# Validating High-Level Synthesis *

Sudipta Kundu, Sorin Lerner, and Rajesh Gupta

University of California, San Diego, La Jolla, CA 92093-0404
Email: {skundu, lerner, rgupta}@ cs.ucsd.edu

**Abstract.** The growing design-productivity gap has made designers shift toward using high-level languages like C, C++ and Java to do system-level design. High-Level Synthesis (HLS) is the process of generating Register Transfer Level (RTL) design from these initial high-level programs. Unfortunately, this translation process itself can be buggy, which can create a mismatch between what a designer intends and what is actually implemented in the circuit. In this paper, we present an approach to validate the result of HLS against the initial high-level program using insights from translation validation, automated theorem proving and relational approaches to reasoning about programs. We have implemented our validating technique and have applied it to a highly parallelizing HLS framework called SPARK. We present the details of our algorithm and experimental results.

## 1 Introduction

While hardware designer productivity has grown at an impressive rate over the past few decades, the rate of improvement has not kept pace with chip capacity growth. High-Level Synthesis (HLS) [22, 17, 10] is often seen as a solution to bridge the design-productivity gap. HLS is the process of generating Register Transfer Level (RTL) design consisting of a data path and a control unit from a high-level behavioral description of a digital system, expressed in languages like C, C++ and Java. The synthesis process consists of several inter dependent sub-tasks such as: specification, compilation, scheduling, allocation, binding and control generation. HLS is an area that has been widely explored and relatively mature implementations of various HLS algorithm have started to emerge [22, 17, 10]. HLS tools are large and complex software systems, often with hundreds of thousands of lines of code, and as with any software of this scale, they are prone to logical and implementation errors. Errors in these tools lead to the synthesis of RTL designs with bugs in them, which often have expensive ramifications if they go undetected until after fabrication or large-scale production. Hence, correctness of these HLS tools has always been an important concern.

Despite significant amount of work in the area of verification we are still far from being able to prove automatically that a given optimizing HLS tool always produces target programs that are semantically equivalent to their source versions. However, even if one cannot prove an HLS tool correct once and for all, one can try to show, for each translation that the HLS tool performs, that the output program produced by the tool has the same behavior as the original

* This research was supported in part by NSF CAREER Grant 0644306.

program. Although this approach does not guarantee that the HLS tool is bug free, it does guarantee that any errors in translation will be caught when the tool runs, preventing such errors from propagating any further in the hardware fabrication process. This approach to verification, called *translation validation*, has previously been applied with success in the context of optimizing compilers [20, 19, 21, 23, 8], and for automatically proving refinements of CSP programs [14].

The main contribution of this paper is to show how translation validation can effectively be implemented in a previously unexplored setting: an HLS tool. In particular, we present an algorithm for validating all the phases (except for parsing, binding and code generation) of the SPARK HLS tool [10] against the initial behavioral description. With over 4,000 downloads, and over 100 active members in the user community, SPARK is a widely used tool. Although commercial HLS tools exists, these tools are not available for academic experimentation – SPARK represents the state of the art in the academic community.

Our algorithm uses a bisimulation relation approach to prove equivalence. In particular, we automatically establish a bisimulation relation that states what points in the specification program are related to what points in the implementation program. This bisimulation relation guarantees that for each execution sequence in the specification, a related and equivalent execution sequence exists in the implementation and vice versa. To deal with the parallelism introduced by the scheduling step of SPARK, we exploit the structure of the transformations that SPARK applies during scheduling. These transformations convert a sequential program to a program that contains instruction-level parallelism. Our algorithm deals with this parallelism using standard techniques for computing weakest preconditions and strongest postconditions of parallel programs [4]. Furthermore, the algorithm we present also draws insights and techniques from various areas, including translation validation [20, 19], theorem proving [5], and relational approaches to reasoning about programs [11, 15].

We implemented our algorithm in a tool that validates SPARK's HLS process. We used our tool to verify the translation of a variety of benchmarks. Because our verification approach works on one procedure at a time, it is modular. Furthermore, our validation tool took on average 6 seconds to run per procedure, showing that translation validation of HLS transformations can be fast enough to be practical. Finally, in running our tool, two failed validation runs have lead us to discover two previously unknown bugs in the SPARK tool. These bugs cause SPARK to generate *incorrect* RTL for a given high-level program. This demonstrates that translation validation of the HLS process can catch bugs that even testing and long-term use may not uncover.

## 2 Overview

We start by presenting a simple example that illustrates our approach (Figure 1). The specification is a sequential function shown in Figure 1(a) using a transition diagram. This function takes $\mathsf{p}$ as input and computes the sum from $(\mathsf{p}+1)$ to $\mathsf{10}$ using a loop and returns it. A parallelizing HLS tool will apply various kinds of transformations to this sequential function, with the goal of scheduling each operation based on some resource constraints.

