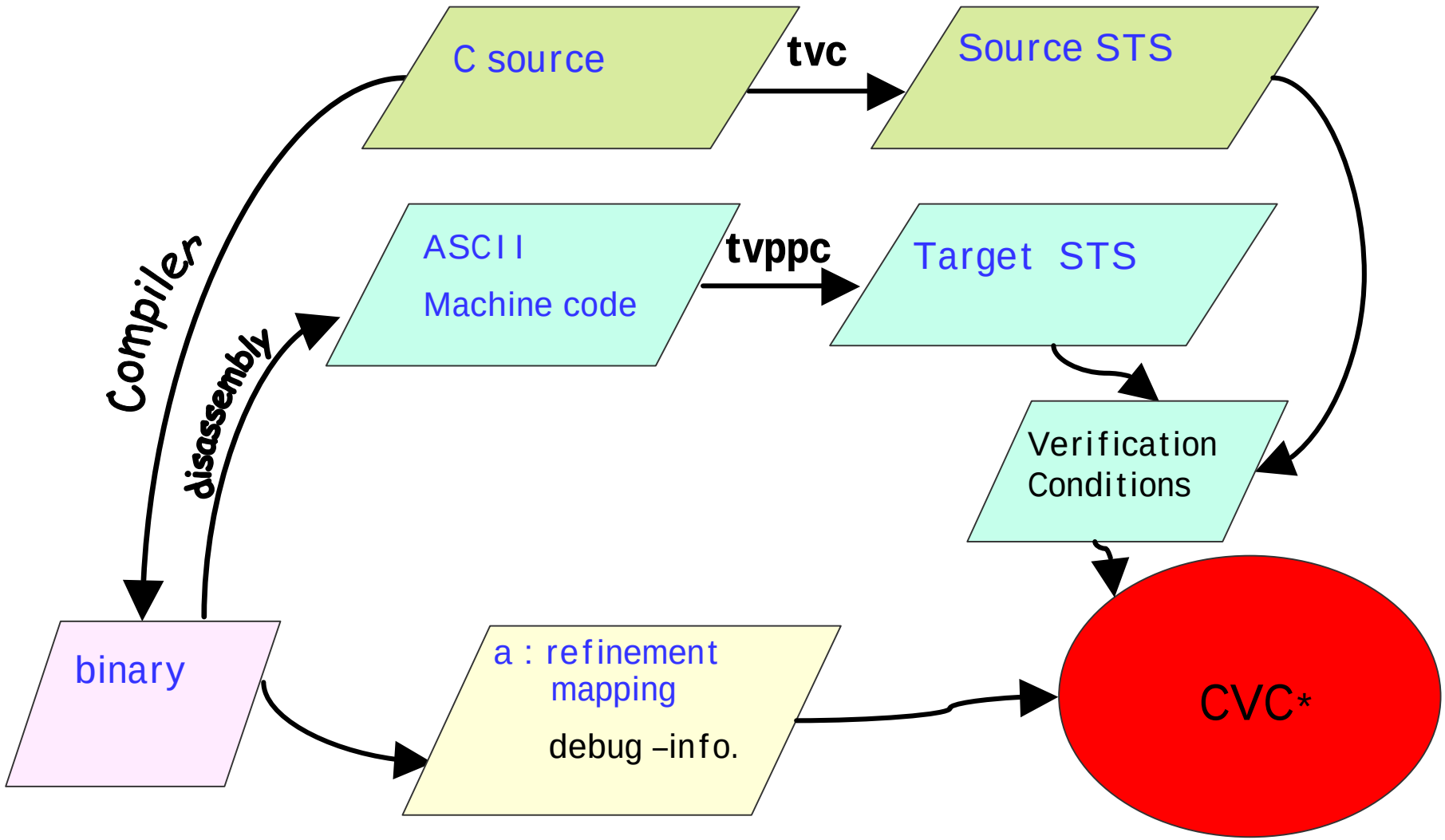
Verifying the Verification conditions

Each verification condition is verified separately by:

