Translation Validation *

A. Pnueli M. Siegel E. Singerman **

Weizmann Institute of Science, Rehovot, Israel

Abstract. We present the notion of *translation validation* as a new approach to the verification of translators (compilers, code generators). Rather than proving in advance that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program.
Several ingredients are necessary to set up the – fully automatic – translation validation process, among which are:

1. A common semantic framework for the representation of the source code and the generated target code.
2. A formalization of the notion of "correct implementation" as a refinement relation.
3. A syntactic simulation-based proof method which allows to automatically verify that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source.

These, and other ingredients are elaborated in this paper, in which we illustrate the new approach in a most challenging case. We consider a translation (compilation) from the *synchronous* multi-clock data-flow language SIGNAL to *asynchronous* (sequential) C-code.

1 Introduction

In this paper, we present the notion of *translation validation* as a new approach to the verification of translators (compilers, code generators). The idea of translation validation is the following: Rather than proving in advance that the compiler always produces a target code which correctly implements the source code (compiler verification), each individual translation (i.e. a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source code.

Since compiler verification is an extremely complex task and every change to the compiler (even minor revisions) requires redoing the proof, compiler verification tends to "freezes" the compiler design, and discourages any future improvements and revisions. This drawback is avoided in the translation validation

* This research was done as part of the European Community project SACRES (EP 20897) and was supported in part by the Minerva Foundation and an infrastructure grant from the Israeli Ministry of Sciences and Art.

** Current address: Computer Science Laboratory, SRI International, Menlo Park, California.

approach since it compares the input and the output of the compiler for each individual run independently of *how* the output is generated from the input.

The concept of translation validation is depicted in Fig. 1.

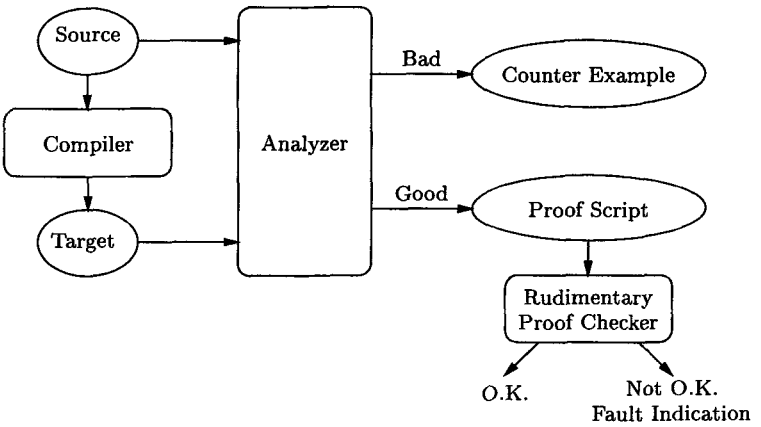


Fig. 1. The concept of Translation Validation

Both the *source* and the *target* programs are fed as inputs to an *Analyzer*. If the analyzer finds that the generated target program correctly implements the source program, it generates a detailed proof script. If the analyzer fails to establish the correct correspondence between source and target, it produces a counter-example. The counter example consists of a scenario in which the generated code behaves differently than the source code. Thus, the counter-example provides an evidence that the compiler is faulty and needs to be fixed.

The following ingredients are necessary to set up the – fully automatic – translation validation process:

1. A common semantic framework for the representation of the source code and the generated target code.
2. A formalization of the notion of "correct implementation" as a refinement relation, based on the common semantic framework.
3. A proof method which allows to prove that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source.
4. Automation of the proof method, to be carried out by the *analyzer* which, if successful, will also generate a *proof script*; and
5. A rudimentary *proof checker* that examines the proof script produced by the analyzer and provides the last confirmation for the correctness of the translation.

These ingredients are elaborated in this paper, in which we illustrate the new approach in a most challenging case. We consider a translation (compila-

tion) from the *synchronous* multi-clock data-flow language SIGNAL [BGJ91] to *asynchronous* (sequential) C-code.

As part of the Esprit-supported SACRES project (Safety Critical Embedded Systems), the proposed translation validation tool described here is expected to provide repeated validation of each run of the translator. To increase the confidence in the correctness of the validation tool itself, it has been structured into an *analyzer* which produces a proof script and a (rudimentary) *proof checker*. This decomposition enables us to make the proof checker, which is responsible for providing the last seal of approval, very simple and almost "verifiable by inspection".

The paper is structured as follows. The next section introduces the basic notions: We present the small, yet representative, SIGNAL pprogram MUX; give the generated C-code of MUX, and explain why it "correctly implements" the source code. Then, we turn to the formal side. In Section 3 we introduce the *synchronous transition system* (STS) computational model. This formalism is used as the common semantic base for the description of both the synchronous source and the asynchronous target programs (SIGNAL and C resp., in our example). Section 4 formalizes the notion of "correct implementation" by means of a refinement relation. A generalization of the refinement-mapping simulation method is advocated as a proof method for the refinement relation. Automation of this proof method, based on syntactic representation of an appropriate proof rule, is the topic of Section 5, and finally, concluding remarks appear in Section 6. A more detailed discussion of the proof-checker and the decision procedures that were used is saved for the full version.

Related Work

Work in a similar direction was recently reported by Cimatti et al. [C97]. Due to the similarity between the source and target languages, the translation they considered is rather straightforward, and is therefore verified using a much simpler technique than the one we develop here.