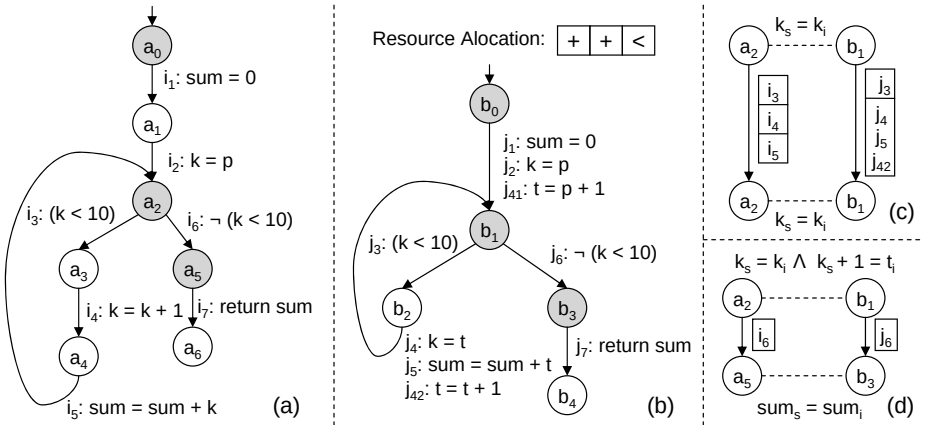


**Fig. 1.** Our running example (a) Specification (b) Implementation (c) and (d) Parts of $2^{nd}$ iteration

| $(l_1, l_2)$ | $1^{st}$ iteration | $2^{nd}$ iteration | $3^{rd}$ iteration ($\phi$) |
|---|---|---|---|
| 1. $(a_0, b_0)$ | $\mathsf{p_s} = \mathsf{p_i}$ | $\mathsf{p_s} = \mathsf{p_i}$ | $\mathsf{p_s} = \mathsf{p_i}$ |
| 2. $(a_2, b_1)$ | $\mathsf{k_s} = \mathsf{k_i}$ | $(\mathsf{k_s} = \mathsf{k_i}) \wedge$ $(\mathsf{sum_s} = \mathsf{sum_i}) \wedge$ $((\mathsf{k_s} + 1) = \mathsf{t_i})$ | $\mathsf{k_s} = \mathsf{k_i} \wedge$ $\mathsf{sum_s} = \mathsf{sum_i} \wedge$ $(\mathsf{k_s} + 1) = \mathsf{t_i}$ |
| 3. $(a_5, b_3)$ | $\mathsf{sum_s} = \mathsf{sum_i}$ | $\mathsf{sum_s} = \mathsf{sum_i}$ | $\mathsf{sum_s} = \mathsf{sum_i}$ |

**Table 1.** Iterations for computing the bisimulation relation

Figure 1(b) shows the result of running SPARK's HLS algorithm given resource constraints of 2 adders and a comparator. Instructions on the same transition edge are executed in *parallel*. For this example SPARK has performed several transformations. First, it applied a loop-shifting transformation that moves the operation $\mathsf{i}_4$ from the beginning of the loop body to the end of the loop body ($\mathsf{j}_{42}$), while also placing a copy of the operation in the loop header ($\mathsf{j}_{41}$) using the temporary variable t. The effect of this loop-shifting transformation is a form of software pipelining [16]. Notice that without this pipelining transformation it would not have been possible to schedule the operation $\mathsf{i}_4$ and $\mathsf{i}_5$ together due to data dependence between them. In addition to loop-shifting, SPARK also performed copy propagation of instruction $\mathsf{j}_2$ to $\mathsf{j}_{41}$ and instruction $\mathsf{j}_4$ to $\mathsf{j}_{42}$.

**Bisimulation Relation.** In order to show that the implementation is equivalent to the specification, our approach computes a bisimulation relation between the two programs. The goal of the bisimulation relation is to guarantee that the specification and the implementation perform the same set of visible instructions. In our case, we consider visible instructions to be function calls and return statements. Our technique thus guarantees that the specification and the implementation perform the same sequence of function calls (with the same arguments) and returns (with the same returned values).

The bisimulation relation (defined formally in Section 4) consists of a set of entries of the form $(l_1, l_2, \phi)$, where $l_1$ and $l_2$ are locations in the specification and implementation respectively, and $\phi$ is a predicate over variables of the specification and implementation. The pair $(l_1, l_2)$ captures how the control state of the specification is related to the control state of the implementation, whereas $\phi$ captures how the data is related. For instance, Table 1 shows the bisimulation relation for our running example. The control component of entries in the bisimulation relation are shown in the first column and the data component in the last column of the table.

The first entry in the bisimulation relation relates the start location of the specification and the implementation. For this entry, the relevant data invariant is $\mathsf{p_s} = \mathsf{p_i}$, which states that the value of the input argument $\mathsf{p}$ in the specification is equal to the value of the input argument $\mathsf{p}$ in the implementation. We use subscript $\mathsf{s}$ to denote variables in the specification and subscript $\mathsf{i}$ for variables in the implementation. The second entry in the bisimulation relation relates the loop head $(a_2)$ in the specification with the loop head $(b_1)$ of the implementation. This entry represent two loops that run in synchrony, one loop being in the specification and the other being in the implementation. The invariant can be seen as a loop invariant across the specification and the implementation, which guarantee that the two loops produce the same effect on the visible instructions. The control part of this entry guarantee that the two loops are in fact synchronized. The last entry in the bisimulation relation relates the location $a_5$ in the specification with the location $b_3$ of the implementation. The relevant invariant for this entry is $\mathsf{sum_s} = \mathsf{sum_i}$, since the value returned by both the program should be same (our equivalence criterion).

