Translation Validation of High-Level Synthesis

Sudipta Kundu, Sorin Lerner, and Rajesh K. Gupta, *Fellow, IEEE*

***Abstract*—The growing complexity of systems and their implementation into silicon encourages designers to look for ways to model designs at higher levels of abstraction and then incrementally build portions of these designs—automatically or manually—from these high-level specifications. 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. Therefore, checking if the implementation is a refinement or equivalent to its initial specification is of tremendous value. In this paper, we present an approach to automatically validate the implementation against its initial high-level specification using insights from translation validation, automated theorem proving, and relational approaches to reasoning about programs. In our experiments, we first focus on concurrent systems modeled as communicating sequential processes and show that their refinements can be validated using our approach. Next, we have applied our validation approach to a realistic scenario—a parallelizing high-level synthesis framework called Spark. We present the details of our algorithm and experimental results.**

***Index Terms*—Communicating sequential processes, correctness, equivalence checking, high-level synthesis, refinement checking, translation validation.**

I. Introduction

With the number of transistors on an integrated chip doubling every 18 months for the last three decades, the quantitative changes brought about by the increased chip capacities affect not only the scale of chip designs, but also the scale of the process to design and validate such chips. Despite improvements in the productivity of hardware designers, the rate of improvement has not kept pace with chip capacity growth.

High-level synthesis (HLS) [17], [22], [23], [38], [40], [52] is often seen as a solution to bridge the design-productivity-gap. HLS is the process of generating register transfer level (RTL) design from a high-level behavioral description of a digital system. The synthesis process consists of several interdependent manual or automatic subtasks such as: algorithmic transformations, compilation, scheduling, allocation, binding, and control generation. HLS has been widely explored and relatively mature implementations of various HLS algorithms have started to emerge [23], [38], [52]. 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. Apart from applying a monolithic tool, HLS process is characterized by significant user intervention from recoding to directing the synthesis goals. Consequently, the HLS process, even with automated HLS tools, is error prone and may 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 the HLS process (manual or automatic) 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 the HLS process always produces target RTL designs that are semantically equivalent or refinement to their source versions. However, even if one cannot prove an HLS process correct once and for all, one can try to show, for each translation that HLS performs, that the output program produced by these steps has the same behavior as the original program. Although this approach does not guarantee that the HLS process is bug free, it does guarantee that any errors in translation will be caught when the particular steps of HLS are performed, 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 [19], [42], [46], [47], [53].

During the HLS process, an engineer starts with a high-level description of the design, usually called a specification, which is then refined into progressively more concrete implementations. Checking correctness of these refinement steps has many benefits, including finding bugs in the translation process, while at the same time guaranteeing that properties checked at higher levels in the design are preserved through the refinement process, without having to recheck them at lower levels. For example, if one checks that a given specification satisfies a safety property, and that an implementation is a correct trace refinement of the specification, then the implementation will also satisfy the safety property. The main contribution of this paper is to show, using a novel algorithm, how translation validation can effectively be implemented in a previously unexplored setting, namely HLS. The novelty of our approach comes from the fact that it can account for concurrency which is inherent in hardware design. Our algorithm deals with this concurrency using standard techniques for computing weakest preconditions and strongest postconditions of parallel programs [10].

This paper is an extended and generalized version of our previous work published at ICCAD 2007 [33] and CAV 2008 [34]. This paper extends our previous work in the following ways: 1) it presents a generalization of our previous algorithms so that they support both synchronous and asynchronous semantics of concurrency; 2) it shows the details of our checking algorithm, which have not been previously presented; 3) it presents more detailed experimental results; and 4) it presents more detailed explanations throughout.

Manuscript received July 22, 2009; revised December 31, 2009. Current version published March 19, 2010. This work was supported in part by National Science Foundation under, Grant No. 0 702 792. This paper was recommended by Associate Editor, R. D. (Shawn) Blanton.