Another related work is the "Proof-Carrying Code" mechanism of Necula and Lee, cf. [NL96,N97]. We believe that the translation validation approach may have several advantages over proof-carrying code. The translation validation framework is more general due to its abstract computational model and refinement notions, which the proof-carrying code method does not enjoy. Another important advantage of translation validation is that it is fully automatic, while in proof-carrying code the crucial part of the correctness proof, namely, the verification condition, is generated manually.

2 An Illustrative Example

In this section we first illustrate details of the compilation process by means of an example and then explain the principles which underly the translation validation process.

SIGNAL [BGJ91] is a synchronous programming language used for design and implementation of reactive systems. Statements of SIGNAL are intended to relate clocks (frequencies) as well as values of the various (internal and external) signal flows involved in a given reactive system. Variables (signals) in SIGNAL , as is often the case in synchronous languages, are *volatile*. That is, they only hold values at specific time instances along a computation. Put differently, variables are *absent* almost everywhere along a computation.

Consider the following SIGNAL program:

```
process MUX=
     ( ? integer FB
       ! integer N
     )
       (| N:= FB default (ZN-1)
        | ZN:= N $ 1
        | FB^=when (ZN<=1)
        |)
     where
        integer ZN init 1 ;
end
```

This program uses the integer variable FB as input, the integer variable N as output and the local variable ZN. The body of MUX is composed of three statements which are executed concurrently as follows. An input FB is read and copied to N. If N is greater than 1 it is successively decremented by referring to ZN, which holds the previous value of N (using $ to denote the "previous value" operator) . No new input value for FB is accepted until ZN becomes (or is, in case of a previous non-positive input value for FB) less than or equal to 1. This is achieved by the satatement

```
FB^=when (ZN<=1),
```

which is read "the *clock* of FB is on when ZN ≤ 1", and allows FB to be present only when ZN ≤ 1. A possible computation of this program is:

$$\begin{pmatrix} \text{FB}:3 \\ \text{N}:3 \\ \text{ZN}:1 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB}:\perp \\ \text{N}:2 \\ \text{ZN}:3 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB}:\perp \\ \text{N}:1 \\ \text{ZN}:2 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB}:5 \\ \text{N}:5 \\ \text{ZN}:1 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB}:\perp \\ \text{N}:4 \\ \text{ZN}:5 \end{pmatrix} \rightarrow \ldots$$

Where $\perp$ denotes the absence of a signal. Note that SIGNAL programs are not expected to terminate.

Let us now consider the C-code obtained by compiling a SIGNAL program. The main-program consists basically of two functions:

- An *initialization* function, which is called once to provide initial values to the program variables.
- An *iteration* function which is called repeatedly in an infinite loop. This function, whose body calculates the effect of one synchronous "step" of the abstract program, is the essential part of the concrete code.

The iteration function obtained by compiling MUX is given below.

```
logical MUX_iterate()
{
l0:     h1 = TRUE;
l1:     h2 = ZN <= 1;
l2:     if (h2)
l2.1:       read(FB);
l3:     if (h2)
l3.1:       N = FB;
        else
l3.2:       N = ZN - 1;
l4:     write(N);

l5:     ZN = N;
        return TRUE;
}
```

Remark 1. The labels are not generated by the compiler but have been added for reference.

The C-code introduces explicit boolean variables to represent the clocks of SIGNAL variables and events. Variable h1 is the clock of N and ZN, and h2 is the clock of FB.

The C program works as follows. If h2, the clock of FB, has the value *true*, a new value for FB is read and assigned to the variable N. If h2 is *false*, N gets the value ZN − 1. In both cases the updated value of N is output (at l_4) and also copied into ZN, for reference in the next step .

A computation of this program is given below. We skip some of the intermediate states and use the notation $X : *$ to denote that variable X has an arbitrary value.

$$\begin{pmatrix} \text{FB} : * \\ \text{N} : * \\ \text{ZN} : 1 \\ \text{h1} : * \\ \text{h2} : * \\ \text{pc} : l_0 \end{pmatrix} \rightarrow^* \begin{pmatrix} \text{FB} : * \\ \text{N} : * \\ \text{ZN} : 1 \\ \text{h1} : t \\ \text{h2} : t \\ \text{pc} : l_2 \end{pmatrix} \rightarrow^* \begin{pmatrix} \text{FB} : 3 \\ \text{N} : * \\ \text{ZN} : 1 \\ \text{h1} : t \\ \text{h2} : t \\ \text{pc} : l_3 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB} : 3 \\ \text{N} : * \\ \text{ZN} : 1 \\ \text{h1} : t \\ \text{h2} : t \\ \text{pc} : l_{3.1} \end{pmatrix} \rightarrow^* \begin{pmatrix} \text{FB} : 3 \\ \text{N} : 3 \\ \text{ZN} : 1 \\ \text{h1} : t \\ \text{h2} : t \\ \text{pc} : l_5 \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} \text{FB} : 3 \\ \text{N} : 3 \\ \text{ZN} : 3 \\ \text{h1} : t \\ \text{h2} : t \\ \text{pc} : l_0 \end{pmatrix} \rightarrow^* \begin{pmatrix} \text{FB} : 3 \\ \text{N} : 3 \\ \text{ZN} : 3 \\ \text{h1} : t \\ \text{h2} : f \\ \text{pc} : l_2 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB} : 3 \\ \text{N} : 3 \\ \text{ZN} : 3 \\ \text{h1} : t \\ \text{h2} : f \\ \text{pc} : l_3 \end{pmatrix} \rightarrow \begin{pmatrix} \text{FB} : 3 \\ \text{N} : 3 \\ \text{ZN} : 3 \\ \text{h1} : t \\ \text{h2} : f \\ \text{pc} : l_{3.2} \end{pmatrix} \rightarrow^* \begin{pmatrix} \text{FB} : 3 \\ \text{N} : 2 \\ \text{ZN} : 3 \\ \text{h1} : t \\ \text{h2} : f \\ \text{pc} : l_5 \end{pmatrix} \rightarrow \ldots$$