The entries in the bisimulation relation must satisfy some simple local requirements (which are made precise in Section 4). Intuitively, for any entry $(l_1, l_2, \phi)$ in the bisimulation relation, if the specification and implementation start executing in parallel at control locations $l_1$ and $l_2$ in states where $\phi$ holds, and in doing so reach another bisimulation entry $(l_1', l_2', \phi')$, then $\phi'$ must hold in the resulting states.

**Our Approach.** Our technique for equivalence checking starts by finding pairs of locations in the implementation and the specification that need to be related in the bisimulation. This amounts to computing the first column of Table 1. In the given example, our algorithm first adds $(a_0, b_0)$ as a pair of interest, which is the entry location of both programs. Then it moves forward simultaneously in the implementation and the specification until it reaches a branch, a function call or a return instruction. In the example from Figure 1, our algorithm finds that there is a branch and a return instruction that must be matched (the specification locations $a_2$ and $a_5$ should match, respectively, with the implementation location $b_1$ and $b_3$). While finding these pairs of locations, our algorithm correlates the branch in the specification and the implementation (the details of how we establish branch correlations is explained in Section 5).

Once the related pairs of locations have been collected we define, for each pair of locations $(l_1, l_2)$, a constraint variable $\psi_{(l_1,l_2)}$ to represent the state-relating

formula that will be computed in the bisimulation relation for that pair. We then define a set of constraints over these variables that must be satisfied in order for the would-be bisimulation relation to in fact be a bisimulation.

There are two kinds of constraints. First, for each pair of locations $(l_1, l_2)$ that are related, we want $\psi_{(l_1,l_2)}$ to imply that any visible instructions about to execute at $(l_1, l_2)$ behave the same way. For example, $\psi_{(a_5,b_3)}$ should imply $\mathsf{sum_s} = \mathsf{sum_i}$, so that the returned values are the same. Such constraints guarantee that the computed bisimulation relation is strong enough to show that the visible instructions behave the same way in the specification and the implementation. A second kind of constraint is used to state the relationship between one pair of related locations and other pairs of related locations. For example, if starting at $(l_1, l_2)$ in states satisfying $\psi_{(l_1,l_2)}$, the specification and implementation can execute in parallel to reach another related pair of locations $(l_1', l_2')$, then $\psi_{(l_1',l_2')}$ must hold in the resulting states. As shown in Section 5, such constraints can be stated over the variables $\psi_{(l_1,l_2)}$ and $\psi_{(l_1',l_2')}$ using the weakest precondition operator ($\mathsf{wp}$). This second kind of constraint guarantees that the computed bisimulation relation is in fact a bisimulation.

Once the constraints are generated, we solve them using an iterative algorithm that starts with all constraint variables set to *true* and then iteratively strengthens the constraint variables until a theorem prover is able to show that all constraints are satisfied. Although in general this constraint-solving algorithm is not guaranteed to terminate, in practice it can quickly find the required bisimulation relation.

The constraint solving for our example is shown in Table 1. Our algorithm first initializes the constraint variables with the conditions that are required for the visible instructions to be equivalent. Then it chooses any entry from the table, say $(a_2, b_1)$ and finds the entries that can reach it (i.e. $(a_2, b_1)$ and $(a_0, b_0)$). Consider the synchronized loop from $(a_2, b_1)$ to $(a_2, b_1)$ shown in Figure 1(c). Our algorithm computes the weakest precondition of the formula at the bottom $(\mathsf{k_s} = \mathsf{k_i})$ over the instructions in the implementation and in the specification, which happens to be $\delta = [(\mathsf{k_s} < 10) \Rightarrow (\mathsf{k_i} < 10) \Rightarrow (\mathsf{k_s} + 1) = \mathsf{t_i}]$. Next, it asks a theorem prover if the condition at the top i.e. $\mathsf{k_s} = \mathsf{k_i}$ implies $\delta$. Since it does not, our algorithm strengthens the condition at the top with $(\mathsf{k_s} + 1) = \mathsf{t_i}$ which is a stronger condition than $\delta$. A similar pass through Figure 1(d) strengthens the condition at $(a_2, b_1)$ with $(\mathsf{sum_s} = \mathsf{sum_i})$. Our constraint solving continues in this manner until a fixpoint is reached.

## 3 Definition of Equivalence

Having illustrated our approach using a simple example, we now present a formal description. Our approach verifies each procedure from the specification against the corresponding procedure from the implementation. We represent each procedure in the specification and the implementation using a *transition diagram* that describes the control structure of the procedure in terms of *program locations* and *program transitions*. A program location represents a point of control in the procedure, and a transition describes how the program state changes from one program location to another. We represent these transitions by *instructions*.