S. Kundu is with Synopsys, Inc., Hillsboro, OR 97124-6559 USA (e-mail: sudipta.kundu@synopsys.com).

S. Lerner and R. K. Gupta are with the Department of Computer Science and Engineering, University of California, San Diego, CA 92093-0404 USA (e-mail: lerner@cs.ucsd.edu; gupta@cs.ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2010.2042889

Our translation validation algorithm uses a simulation relation approach to prove refinement. Our algorithm consists of two components. The first component is given a relation, and checks that this relation satisfies the properties required for it to be a correct refinement simulation relation. The second component automatically infers a correct simulation relation just from the specification and the implementation programs. In particular, our inference algorithm automatically establishes a relation that states what points in the implementation program are related to what points in the specification program. This relation guarantees that for each execution sequence in the implementation, an equivalent execution sequence exists in the specification. Apart from refinement checking, we also generalize both of our checking and inference algorithms to prove equivalence between the specification and the implementation programs using a bisimulation relation approach.

To evaluate our approach, we used the Simplify theorem prover [12] to implement our algorithms in a validating system called SURYA. We then used SURYA to check the correctness of a variety of refinements of infinite state concurrent systems represented using communicating sequential processes (CSP) [25] programs. Next, we used SURYA to validate the results of the SPARK HLS tool [23] against the initial behavioral description. Our choice of SPARK is motivated by the fact that it is publicly available at the source code level, and with over 100k lines of code, and over 4000 downloads, represents a state-of-the-art HLS tool. Our validation tool take on an average 6 s to run per procedure, making it possible to do such checking in practical settings. Finally, in running SURYA, 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.

The remainder of this paper is organized as follows. Section II presents an overview of our approach, and shows how our algorithm works on a simple example. Sections III and IV then describes the meaning of refinement and simulation relation respectively. Section V then provides the full details of our algorithms—checking and inference. In particular, Section V-A covers the checking algorithm, which verifies that a given simulation relation is a correct refinement checking relation. Section V-B then presents our inference algorithm, which automatically infers a simulation relation from a specification program and an implementation program. In Section VI, we describe how our algorithms for refinement can be generalized to check for equivalence. Section VII describes our experimental results, Section VIII describes related work, and finally Section IX presents our conclusions and plans for future work.

II. OVERVIEW

At the heart of a HLS process is a model of a digital system consisting of concurrent pieces of functionality, often expressed as sequential program-like behavior, along with synchronous or asynchronous interactions [37], [50]. CSP is a calculus for describing such concurrent systems as a set of processes that communicate synchronously over explicitly named channels. In this paper, we describe our algorithm using CSP-style concurrent programs. While CSP presents a good model for a large number of hardware models described using hardware description languages (HDLs), we note that the core algorithms of our approach do not depend on the choice of the input language. For example, in our experiments we have used our approach for programs that may include arrays, and function calls that are generally not part of CSP programs.

We start out by describing the salient features of CSP required for understanding the examples in this paper. A CSP program is a set of (possibly mutually recursive) process definitions. An asynchronous parallel composition of two processes P and Q is written as $(P \mid\mid Q)$. Asynchronous parallel processes in our version of CSP (and Hoare's original version [25]) can only communicate through messages on channels. Although there are no explicit shared variables, these can easily be simulated using a process that stores the value of the shared variable, and that services reads and writes to the variable using messages. $c?v$ denotes reading a value from a channel c into a variable v and $c!v$ denotes writing a variable v to a channel c. Reads and writes are synchronous. Channels can be visible or hidden. Visible channels are externally observable, and these are the channels that we preserve the behavior of when checking for correctness. We also allow simple C-style control instructions and synchronous parallel composition. By allowing both asynchronous (inherent in CSP) and synchronous semantics of concurrency we support system designs which are globally asynchronous and locally synchronous.

We now present a simple example that illustrates our approach (Fig. 1). For now, ignore the dashed lines in the figure. The specification is a sequential process X shown in Fig. 1(a) using our internal control flow graph (CFG) representation after tail recursion elimination has been performed. We omit the details of the actual CSP code, because the CFG representation is complete, and we believe the CSP code only makes the example harder to follow. This process is continually reading values from an input channel called inp into a variable p and then computes the sum from $(2 \times \mathsf{p} + 1)$ to $\mathsf{10}$ using a loop. Finally, it writes the sum out to a channel named outp. In refinement-based hardware development, the designer often starts with such a high-level description of a sequential design, refining the details of the implementation later on.