Note the introduction of the variable *pc* which is the *program counter* pointing to the location of the statement which is next to be executed. When comparing this computation to the computation of the SIGNAL program, one finds that the location l_5 is of particular interest: at this location the values of the concrete variables FB, N, and ZN, whose absence or presence is determined by the variables h1 and h2, *coincide* with the values of the corresponding abstract variables.

Taking into account that h1 is the clock of N and ZN and that h2 is the clock of FB, we have an accurate state correspondence between the computation of the SIGNAL program and the following computation of the C-code, where we restrict our observations to subsequent visits at location l_5:

$$\begin{pmatrix} \texttt{FB} : 3 \\ \texttt{N} : 3 \\ \texttt{ZN} : 1 \\ \texttt{h1} : t \\ \texttt{h2} : t \\ \texttt{pc} : l_5 \end{pmatrix} \rightarrow^* \begin{pmatrix} \texttt{FB} : 3 \\ \texttt{N} : 2 \\ \texttt{ZN} : 3 \\ \texttt{h1} : t \\ \texttt{h2} : f \\ \texttt{pc} : l_5 \end{pmatrix} \rightarrow^* \begin{pmatrix} \texttt{FB} : 3 \\ \texttt{N} : 1 \\ \texttt{ZN} : 2 \\ \texttt{h1} : t \\ \texttt{h2} : f \\ \texttt{pc} : l_5 \end{pmatrix} \rightarrow^* \begin{pmatrix} \texttt{FB} : 5 \\ \texttt{N} : 5 \\ \texttt{ZN} : 1 \\ \texttt{h1} : t \\ \texttt{h2} : t \\ \texttt{pc} : l_5 \end{pmatrix} \rightarrow^* \begin{pmatrix} \texttt{FB} : 5 \\ \texttt{N} : 4 \\ \texttt{ZN} : 5 \\ \texttt{h1} : t \\ \texttt{h2} : f \\ \texttt{pc} : l_5 \end{pmatrix} \rightarrow^* \ldots$$

The central observation is that there exists a *designated control location* in the C-code (l_5 in our example) where the variables of the concrete (target) system correspond to their abstract (source) counterparts. This is a general pattern for programs generated by the SACRES compiler. Intuitively, the generated C-code correctly implements the original SIGNAL program if the sequence of states obtained at the designated control location corresponds to a possible sequence of states in the abstract system.

In the rest of the paper, we show how this approach can be put on formal grounds and yield a fully automatic translation validation process.

3 The Computational Model

In this section, we present *synchronous transition systems* (STS), which is the computational model on which the process of translation validation is based.

We assume a vocabulary of typed variables $\mathcal{V}$. Some of the variables are identified as *persistent* while the others are identified as *volatile*. The volatile variables are intended to represent *signals* in the sense of the language SIGNAL. The domains of volatile variables contain the designated element $\perp$ to indicate absence of the respective signal.

A *state* s is a type-consistent interpretation of $\mathcal{V}$, assigning to each variable $v \in \mathcal{V}$ a value $s[v]$ over its domain. We denote by Σ the set of all states over $\mathcal{V}$.

Definition 1. *The following components define a* synchronous transition system (STS) $A = (V, \Theta, \rho, E)$ (cf. [PS97]):

- $V \subseteq \mathcal{V}$: A finite set of *system variables.*
- Θ : An *initial condition.* A satisfiable assertion characterizing the initial states of system A.
- ρ : A *transition relation.* This is an assertion $\rho(V, V')$, which relates a state $s \in \Sigma$ to its possible successors $s' \in \Sigma$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value in s'. If $(s, s') \models \rho(V, V')$, we say that state s' is a *ρ-successor* of state s.
- $E \subseteq V$: A set of *externally observable variables.*

Next, we define a *computation* of an STS.

Definition 2. *Let $A = (V, \Theta, \rho, E)$ be an* STS. *The infinite sequence $\sigma = s_0, s_1, s_2, \ldots$, where $s_i \in \Sigma$ for each $i \in \mathbf{N}$, is a* computation *of A if it satisfies the following requirements:*

$$\begin{array}{ll} \textit{Initiation}: & s_0 \models \Theta \\ \textit{Consecution}: & (s_i, s_{i+1}) \models \rho \qquad \textit{for each } i \in \mathbf{N}. \end{array}$$

We denote by $||A||$ the set of computations of the STS A.