More formally, we define a program state to be a function $VAR \rightarrow VAL$ assigning values to variables, where $VAR$ denotes the set of variables and $VAL$ denotes the domain of values. We denote by $\Sigma$ the a set of all program states. We define an instruction to be a pair $(c, f)$ where $c : \Sigma \rightarrow \mathcal{B}$ is a predicate and $f : \Sigma \rightarrow \Sigma$ is a state transformation function. The predicate $c$ is the condition under which the state transformation function $f$ can happen. For instance, in Figure 1(a) the instruction $\mathsf{i}_3$ has $c = (\mathsf{k} < 10)$ and $f(\sigma) = \sigma$, whereas the instruction $\mathsf{i}_2$ has $c = \mathit{true}$ and $f(\sigma) = \sigma[\mathsf{k} \mapsto \sigma(\mathsf{p})]$.

Finally a transition diagram is defined as follows.

**Definition 1 (Transition Diagram)** *A transition diagram $\pi$ is a tuple $(\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, where $\mathcal{L}$ is a finite set of locations, $\mathcal{I}$ is a finite set of instructions, $\rightarrow \subseteq \mathcal{L} \times \mathcal{I} \times \mathcal{L}$ is a finite set of triples $(l, a, l')$ called transitions, and $\iota \in \mathcal{L}$ is the entry location. We write $l \xrightarrow{i} l'$ to denote $(l, i, l') \in \rightarrow$.*

**Definition 2 (Semantic Step)** *Given a transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, we define a configuration to be a pair $\langle l, \sigma \rangle$, where $l \in \mathcal{L}$ and $\sigma \in \Sigma$. Given two configurations $\langle l, \sigma \rangle$ and $\langle l', \sigma' \rangle$, and an instruction $i \in \mathcal{I}$, the semantic step relation is defined as follows:*

$$\langle l, \sigma \rangle \overset{i}{\rightsquigarrow} \langle l', \sigma' \rangle \quad \textit{iff} \quad l \xrightarrow{i} l' \textit{ and } i = (c, f) \textit{ and } c(\sigma) = \mathit{true} \textit{ and } \sigma' = f(\sigma)$$

**Definition 3 (Execution Sequence)** *For a given transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, an execution sequence $\eta$ starting in $\sigma_0 \in \Sigma$ is a sequence of configurations such that:*

$$\langle l_0, \sigma_0 \rangle \overset{i_1}{\rightsquigarrow} \langle l_1, \sigma_1 \rangle \overset{i_2}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle l_n, \sigma_n \rangle$$

*We denote by $\mathcal{N}$ the set of all execution sequences.*

We define $\vartheta$ to be the set of *visible instructions*. These are the instructions whose semantics we would like preserved between the specification and implementation. In our system we consider visible instructions to be function calls and returns. For $v_1, v_2 \in \vartheta$, we write $\langle v_1, \sigma_1 \rangle \equiv \langle v_2, \sigma_2 \rangle$ to represent that $v_1$ in program state $\sigma_1$ is equivalent to $v_2$ in program states $\sigma_2$. For two visible instructions to be equivalent, they must both be returns, or both calls. Furthermore, returns are equivalent if the returned value and the state of the memory are the same. Two function calls are equivalent if the state of globals, the arguments and the address of the called function are the same. This concept of equivalence for visible instruction can be extended to execution sequences as follows.

**Definition 4 (Equivalence of Execution Sequences)** *Two execution sequences $\eta_1$ and $\eta_2$ are said to be equivalent, written $\eta_1 \equiv \eta_2$, if the two sequences contain visible instructions that are pairwise equivalent.*

**Definition 5 (Equivalence of Transition Diagrams)** *For given initial states $\sigma_1 \in \Sigma_1$ and $\sigma_2 \in \Sigma_2$, two transition diagrams $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ are said to be equivalent if for every execution sequence of $\pi_1$ starting in configuration $\langle \iota_1, \sigma_1 \rangle$ there is an equivalent execution sequence of $\pi_2$ starting in configuration $\langle \iota_2, \sigma_2 \rangle$ and vice-versa.*

## 4 Bisimulation Relation

A *verification relation* between two transition diagrams $\pi_1$ and $\pi_2$ is a set of triples $(l_1, l_2, \phi)$, where $l_1 \in \mathcal{L}_1$, $l_2 \in \mathcal{L}_2$ and $\psi$ is a predicate over the variables live at locations $l_1$ and $l_2$. Let the set of such predicates be denoted by $\Phi \stackrel{def}{=} \Sigma \times \Sigma \to \mathcal{B}$. We write $\phi(\sigma_1, \sigma_2) = \mathit{true}$ to indicate that $\phi$ is satisfied in $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$.

*Simulation relations* and *bisimulation relations* are verification relations with a few additional properties. To define these properties, we make use of a *cumulative semantic step* relation $\rightsquigarrow^*$, which works like $\rightsquigarrow$, except that it can take multiple steps at once, and it accumulates the steps taken into an execution sequence.