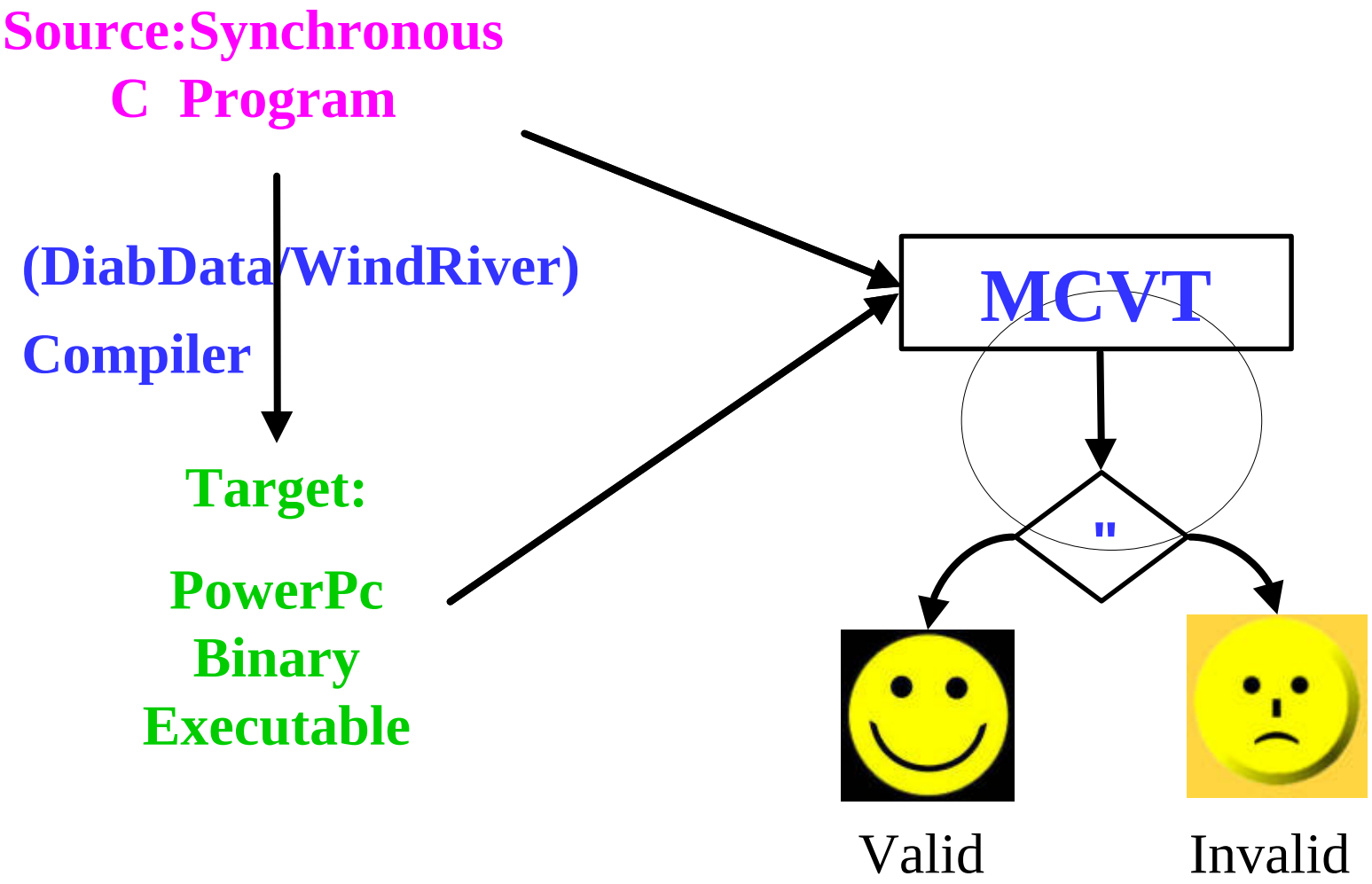
CVC

Cooperating Validity Checker form Stanford University.

MCVT Structure



MCVT – Machine code Validation Tool



Run Time Results
Jet Engine controller hispano Suiza

Number of Lines	Module Name	CPU Time [seconds]
266	Est_Pos_ERe_ssm.c	1221.82
144	LogicalConstantFalse.c	0.09
144	LogicalConstantFalse1.c	0.07
144	LogicalConstantFalse10.c	0.07
144	LogicalConstantFalse1_1.c	0.06
144	LogicalConstantFalse2.c	0.09
144	LogicalConstantFalse3.c	0.08
144	LogicalConstantFalse5.c	0.07
144	LogicalConstantFalse6.c	0.08
144	LogicalConstantFalse7.c	0.08
144	LogicalConstantFalse8.c	0.08
144	LogicalConstantFalse9.c	0.05
144	LogicalConstantFalse_1.c	0.06
144	LogicalConstantFalse_2.c	0.06
144	LogicalConstantFalse_3.c	0.07
144	LogicalConstantTrue.c	0.05
144	LogicalConstantTrue1.c	0.08
144	LogicalConstantTrue_1.c	0.07
144	LogicalConstantTrue_2.c	0.07
145	NOT.c	0.10
145	NOT1.c	0.08
145	NOT2.c	0.11
155	S2S_Demux.c	0.18
149	S2S_Mux.c	0.11

Cont'

Number of Lines	Module Name	CPU Time [seconds]
155	S2S_Mux_1.c	0.18
149	S2S_Mux_2.c	0.10
145	S2S_Selector.c	0.05
145	S2S_Selector_1.c	0.08
145	S2S_Selector_2.c	0.08
148	S2S_Vect_3_2.c	0.16
148	S2S_Vect_3_2_1.c	0.19
148	S2S_Vect_3_3.c	0.21
221	Seq_Cmd_ER.c	0.58
220	Commande_ERe	-
2327	Command_electro_robinet	-
178	S2S_MPSwitch	-
145	SMLK_Terminator_2	-
8286	Total	1225.31

Cont'

Number of Lines	Module Name	CPU Time [seconds]
6414	Calc_Mauv_DebitsPC	0.68
6287	Elab_MVDBe	0.35
6286	Elab_PASSSPCAV	0.82
6286	Elab_PASSSPCV	0.86
6285	Elab_PASSeAV_Act	0.77
6286	Elab_PASSeAV_Deb	0.85
6286	Elab_PASSeV_Deb	0.87
6336	LR_PASS_Ac_ERSPC	32.98
6336	Lo_Re_PASS_Ac_e	32.34

Our Contribution

Handling Industrial environment.

The processor, the compiler, and the applications are all industrial grade.

Not available:

- Intermediate language
- Internal information
- Optimization stages

The process is fully automatic

Limitation: synchronous programs