3.1 STS representation of the SIGNAL program

The SIGNAL program MUX is represented by the STS $A = (V, \Theta, \rho, E)$, where

$$V = \{\mathrm{FB}, \mathrm{N}, \mathrm{ZN}, \mathrm{x.N}\}$$

$$\Theta = (\mathrm{FB} = \bot \wedge \mathrm{N} = \bot \wedge \mathrm{ZN} = \bot \wedge \mathrm{x.N} = \bot)$$

$$\rho = \left[\begin{array}{l} \vee \left[\begin{array}{l} \wedge\ \mathrm{N}' = \left(\begin{array}{l} \text{if } \mathrm{FB}' \neq \bot \text{ then } \mathrm{FB}' \\ \text{else if } \mathrm{ZN}' \neq \bot \text{ then } \mathrm{ZN}' - 1 \\ \text{else } \bot \end{array} \right) \\ \wedge\ \mathrm{x.N}' = \text{if } \mathrm{N}' \neq \bot \text{ then } \mathrm{N}' \text{ else } \mathrm{x.N} \\ \wedge\ \mathrm{ZN}' = \left(\begin{array}{l} \text{if } \mathrm{N}' = \bot \text{ then } \bot \\ \text{else if } \mathrm{x.N} = \bot \text{ then } 1 \\ \text{else } \mathrm{x.N} \end{array} \right) \\ \wedge\ \mathrm{ZN}' \leq 1 \leftrightarrow \mathrm{FB}' \neq \bot \end{array} \right] \\ \vee\ (\mathrm{FB}' = \bot \wedge \mathrm{N}' = \bot \wedge \mathrm{ZN}' = \bot \wedge \mathrm{x.N}' = \mathrm{x.N}) \end{array} \right]$$

$$E = \{\mathrm{FB}, \mathrm{N}, \mathrm{ZN}\}$$

Two points here require further explanation:

- Besides maintaining all variables occurring in the SIGNAL-program as volatile variables, the STS-encoding of SIGNAL-programs introduces persistent *memorization variables* for those variables occurring in $-expressions. In our example, there is only one memorization variable, namely, x.N.
- The second disjunct of ρ guarantees the stutter robustness of A. That is, at any step, the system may choose to take a stutter (idling) step in which all signals are set to $\bot$ and all memorization variables retain their previous values.

3.2 STS representation of the C program

The representation of the C code is less straightforward than that of the SIGNAL program. So, we first present the STS and then follow with detailed explanations.

The C code is described by STS C presented below. The predicate $pres(U) = \bigwedge_{v \in U}(v' = v)$ in this presentation expresses that the variables in set $U \subseteq V$ remain unchanged during the current transition, cf. [MP91].

$C = (V, \Theta, \rho, E)$ where

$$V = \{\mathrm{FB}, \mathrm{N}, \mathrm{ZN}, \mathrm{x.N}, \mathrm{h1}, \mathrm{h2}, pc\}$$

$$\Theta = (\mathrm{FB} \neq \bot \wedge \mathrm{N} \neq \bot \wedge \mathrm{ZN} = 1 \wedge \mathrm{x.N} = \bot \wedge pc = l_0)$$

$$\rho = \left[\begin{array}{l} \vee\ (pc = l_0 \wedge \mathrm{h1}' = \mathit{true} \wedge pc' = l_1 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{h1}\})) \\ \vee\ (pc = l_1 \wedge \mathrm{h2}' = (\mathrm{ZN} \leq 1) \wedge pc' = l_2 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{h2}\})) \\ \vee\ (pc = l_2 \wedge \mathrm{h2} \wedge pc' = l_{2.1} \wedge \mathit{pres}(V \setminus \{\mathrm{pc}\})) \\ \vee\ (pc = l_2 \wedge \neg\mathrm{h2} \wedge pc' = l_3 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}\})) \\ \vee\ (pc = l_{2.1} \wedge \mathrm{FB}' \neq \bot \wedge pc' = l_3 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{FB}\})) \\ \vee\ (pc = l_3 \wedge \mathrm{h2} \wedge pc' = l_{3.1} \wedge \mathit{pres}(V \setminus \{\mathrm{pc}\})) \\ \vee\ (pc = l_3 \wedge \neg\mathrm{h2} \wedge pc' = l_{3.2} \wedge \mathit{pres}(V \setminus \{\mathrm{pc}\})) \\ \vee\ (pc = l_{3.1} \wedge \mathrm{N}' = \mathrm{FB} \wedge pc' = l_4 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{N}\})) \\ \vee\ (pc = l_{3.2} \wedge \mathrm{N}' = \mathrm{ZN} - 1 \wedge pc' = l_4 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{N}\})) \\ \vee\ (pc = l_4 \wedge \mathrm{x.N}' = \mathrm{N} \wedge pc' = l_5 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{x.N}\})) \\ \vee\ (pc = l_5 \wedge \mathrm{ZN}' = \mathrm{N} \wedge pc' = l_0 \wedge \mathit{pres}(V \setminus \{\mathrm{pc}, \mathrm{ZN}\})) \end{array}\right]$$

$$E = \{\mathrm{FB}, \mathrm{N}, \mathrm{ZN}\}$$

Some remarks are in order.

Input for FB: Being at location $l_{2.1}$, we allow FB to take an arbitrary non-bottom value, which corresponds to a new input for FB from the environment. If h2 is false and we proceed directly from l_2 to l_3, the value of FB remains unchanged as stated by the $\mathit{pres}(V \setminus \{\mathrm{pc}\})$ clause.