**Definition 6 (Cumulative Semantic Step)** *Given configurations $\langle l_0, \sigma_0\rangle$ and $\langle l_n, \sigma_n\rangle$, and an execution sequence $\eta$ that contains at least one transition, we define $\rightsquigarrow^*$ as follows:*

$$\langle l_0, \sigma_0\rangle \overset{\eta}{\rightsquigarrow}{}^* \langle l_n, \sigma_n\rangle \quad \textit{iff} \quad \eta = \langle l_0, \sigma_0\rangle \overset{i_1}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle l_n, \sigma_n\rangle$$

**Definition 7 (Simulation Relation)** *A simulation relation $R$ for two transition diagrams $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ is a verification relation such that:*

$$R(\iota_1, \iota_2, \mathit{true})$$

$$\forall (l_1, l_2, l_1', \sigma_1, \sigma_2, \sigma_1', \phi, \eta_2) \in \mathcal{L}_1 \times \mathcal{L}_2 \times \mathcal{L}_1 \times \Sigma \times \Sigma \times \Sigma \times \Phi \times \mathcal{N} \ .$$

$$\left[\langle l_1, \sigma_1\rangle \overset{\eta_1}{\rightsquigarrow}{}_1^* \langle l_1', \sigma_1'\rangle \wedge R(l_1, l_2, \phi) \wedge \phi(\sigma_1, \sigma_2) = \mathit{true}\right] \Rightarrow$$

$$\exists (l_2', \sigma_2', \phi', \eta_2) \in \mathcal{L}_2 \times \Sigma \times \Phi \times \mathcal{N} \ .$$

$$\left[\langle l_2, \sigma_2\rangle \overset{\eta_2}{\rightsquigarrow}{}_2^* \langle l_2', \sigma_2'\rangle \wedge R(l_1', l_2', \phi') \wedge \phi'(\sigma_1', \sigma_2') = \mathit{true} \wedge \eta_1 \equiv \eta_2\right]$$

Intuitively, these conditions respectively state that (1) the entry location of $\pi_1$ must be related to the entry location of $\pi_2$; and (2) if $\pi_1$ and $\pi_2$ are in a pair of related configurations, and $\pi_1$ can proceed one or more steps producing an execution sequence $\eta_1$, then $\pi_2$ must also be able to proceed one or more steps, producing a sequence $\eta_2$ that is equivalent to $\eta_1$, and the two resulting configurations must be related.

Even though in the above definition, the state-relating predicate for the entry locations is *true*, dummy assignments to a procedure's arguments allow us to prove that the arguments in the specification are equal to those in the implementation, at the beginning of each procedure.

**Definition 8 (Bisimulation Relation)** *A verification relation $R$ is a bisimulation relation for $\pi_1, \pi_2$ iff $R$ is a simulation relation for $\pi_1, \pi_2$ and $R^{-1} = \{(l_2, l_1, \phi) \mid R(l_1, l_2, \phi)\}$ is a simulation relation for $\pi_2, \pi_1$.*

The following lemma and theorem connect the definition of bisimulation with our definition of equivalence for transition diagrams (Definition 5).

**Lemma 1.** *If $R$ is a bisimulation relation for $\pi_1, \pi_2$, then for each element $(l_1, l_2, \psi) \in R$, all pairs of executions of $\pi_1$ started at $l_1$ and of $\pi_2$ started at $l_2$, in states that satisfy the predicate $\psi$, are equivalent.*

**Theorem 1.** *If there exists a bisimulation relation for $\pi_1, \pi_2$, then $\pi_1$ and $\pi_2$ are equivalent.*

The conditions from Definition 7 are the base case and the inductive case of a proof by induction showing that $\pi_2$ is equivalent to $\pi_1$. Thus, a bisimulation relation is a witness that two transition diagrams are equivalent.

## 5 Translation Validation Algorithm

Our translation validation algorithm works by inferring a bisimulation relation. Given a transition diagram $\pi$, we define $\mathcal{P}_\pi$ to be the set of locations for which our approach will try to infer bisimulation entries. These include all locations before visible events and also all locations before branch statements. To focus our attention on only those locations for which our approach infers bisimulation entries, we define the *skipping transition* relation $\hookrightarrow$, which is a version of $\longrightarrow$ that skips over all locations not in $\mathcal{P}_\pi$.

**Definition 9 (Skipping Transition)** *Let $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$ be a transition diagram, $l, l' \in \mathcal{P}_\pi$, and $w \in \mathcal{I}^*$, where $w = i_0 \cdots i_n$. We define $\hookrightarrow$ as follows:*

$$l \overset{w}{\hookrightarrow} l' \textit{ iff there exists } l_1, \cdots, l_n \in (\mathcal{L} - \mathcal{P}_\pi) \textit{ such that } l \overset{i_0}{\longrightarrow} l_1 \cdots l_n \overset{i_n}{\longrightarrow} l'$$

Throughout the rest of this section, we assume that $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ represents the procedure in the specification whose translation we want to verify, and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ represents the corresponding procedure in the implementation. We let $\mathcal{P}_1 = \mathcal{P}_{\pi_1}$ and $\mathcal{P}_2 = \mathcal{P}_{\pi_2}$. We also let $\hookrightarrow_1$ and $\hookrightarrow_2$ be the skipping transitions for $\pi_1$ and $\pi_2$ respectively.