An implementation [Fig. 1(b)] may use two separate parallel processes (components) Y and Z, communicating via a hidden channel mid and an acknowledgment channel ack as shown in Fig. 2(b). Like its specification it also takes a value from the inp channel into a variable p and outputs the sum from $(2 \times \mathsf{p} + 1)$ to $\mathsf{10}$ in the outp channel. However, now it does so in two steps, first the process Y multiplies p by 2 and sends it to the component Z then process Z computes the sum and writes it to the outp channel. One additional subtlety of this example is that, in order for the refinement to be correct, an additional channel needs to be added for sending an acknowledgment token (in this case the value 1) back to the process Y, so that a new value isn't read from the inp channel until the current value has been written out to the outp channel. The value read from the ack channel is not used, and so we use an "_" for the variable being read. Instructions on the same transition edge are executed in *parallel* (synchronously).

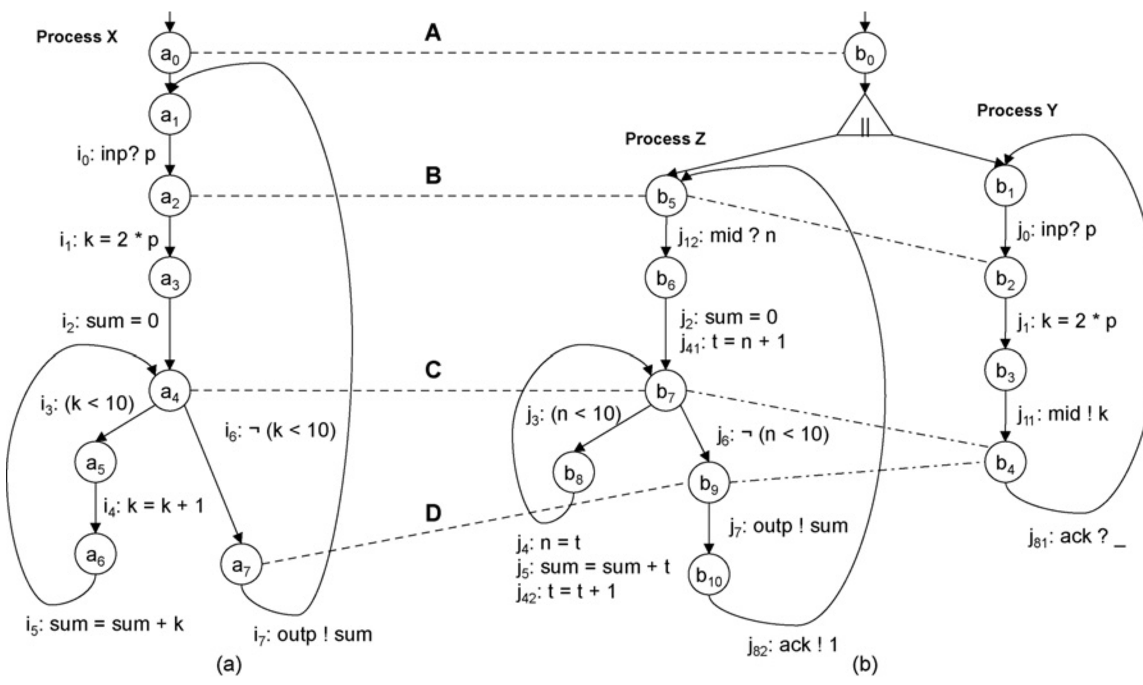

Fig. 1. CFGs of our running example along with a simulation relation. (a) Specification. (b) Implementation.

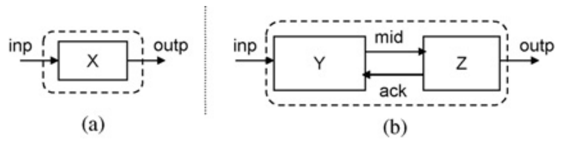

Fig. 2. Communication diagrams of our running example.