Output of N: The explicit writing of N at location l_4 in the C-program has been removed; instead, the memorization of N is introduced.

The observation point: As explained above, entering location l_5 means that the `mux_iterate` function has cumulatively computed one transition of the abstract system. The values of the persistent variables FB, N, and ZN are considered to be present only when being at location l_5 and if their respective clock expressions have the value *true*. This will become apparent when we define the refinement mapping from STS C to STS A. All other persistent variables are considered internal.

Memorization of N: The generated C-code does not use any memorization variables but rather encode memorization by means of scheduling. In order to match the abstract memorization variables we augment the STS-encoding of the generated C-program with memorization variables which have the same name as their abstract counterparts. The general pattern for memorization is that all variables which are memorized in the abstract system, are memorized in the concrete system *directly before entering the observation location*, i.e. the location where the state correspondence is to be established.

In our example, the value of N is copied to a memorization variable x.N, at location l_4, just before the observation location l_5.

4 Correct Implementation: Refinement

In this section, we consider the notion of *correct implementation* which is the relation that should hold between a source code and its correct translation. We suggest that the appropriate relation is that of *refinement* adapted to our special circumstances that involve a translation from a synchronous language such as SIGNAL into an asynchronous language such as C.

In general, we consider refinement between an *abstract system* A and a *concrete system* C. System A can be viewed as a specification or a high-level description of the application we wish to construct, while C is a description closer to the final implementation. An elaborate development process may progress through several refinement steps, each making the representation more concrete. In many cases, the abstract system is described in a more declarative style while the concrete system is presented in a more operational/imperative style.

In order to make the implementation refinement relation maximally effective, we should make it as liberal as possible, provided the *essential* features of the system are preserved.

4.1 Refinement between Systems

Consider the two systems $A = (V_A, \Theta_A, \rho_A, E_A)$ and $C = (V_C, \Theta_C, \rho_C, E_C)$, to which we refer as the *abstract* and *concrete* systems, respectively.

We assume that $E_A \subseteq E_C$. That is, the abstract observable variables are a subset of the concrete observable variables.

For $T \in \{A, C\}$, we denote by Σ_T, the set of T-states, i.e., the set of states obtained by assigning values to the variables V_T. We denote by Σ_T^E the set of states which only assign values to the variables in $E_T \subseteq V_T$.

For a state $s \in \Sigma_T$, we denote by s^E the restriction of s to the subset of observable T-variables, i.e., to E_T. This restriction can be lifted point-wise to a computation $\sigma \in ||T||$, denoted by σ^E, and then to the complete set of computations $||T||$, denoted by $||T||^E$.

For the two systems A and C, we define an *interface mapping* to be a function

$$\mathcal{I}: \quad \Sigma_C \mapsto \Sigma_A^E,$$

mapping each concrete state $s \in \Sigma_C$ to an abstract observable state $\mathcal{I}(s) \in \Sigma_A^E$.

An interface mapping $\mathcal{I}$ is said to be a *clocked mapping* if, for each observable variable $x \in E_A$ (which also belongs to E_C since $E_A \subseteq E_C$) and every concrete state $s \in \Sigma_C$,

$$\mathcal{I}(s)[x] = s[x] \qquad \text{or} \qquad \mathcal{I}(s)[x] = \perp.$$

That is, the effect of the mapping $\mathcal{I}$ on a variable x which is observable in both systems is either to preserve its value ($\mathcal{I}(s)[x] = s[x]$) or to declare it absent at the current abstract state ($\mathcal{I}(s)[x] = \perp$).

We can point-wise lift the interface mapping $\mathcal{I}$ to a concrete computation $\sigma \in \|C\|$, denoted by $\mathcal{I}(\sigma)$, and then to the complete set of concrete computations $\|C\|$, denoted by $\mathcal{I}(\|C\|)$.

Definition 3. *For systems A and C with $E_A \subseteq E_C$, and a clocked interface mapping $\mathcal{I}$ from C to A, we say that C* **refines** *A* **relative to** *$\mathcal{I}$ if $\mathcal{I}(\|C\|) \subseteq \|A\|^E$.*

That is, C refines A relative to $\mathcal{I}$ if applying the mapping $\mathcal{I}$ to any concrete computation $\sigma \in \|C\|$, we obtain an abstract computation restricted to the observable variables E_A.

Definition 4. *For systems A and C, we say that C* **refines** *A if there exists a clocked interface mapping $\mathcal{I}$ from C to A such that C refines A relative to $\mathcal{I}$.*

We write $C \sqsubseteq A$ to denote the fact that system C refines system A. In the next section we investigate a proof method which allows to establish that $C \sqsubseteq A$ indeed holds for some given $A, C \in \mathrm{STS}$.

4.2 Proving Refinement by the Method of Refinement Mapping (Simulation)

As proof method for the refinement notion introduced above we employ a generalization of the well-established concept of simulation with refinement mapping [AL91]. Refinement mappings define a correspondence between the variables of a concrete system and the variables of an abstract system such that observations are preserved. Refinement mappings, or more generally *simulation techniques* (see, e.g., [Jon91,LV91]), are the means to inductively prove a semantically defined notion of containment between observable behaviors.

Note that, while we employed the notion of clocked interface mapping in the definition of refinement, requiring mapping of concrete states only to the observable part of the abstract state, a general refinement mapping is expected to yield a mapping of a concrete state to a full abstract state. Thus, a refinement mapping can be viewed as one of the many possible extensions of an interface mapping.