We now define a parallel transition relation $\overset{}{\hookrightarrow\!\!\!\!\rightarrow}$ that essentially traverses the two transition diagrams (specification and implementation) in synchrony.

**Definition 10 (Parallel Transition)** *Given $(l_1, l_2) \in \mathcal{P}_1 \times \mathcal{P}_2$, $(l_1', l_2') \in \mathcal{P}_1 \times \mathcal{P}_2$, $w_1 \in \mathcal{I}_1^*$ and $w_2 \in \mathcal{I}_2^*$, we define $\overset{}{\hookrightarrow\!\!\!\!\rightarrow}$ as follows:*

$$(l_1, l_2) \overset{(w_1, w_2)}{\hookrightarrow\!\!\!\!\rightarrow} (l_1', l_2') \quad \textit{iff} \quad l_1 \overset{w_1}{\hookrightarrow}_1 l_1' \textit{ and } l_2 \overset{w_2}{\hookrightarrow}_2 l_2' \textit{ and } \mathsf{Rel}(w_1, w_2, l_1, l_2)$$

The predicate $\mathsf{Rel} : \mathcal{I}^* \times \mathcal{I}^* \times \mathcal{P}_1 \times \mathcal{P}_2 \rightarrow \mathcal{B}$ used in the above definition is a heuristic that tries to estimate when a path in the specification is related to a path in the implementation. Consider for example the branch in the specification of Figure 1 and the corresponding branch in the implementation. For any two such branches, the $\mathsf{Rel}$ function uses heuristics to guess a correlation between them: either they always go in the same direction, or they always go in opposite direction. Using these correlations, $\mathsf{Rel}(w_1, w_2, l_1, l_2)$ returns true only if the paths $w_1$ and $w_2$ follow branches in a correlated way. Although $\mathsf{Rel}$ makes guesses about

the correlation of branches, the later constraint solving phase of our approach makes sure that these guesses are correct.

Our implementation of Rel correlates branches in two ways. First, using the results of a strongest postcondition pre-pass over the specification and the implementation, Rel tries to use a theorem prover to prove that certain branches are correlated. If the theorem prover is not able to determine a correlation, Rel uses the structure of the branch predicate and the structure of the instructions on each side of the branch to guess a correlation. For instance, in the example of Figure 1, since the strongest postcondition involves the input parameter p, the theorem prover is unable to reason about it. However, because SPARK does not change the structure of the branch predicate, Rel can conclude that the two branches go in the same direction.

We now define the relation $\mathcal{R} \subseteq \mathcal{P}_1 \times \mathcal{P}_2$ of location pairs that will form the entries of our bisimulation relation.

**Definition 11 (Pairs of Interest)** *The relation $\mathcal{R} \subseteq \mathcal{P}_1 \times \mathcal{P}_2$ is defined to be the minimal relation that satisfies the following two properties:*

$$\mathcal{R}(\iota_1, \iota_2)$$

$$\left[\mathcal{R}(l_1, l_2) \ \wedge \ (l_1, l_2) \overset{(w_1,w_2)}{\hookrightarrow\!\!\!\!\!\!\rightarrow} (l_1', l_2')\right] \Longrightarrow \mathcal{R}(l_1', l_2')$$

The set $\mathcal{R}$ defined above can easily be computed by starting with the empty set, and applying the above two rules exhaustively.

For our approach to successfully validate a translation, the computed set $\mathcal{R}$ must relate locations where the instructions to be executed are similar. This is made precise by the following definition of *well-matching* of $\mathcal{R}$. If the computed set $\mathcal{R}$ is *not* well-matched, then our validation approach immediately rejects the translation from specification to implementation.

**Definition 12 (Well-matching)** *For each $(l_1, l_2) \in \mathcal{R}$, if we let $i_1$ and $i_2$ be the instructions to be executed after $l_1$ and $l_2$, respectively, then for $\mathcal{R}$ to be well-matched, the following must hold: $i_1$ is a branch iff $i_2$ is a branch; $i_1$ is a function call iff $i_2$ is a function call; and $i_1$ is a return iff $i_2$ is a return.*

We describe our translation validation approach in terms of constraint solving. In particular, for each $(l_1, l_2) \in \mathcal{R}$ we define a constraint variable $\psi_{(l_1,l_2)}$ representing the predicate that we want to compute for the bisimulation entry $(l_1, l_2)$. We denote by $\Psi$ the set of all such constraint variables. Using these constraint variables, the final bisimulation relation will have the form $\{(l_1, l_2, \psi_{(l_1,l_2)}) \mid \mathcal{R}(l_1, l_2)\}$.

To compute the predicates that the constraint variables $\psi_{(l_1,l_2)}$ stand for, we define a set of constraints on these variables, and then solve the constraints. The constraints are defined as follows.