TABLE I
SIMULATION RELATION FOR OUR RUNNING EXAMPLE

	(gl_1, gl_2, ϕ)
A.	$(a_0, b_0, true)$
B.	$(a_2, (b_2, b_5), \mathsf{p_s} = \mathsf{p_i})$
C.	$(a_4, (b_4, b_7), \mathsf{k_s} = \mathsf{n_i} \wedge \mathsf{sum_s} = \mathsf{sum_i} \wedge (\mathsf{k_s} + 1) = \mathsf{t_i})$
D.	$(a_7, (b_4, b_9), \mathsf{sum_s} = \mathsf{sum_i})$

Apart from the architectural differences, the loop-structure in the implementation is different from the one in the specification in several ways. First, a loop-shifting transformation has moved 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 [36]. Note 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 the data dependence between them. In addition to loop-shifting, a copy propagation of instruction $\mathsf{j_4}$ to $\mathsf{j_5}$ and $\mathsf{j_{42}}$ is also performed. This ability to make large scale code transformations via parallelizing code transformations as shown here is an important aspect of parallelizing HLS implemented in SPARK. Even without HLS tools, similar source-level transformations are often done manually by the designer to optimize the generated code as a part of high-level design process.

A. Our Approach

Our approach to high-level validation consists of two parts, which theoretically are independent, but for practical reasons, we have made one part subsume the other as explained below. The first part is a *checking algorithm* that, given a relation, determines whether or not it satisfies the properties required for it to be a valid simulation relation. The second part is an *inference algorithm* that infers a relation given two programs, one of which is a specification, and the other is an implementation. To check that one program is a refinement to another, we apply the inference algorithm to infer a relation, and then use the checking algorithm to verify that the resulting relation is indeed the required relation. Because the inference algorithm does a similar kind of exploration as the checking algorithm, this leads to duplicate work. To reduce this work, we have, therefore, made the inference algorithm also perform checking, with only a small amount of additional work. This avoids having the checking algorithm duplicate the exploration work done by the inference algorithm. The checking algorithm is nonetheless useful by itself, in case our inference algorithm is not capable of finding an appropriate relation, and the relation is manually provided by the system designer.

B. Simulation Relation

The goal of the simulation relation in our approach is to guarantee that the specification and the implementation interact in the same way with any surrounding environment that they would be placed in. The simulation relation guarantees that the set of execution sequences of visible instructions in the implementation is a subset of the set of execution sequences in the specification. In what follows, we consider visible instructions to be read and write operations to visible channels. However, in Section VII-B, we define visible instructions to be function calls and return statements.

The simulation relation (defined formally in Section IV) consists of a set of entries of the form (gl_1, gl_2, ϕ), where

gl_1 and gl_2 are program locations in the specification and implementation respectively, and ϕ is a predicate over variables of the specification and implementation. The pair (gl_1, gl_2) captures how the control state of the specification is related to the control state of the implementation, whereas ϕ captures how the data is related. For our running example, the entries in the simulation relation are labeled A–D in Fig. 1, and each entry has a predicate associated with it as shown in Table I.

The first entry 'A' in the simulation relation relates the start location of the specification and the implementation. For this entry, the relevant data invariant is $true$, as we have no information about the states of the programs in those locations. The second entry 'B' shows the specification just as it finishes reading a value from the inp channel. The corresponding control state of the implementation has the Y process in the same state, just as it finishes reading from the inp channel and the other process Z is at the top of its loop. We use subscript s to denote variables in the specification and subscript i for variables in the implementation. For this entry, the relevant data invariant is $p_s = p_i$, which states that the value of p in the specification is equal to the value of p in the implementation. This is because both the specification and the implementation have stored in p the same value from the surrounding environment. In Section II-D, we explain in further detail how our algorithm models the environment as a set of separate processes that are running in parallel with the specification and the implementation. For now, we hide these additional processes for clarity of exposition.

The next entry 'C' in the simulation relation relates the loop head (a_4) in the specification with the loop head (b_7) of the Z process in the implementation. This entry represents 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 guarantees that the two loops produce the same effect on the visible instructions. The data part of this entry guarantees that the two loops are in fact synchronized. Nominally, we need at least one entry in the simulation that "cuts through" every loop pair, in the same way that there must be at least one invariant through each loop when reasoning about a single sequential program.

The last entry 'D' in the simulation relation relates the location a_7 in the specification with the location (b_4, b_9) of the implementation. The relevant invariant for this entry is $\mathsf{sum_s} = \mathsf{sum_i}$, since the specification is about to write sum to the externally visible outp channel and the implementation is about to write sum to the same channel (our correctness criterion).