We define a *refinement mapping from C to A* to be a function $f : \Sigma_C \longrightarrow \Sigma_A$, mapping concrete to abstract states. A refinement mapping f is called a *clocked refinement mapping* if it satisfies

$$f(s)[x] = s[x] \quad \text{or} \quad f(s)[x] = \perp, \qquad \text{for every } s \in \Sigma_C \text{ and } x \in E_A.$$

From now on, we restrict our attention to clocked refinement mappings, which preserve the observables up to stuttering.

The proposed proof method for refinement is based on finding an inductive refinement mapping as defined below. In the definition, we denote by Σ_C^r the set

of all reachable states of system C, i.e., all states appearing in some computation of C.

Definition 5. *A clocked refinement mapping $f : \Sigma_C \longrightarrow \Sigma_A$ is called* **inductive** *if it satisfies the requirements of*

- *Initiation: $s \models \Theta_C$ implies $f(s) \models \Theta_A$, for all $s \in \Sigma_C$, and*
- *Propagation: $(s, s') \models \rho_C$ implies $(f(s), f(s')) \models \rho_A$, for all $s, s' \in \Sigma_C^r$.*

The use of an inductive refinement mapping as a proof method is stated in the next theorem.

Theorem 1. *If $f : \Sigma_C \longrightarrow \Sigma_A$ is an inductive (clocked) refinement mapping from C to A, then $C \sqsubseteq A$.*

5 Automating the Translation Validation Process

The proof method presented in the previous section was based on an inductive refinement mapping formulated in semantic terms. Among other things, it assumed an available characterization of the set of reachable concrete states Σ_C^r which is very difficult to compute for even the simplest systems.

In the quest for automating the process, we present in this section a *syntactical representation* of the notions of refinement mapping, and its associated proof method. In this, we follow the ideas in [Lam91,KMP94] and adapt them to deal with the particular notion of refinement needed for our case. Then, we describe how the main components used in the proof can be computed, so that the translation validation process can be carried out fully automatically.

5.1 Syntactic Representation and Proof Rules

Consider two STSs A and C with $E_A \subseteq E_C$, to which we refer as the *abstract* and the *concrete* system, respectively. Let $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$ be a *substitution* that replaces each abstract variable $v \in V_A$ by an expression $\mathcal{E}_v$ over the concrete variables V_C. Such a substitution α induces a mapping between states, denoted by $\overrightarrow{\alpha}$. Let s_C be some state in Σ_C; we refer to s_C as a *concrete* state. The abstract state $s_A \stackrel{\text{def}}{=} \overrightarrow{\alpha}(s_C)$ corresponding to s_C under substitution α assigns to each variable $v \in V_A$ the value of expression $\mathcal{E}_v$ evaluated in s_C. In this way, refinement mappings can be syntactically defined by means of an appropriate substitution α.

Now we show how to syntactically formulate the requirements of initiation, propagation, and preservation of observation (the requirement that $\overrightarrow{\alpha}$ is a clocked refinement mapping) for such a state function $\overrightarrow{\alpha}$. For an expression or state formula φ over V_A, we define the formula (resp. expression) $\varphi[\alpha]$ over V_C obtained from φ by replacing each occurrence of $v \in V_A$ by $\mathcal{E}_v$. In the case that φ contains a primed variable v', this variable is replaced by $\mathcal{E}'_v$ obtained by replacing all occurrences of variables $v \in V_C$ in $\mathcal{E}_v$ by their primed versions.

Given a concrete state s_C and substitution α, we have that the value of any φ evaluated over $\vec{\alpha}(s_C)$ is the same as the value of $\varphi[\alpha]$ evaluated over s_C. This holds, since in both cases φ is evaluated using for $v \in V_A$ its value in $\vec{\alpha}(s_C)$ which is the same as the value of $\mathcal{E}_v$ evaluated over s_C. In particular, for a state formula φ over V_A we have $\vec{\alpha}(s_C) \models \varphi$ iff $s_C \models \varphi[\alpha]$. This equivalence allows to write the proof obligations of Definition 5 as stated in the following syntactical proof rule REF for proving refinement of STS-systems.

Definition 5 imposed the requirement of inductiveness only with respect to *reachable* C-states. Since these are difficult to characterize precisely, rule REF makes the stronger requirement which is that the mapping be inductive with respect to all states satisfying some C-invariant *inv*. If *inv* is indeed a C-invariant then all C-reachable states must satisfy *inv* and, therefore, inductiveness over all *inv*-states clearly implies inductiveness over all reachable states.