**Definition 13 (Constraint)** *A constraint is a formula of the form $\psi_1 \Rightarrow f(\psi_2)$, where $\psi_1, \psi_2 \in \Psi$, and $f$ is a boolean function.*

```
1.  function SolveConstraints(C)
2.      for each (l1, l2) ∈ R do
3.          ψ(l1,l2) := true
4.      let worklist := C
5.      while worklist not empty do
6.          let [ψ(l1,l2) ⇒ f(ψ(l1',l2'))] := worklist.Remove
7.          if ATP(ψ(l1,l2) ⇒ f(ψ(l1',l2'))) ≠ Valid then
8.              if (l1, l2) = (ι1, ι2) then
9.                  Error("Start Condition not strong enough")
10.             ψ(l1,l2) := ψ(l1,l2) ∧ f(ψ(l1',l2'))
11.             worklist := worklist ∪ {c ∈ C | ∃ψ, g . c = [ψ ⇒ g(ψ(l1,l2))]}
```

**Fig. 2.** Algorithm for solving constraints

**Definition 14 (Set of Constraints)** *The set $\mathcal{C}$ of constraints is defined by:*

$$\textit{For each } (l_1, l_2) \textit{ in } \mathcal{R}\textit{:}\ \ \left[\psi_{(l_1,l_2)} \Rightarrow \mathsf{CreateSeed}(l_1, l_2)\right] \in \mathcal{C}$$

$$\textit{For each } (l_1, l_2) \overset{(w_1,w_2)}{\rightarrowtail\!\!\!\!\!\!\rightarrow} (l_1', l_2')\textit{:}\ \ \left[\psi_{(l_1,l_2)} \Rightarrow \mathsf{wp}(w_1, \mathsf{wp}(w_2, \psi_{(l_1',l_2')}))\right] \in \mathcal{C}$$

The CreateSeed function above creates for each pair of locations $(l_1, l_2)$ a formula (which does not refer to any constraint variables) that captures the condition under which the instructions about to execute at $l_1$ and $l_2$ are equivalent. Because $\mathcal{R}$ is well-matched (see Definition 12), there are three cases: if the instructions about to execute at $l_1$ and $l_2$ are calls, then the formula returned by CreateSeed states that the parameters of the calls are equal; if the two instructions are returns, then the formula states that the returned values are equal; if the two instructions are branches, then the formula states the two branches are correlated (either they both go in the same direction, or in opposite directions).

The other function wp used above computes the weakest precondition with respect to $w_2$ and then with respect to $w_1$. When computing wp with respect to one sequence, we treat all variables from the other sequence as constants. As a result, the order in which we process the two sequences does not matter.

Having created a set of constraints $\mathcal{C}$, our validation approach now solves these constraints using the algorithm in Figure 2. The algorithm starts by setting each constraint variable to *true* (line 3) and initializing a *worklist* with the set of all constraints (line 4). Next, while the *worklist* is not empty, it removes a constraint from the worklist (line 6), and checks using a theorem prover if it is *Valid* (line 7). If not, then it appropriately strengthens the left-hand-side variable of the constraint (line 10) and adds to the worklist all the constraints that have this variable in the right-hand side (line 11).

## 6 Evaluation

We implemented our validation algorithm on the intermediate representation (IR) of the SPARK HLS framework [10]. SPARK is a C-to-VHDL parallelizing high-level synthesis framework that employs a set of compiler, parallelizing compiler, and synthesis transformations to improve the quality of high-level synthesis

results. SPARK starts with a behavioral description in ANSI-C as input – currently with the restrictions of no pointers, no recursion, and no irregular control-flow jumps. It converts the input program into its own IR, and then applies a set of code transformations, including loop unrolling, loop fusion, common sub-expression elimination, copy propagation, dead code elimination, loop-invariant code motion, induction variable analysis, and operation strength reduction. Following these transformations, SPARK performs a scheduling phase using resource allocation information provided by the user. This scheduling phase also performs a variety of transformations, including speculative code motion, dynamic renaming of variables, dynamic branch balancing, chaining of operations across conditional blocks, and scheduling on multi-cycle operations. The scheduling phase is followed by a resource binding phase and finally by a back-end code generation pass that produces RTL VHDL.

We implemented our translation validation algorithm using the Simplify theorem prover [5] in a tool that validates SPARK's HLS process. Our tool takes as input the IR program that is produced by the parser, and the IR program right before resource binding, and verifies that the two are equivalent. Our tool therefore validates the entire HLS process of SPARK, except for parsing, resource binding and code generation. Our tool is around 7,500 lines of C++ code, whereas SPARK's implementation excluding the parser consists of over 125,000 lines of C++ code. Thus, with around 15 times less effort compared to SPARK's implementation we can build a framework that validates its synthesis process.

We tested our tool on 12 benchmarks obtained from SPARK's test suite. Of these benchmarks, 10 passed and 2 failed. For the ones that passed, our tool was able to quickly find the simulation relation, taking on average around 6 seconds per procedure, and a maximum of 27 seconds for the largest procedure (80 lines of code). Furthermore, the computed bisimulation relations were small, ranging in size from 6 to 29 entries, with an average of about 14. To infer these bisimulation relations, our approach made an average of 189 calls to the theorem prover per procedure (with a minimum of 9 and a maximum of 797). Our approach is compositional since it works on one procedure at a time, and the above results show that our approach can handle realistically size procedures.