Simultaneous execution from the last entry 'D' can reach back to 'B', establishing the invariant $p_s = p_i$, since by the time execution reaches the second entry again, both the specification and the implementation would have read the next value from the environment (the details of how our algorithm establishes that the two next values read from the environment processes are equal is explained in Section II-D).

C. Checking Algorithm

The entries in the simulation relation must satisfy some simple local requirements (which are made precise in Section IV). Intuitively, for any entry (gl_1, gl_2, ϕ) in the simulation relation, if the specification and implementation start executing in parallel at control locations gl_1 and gl_2 in states where ϕ holds, and they reach another simulation entry (gl'_1, gl'_2, ϕ'), then ϕ' must hold in the resulting states.

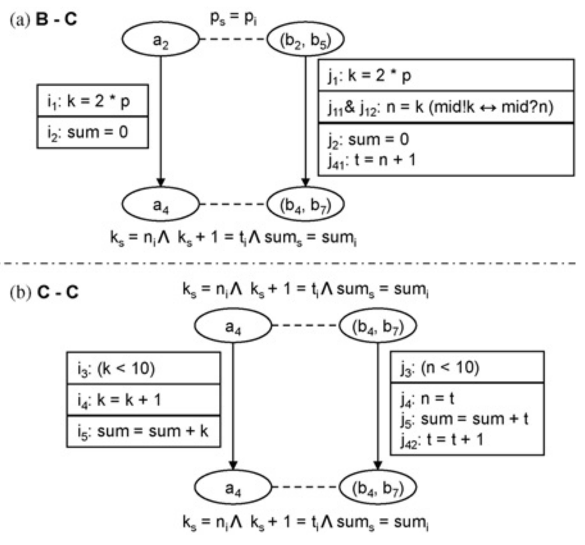

Fig. 3. Checking the simulation relation. (a) Traces from B to C. (b) Traces from C to C.

Given a simulation relation, our checking algorithm checks each entry in the relation individually. For each entry (gl_1, gl_2, ϕ), it finds all other entries that are reachable from (gl_1, gl_2), without going through any intermediate entries. For each such entry (gl'_1, gl'_2, ψ), we check using a theorem prover that if: 1) ϕ holds at gl_1 and gl_2; 2) the specification executes from gl_1 to gl'_1; and 3) the implementation executes from gl_2 to gl'_2, then ψ will hold at gl'_1 and gl'_2.

For our example, the traces in the implementation and the specification from B to C and the trace from C to itself are shown in Fig. 3(a) and (b), respectively. The communication instructions have been transformed into assignments and the original communication instructions are in brackets.

For the B–C path shown in Fig. 3(a), our algorithm uses a theorem prover to validate that if $p_s = p_i$ holds before the two traces, then $k_s = n_i \wedge \mathsf{sum_s} = \mathsf{sum_i} \wedge (k_s + 1) = t_i)$ holds after the traces have been executed. Similarly, it checks the traces from C to C shown in Fig. 3(b) as well as all the other entries in the simulation relation. If there were multiple paths from an entry, our algorithm checks all of them.

D. Inference Algorithm

Our inference algorithm starts by finding the pairs of locations in the implementation and the specification that need to be related in the simulation. 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 or an operation (read or write) on a visible channel. In the example from Fig. 1, our algorithm finds that there is an input, a branch, and an output instruction that must be matched (the specification instructions inp?p and outp!sum should match, respectively, with the implementation instructions inp?p and outp!sum). This amounts to computing