For assertion *inv* and substitution $\alpha : V_A \longrightarrow \mathcal{E}(V_C)$

R1. $\Theta_C \rightarrow inv$	*inv* holds initially
R2. $inv \wedge \rho_C \rightarrow inv'$	*inv* is propagated
R3. $\Theta_C \rightarrow \Theta_A[\alpha]$	Initiation
R4. $inv \wedge \rho_C \rightarrow \rho_A[\alpha]$	Propagation
R5. $inv \rightarrow (v[\alpha] = v \vee v[\alpha] = \perp)$	for all $v \in E_A$

$$C \sqsubseteq A$$

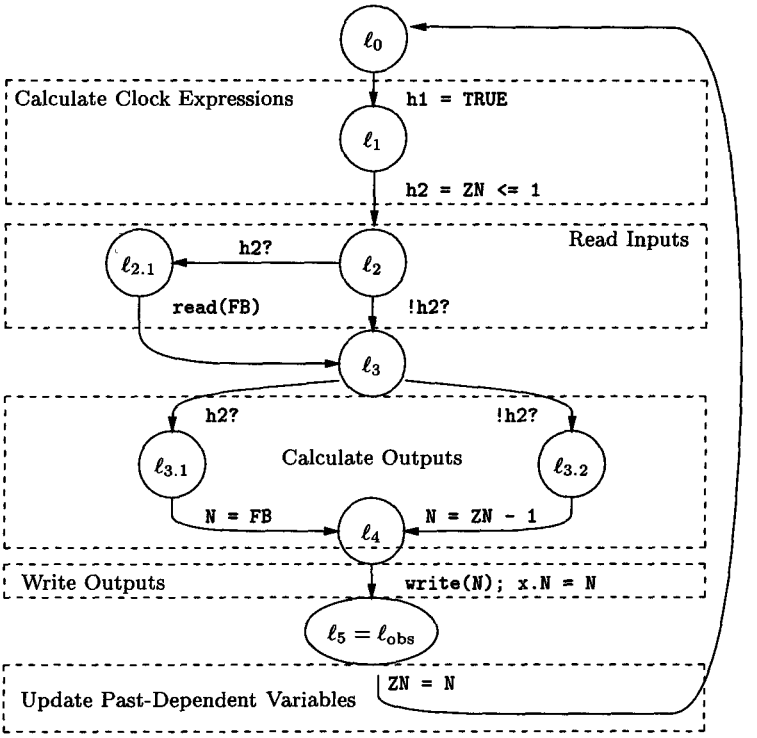
Rule REF: Proving Refinement

Two existential quantifications are hidden in this rule: "find an invariant *inv* and a substitution α, s.t. ...". Generally, finding *inv* and α is left to the ingenuity of the verifier. In order for rule REF to be useful in a fully automatic translation validation process, an appropriate invariant of the concrete system and a suitable substitution have to be generated automatically.

5.2 Generating *inv* and α

In general, there is no chance of developing an algorithm which, presented with arbitrary systems A and C, can automatically construct the needed invariant *inv* and refinement substitution α as well as automatically verify the validity of the premises in rule REF. The reason that this is possible in the case of translation validation applied to the language SIGNAL is that we rely on some very strong assumptions about the connections between A and C, based on the fact that C was produced as a result of translation of system A by a code generator of a very specific structure and mode of operation.

The general structure of the main loop in the C-code is illustrated in the figure below.



As we see in the figure, the body of the infinitely repeated loop consists of the following stages:

1. *Calculation of clock expressions*. This stage assigns values to the boolean auxiliary variables h_i, $i = 1, \ldots, k$. Each of these variables is associated with an abstract observable variable, and is used to represent the "existence"/"absence" of it.
2. *Reading inputs*. This stage reads the inputs of program, sometimes conditioned on the values of the appropriate clock variables.
3. *Calculating outputs*. This stage calculates the value of output variables.
4. *Writing outputs*. This stage write to external files (or channels) the computed values of output variables.
5. *Update "previous" expressions*. This stage updates the values of (usually local) variables defined by expressions containing the *previous* operator ($).

We use this special structure for the construction of the invariant *inv* and the substitution α. We start by noting that using the *program counter* variable *pc*, which is always a member of V_C, we can present the refinement substitution α as follows:

1. For each memorization variable $x.v \in V_A$, we include in α the substitution

$$x.v \quad \longrightarrow \quad x.v.$$

2. For every other variable $v \in V_A$, we include in α the substitution

$$v \quad \longrightarrow \quad \textbf{if } \boldsymbol{pc} = \ell_{\text{obs}} \wedge \boldsymbol{clk}(v) \textbf{ then } v \textbf{ else } \bot,$$

where $clk(v)$ is the clock expression for v, indicating whether a new value had been assigned to v in the current iteration.

The detailed algorithm for computing the clock expressions above, and the accumulative invariant *inv*, which is omitted here for lack of space, is described in the full version of this paper. The construction is based on viewing the main loop of the C-code (procedure MUX-iterate, in our example) as a (cyclic) directed graph, in which ℓ_0 and ℓ_{obs} are two of the nodes, and every edge e is labeled by either a *guard* $\gamma(e)$ or an action which can be a *read* into an input variable, a *write* out of an output variable, or an assignment to a (local or output) variable. For an edge labeled by an action, we can take its guard to be *true*.