As mentioned previously, two benchmarks failed our validation test. Upon further analysis each of them lead us to discover previously unknown bugs in SPARK. One bug occurs in a particular corner case of copy propagation for array elements. The other bug is in the implementation of the code motion algorithm in the scheduler. The fact that our translation validation approach found two previously unknown bugs in a widely-used HLS framework emphasizes the usefulness and bug-isolating capabilities of our tool.

In general, our tool will perform well when the transformations that are performed preserve most of the program's control flow structure. Such transformations are called *structure-preserving transformations* [23]. The only non structure-preserving transformation that SPARK performs is loop unrolling, but in our examples this transformation did not trigger.

## 7 Related Work

Our work is related to translation validation [20, 19, 21, 23, 8, 14], HLS verification [1, 6, 18, 13], and relational approaches to reasoning about programs [7, 3, 15, 2, 11]. Despite long lines of work in each one of these areas, our work distinguishes itself in the following way: it the first to show that translation validation can be effective in the context of a realistic HLS tool. We now discuss each area in more detail.

**Translation Validation.** The technique described in this paper is similar to our previous translation-validation algorithm for CSP programs [14]. The algorithm presented here, however, runs in the context of a realistic HLS tool, as opposed to the more theoretical results from our previous work. Furthermore, our current algorithm handles a more restricted form of concurrency than found in CSP, which allows it to run more efficiently. Our work also bears similarities to Necula's translation-validation algorithm for inferring simulation relations that prove equivalence of sequential programs [19]. Unlike Necula's approach, our algorithm must take into account statements running parallel, since one of the main tasks that HLS tools perform is to schedule statements for parallel execution. Furthermore our algorithm is expressed in terms of calls to a general theorem prover, rather than using specialized solvers and simplifiers. In this sense our algorithm is more modular, since the theorem proving part of the algorithm has been modularized into a component with a very simple interface (it takes a formula and returns *Valid* or *Invalid*).

**HLS verification.** Techniques like correctness-preserving transformations [6], formal assertions [18] and relational approaches for functional equivalence of FSMDs [13, 12] have been used to validate the scheduling step of HLS. However, all these techniques assume that the scheduler does not move code across basic blocks and variable names do not change, which would prevent them from validating SPARK's HLS process. In work that is complementary to ours, model checking was used to validate the binding step of HLS [1], which is the only internal step of SPARK that our tool does not validate.

**Relational approaches.** Relational approaches have been used for a variety of verification tasks, including model checking [7, 3], translation validation [20, 19], and reasoning about optimizations once and for all [15, 2]. In this context, our work can be seen as automating Joseph's relational approach for proving refinement of concurrent systems [11].

## 8 Conclusion and Future Work

We have presented an algorithm for translation validation of the HLS process, and have implemented it within the context of a HLS tool called SPARK. The innovation in our work lies in showing that translation validation approaches work well in the application domain of high-level synthesis. In particular, with only a fraction of the development cost of SPARK, our algorithm can validate the translations performed by SPARK, and it also uncovered bugs that eluded long-term use. Moving forward, we intend to implement translation validation in SPARK for the remaining phases: parsing, binding and code generation. We also intend to adapt our translation validation techniques to SystemC [9] programs.

## References


1. P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama. Verification of RTL generated from scheduled behavior in a high-level synthesis flow. In *ICCAD*, pages 517–524, 1998.
2. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2004*, Jan. 2004.
3. D. Bustan and O. Grumberg. Simulation based minimization. In D. A. McAllester, editor, *CADE 2000*, volume 1831 of *LNCS*. Springer Verlag, 2000.
4. K. M. Chandy. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
5. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of Association Computing Machinery*, 52(3):365–473, May 2005.
6. H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *DATE '99*, NY, USA, 1999.
7. K. Fisler and M. Y. Vardi. Bisimulation and model checking. In *Correct Hardware Design and Verification Methods*, Sept. 1999.
8. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 132(1):53–71, May 2005.
9. T. Grötker. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
10. S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design '03*, 2003.
11. M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3(1):9–18, Mar. 1988.
12. C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade. A formal verification method of scheduling in high-level synthesis. *ISQED*, pages 71–78, 2006.
13. Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machines with datapath (fsmd). In *ISQED '04*, 2004.
14. S. Kundu, S. Lerner, and R. Gupta. Automated refinement checking of concurrent systems. In *ICCAD '07*, 2007.
15. D. Lacey, N. D. Jones, E. V. Wyk, and C. C. Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL '02*, Jan. 2002.
16. M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI 1988*, June 1988.
17. Y.-L. Lin. Recent developments in high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems.*, 2(1):2–21, 1997.
18. N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri. Theorem proving guided development of formal assertions in a resource-constrained scheduler for high-level synthesis. *Form. Methods Syst. Des.*, 19(3):237–273, 2001.
19. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI 2000*, June 2000.
20. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.
21. M. Rinard and D. Marinov. Credible compilation. In *Proceedings of the FLoC Workshop Run-Time Result Verification*, July 1999.
22. R. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, Boston, MA, USA, 1991.
23. L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, Mar. 2003.