The clock expression $clk(v)$ is computed by considering the guards along paths leading to assignments to v. For the MUX-example, the clock expressions obtained are

$$\begin{aligned} clk(\text{FB}) &= \text{h2} \\ clk(\text{N}) &= \text{h2} \vee \neg\text{h2} \ (= \mathit{true}) \\ clk(\text{ZN}) &= \text{h2} \vee \neg\text{h2} \ (= \mathit{true}) \end{aligned}$$

Based on this, the identification of the observation point as l_5 and the general "skeleton" of α given above, we obtain the following refinement substitution

$$\alpha : \begin{pmatrix} \text{FB} \\ \text{N} \\ \text{ZN} \\ \text{x.N} \end{pmatrix} \longrightarrow \begin{pmatrix} \text{if } \ \text{h2} \wedge pc = l_5 \text{ then } \ \text{FB else } \ \bot \\ \text{if } \ pc = l_5 \text{ then } \ \text{N else } \ \bot \\ \text{if } \ pc = l_5 \text{ then } \ \text{ZN else } \ \bot \\ \text{x.N} \end{pmatrix}$$

The invariant *inv* is computed by taking the initial values of variables, and then adding the cumulative effect of the actions that are executed along paths. For the MUX-example, we obtain the following proposal for an invariant

$$inv = \begin{pmatrix} \wedge\, \text{FB} \neq \bot \ \wedge \ \text{N} \neq \bot \ \wedge \ \text{ZN} \neq \bot & & \\ \wedge\, pc \in \{l_0, l_1, l_2, l_{2.1}, l_3, l_{3.1}, l_{3.2}, l_4, l_5\} & & \\ \wedge\, pc \in \{l_1, l_2, l_{2.1}, l_3, l_{3.1}, l_{3.2}, l_4, l_5\} & \rightarrow & \text{h1} \\ \wedge\, pc \in \{l_2, l_{2.1}, l_3, l_{3.1}, l_{3.2}, l_4, l_5\} & \rightarrow & \text{h2} = (\text{ZN} \leq 1) \\ \wedge\, pc \in \{l_4, l_5\} \wedge \text{h2} & \rightarrow & \text{N} = \text{FB} \\ \wedge\, pc \in \{l_4, l_5\} \wedge \neg\text{h2} & \rightarrow & \text{N} = \text{ZN} - 1 \\ \wedge\, pc = l_{2.1} & \rightarrow & \text{h2} \\ \wedge\, pc = l_{3.1} & \rightarrow & \text{h2} \\ \wedge\, pc = l_{3.2} & \rightarrow & \neg\text{h2} \\ \wedge\, pc \neq l_5 & \rightarrow & \text{ZN} = (\textbf{if } \text{x.N} = \bot \textbf{ then } 1 \textbf{ else } \text{x.N}) \\ \wedge\, pc = l_5 & \rightarrow & \text{x.N} = \text{N} \end{pmatrix}$$

We have verified all the premises of rule REF, using the TLV proof system of [PS96]. The script files, which are omitted here for lack of space, will appear in the full version of this paper.

6 Conclusions

We introduced the new approach of *translation validation*, described the main components of the construction together with the underline theory, and presented an illustrative example of the method by validating a compilation from a synchronous language to an asynchronous one.

The concept of translation validation is general, and the interest is obviously not limited to translations from SIGNAL to C. We believe that the main ideas presented in this paper can serve as a basis to the translation validation for a large family of source and target languages.

Our intuition is based on the following. First, the STS computational model is very general and can model both synchronous and asynchronous languages. Second, the existence of designated control location(s) in the STS computations of the source and target programs, that can serve as an observation point(s) for comparing the values of a set of externally observable variables (input/output variables, for example), is a reasonable thing to expect for. Otherwise, in what sense could one say that the target program correctly implements the submitted source code? Finally, our notion of refinement via an interface mapping and the associated proof method, based on syntactic representation of the refinement mapping, is again of a general kind.

The approach described here seems to work in all cases that the source and the target programs each consist of a repeated execution of a single loop body, and the correspondence between the executions is such that a single loop iteration in the source corresponds to as single iteration in the target. This seems to be a characteristic of most code generators for synchronous languages such as Esterel [BG], Lustre [CHPP87], and Statecharts [H87], as well as for languages such as Unity [CMB88].

It is clear that a translation validation "tool-set" should be tailored for the particular translator (compiler) involved. The construction can be carried out by following (and modifying) the guidelines of the framework presented here. (In some cases, it may be useful to augment the translator as to make it easier to identify the observation points.) We suspect that in some cases the construction would turn out to be simpler than what was called for in the example presented here. This is so because most of the difficulties we had faced were due to the fact that SIGNAL is a *synchronous* language while C in *asynchronous*.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2), 1991.

[BG] G. Berry and G. Gonthier. The Synchronous Programming Language Esterel, Design, Semantics, Implementation. Technical Report 327, INRIA.

[BGJ91] A. Benviniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with event and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16, 1991.

[C97] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In O. Grumberg, editor, *Proc.* 9^{th} *Intl. Conference on Computer Aided Verification (CAV'97)*, Lect. Notes in Comp. Sci., vol. 1254, pages 202–213. Springer-Verlag, 1997.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a Declarative Language for Programming Synchronous Systems. *POPL'87*, ACM Press, pages 178–188, 1987.

[CMB88] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[H87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, pages 231–274, 1987.

[Jon91] B. Jonsson. Simulations between specifications of distributed systems. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91*, volume 527 of *LNCS*, 1991.

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, DEC, Systems Research Center, December 1991. To appear in Transactions on programming Languages and Systems.

[LV91] N. Lynch and F. Vaandrager. Forward and backward simulations for timing based systems. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, 1991.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[N97] G. C. Necula. Proof-Carrying Code. In *POPL'97*, ACM press, pages 106–119, 1997.

[NL96] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations*, Usenix, 1996.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci., pages 184–195. Springer-Verlag, 1996.

[PS97] A. Pnueli and E. Singerman. Fair synchronous transition systems and their liveness proofs. Technical report, Weizmann Institute of Science, 1997. Sacres Report.