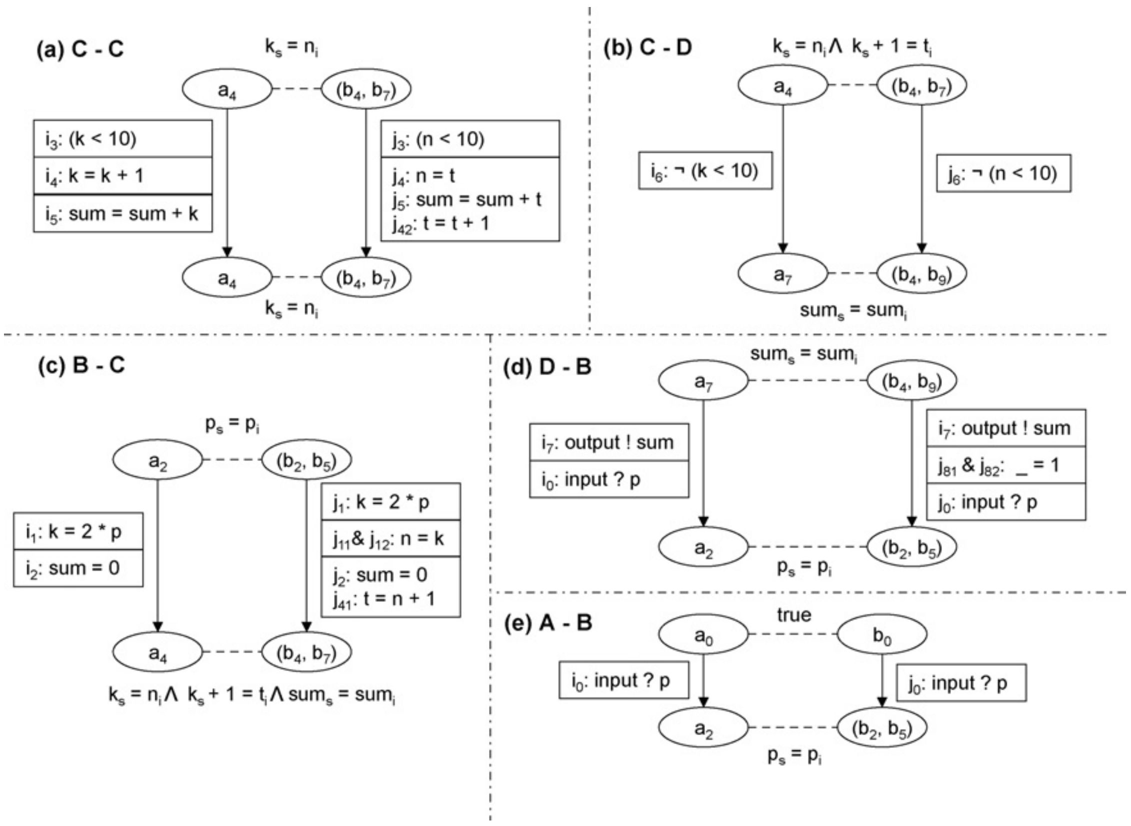

Fig. 4. Steps of the second iteration for computing the simulation relation. (a) C–C. (b) C–D. (c) B–C. (d) D–B. (e) A–B.

TABLE II
ITERATIONS FOR COMPUTING THE SIMULATION RELATION

(gl_1, gl_2)	1st Iteration	2nd Iteration	3rd Iteration (ϕ)
A. (a_0, b_0)	$true$	$true$	$true$
B. $(a_2, (b_2, b_5))$	$p_s = p_i$	$p_s = p_i$	$p_s = p_i$
C. $(a_4, (b_4, b_7))$	$k_s = n_i$	$k_s = n_i \wedge sum_s = sum_i \wedge (k_s + 1) = t_i$	$k_s = n_i \wedge sum_s = sum_i \wedge (k_s + 1) = t_i$
D. $(a_7, (b_4, b_9))$	$sum_s = sum_i$	$sum_s = sum_i$	$sum_s = sum_i$

the first column of Table II. While finding these pairs of locations, our algorithm also does two things. First, it correlates the branch in the specification and the implementation (details of how we establish branch correlations are explained in Section V). Next, it finds the local conditions that must hold for the visible instructions to match. For instructions that output to externally visible channels, the local condition states that the written values in the specification and the implementation must be the same. For example, the local condition for the output instruction is $sum_s = sum_i$.

For instructions that read from externally visible channels, the local condition states that the specification and the implementation are reading from the same point in the conceptual stream of input values. To achieve this, we use an environment process that models each externally visible input channel c as an unbounded array values of input values, with an index variable i stating which value in the array should be read next. This environment process runs an infinite loop that continually outputs values[i] to c and increments i. Assuming that i and j are the index variables from the environment processes that model an externally visible channel c in the specification and the implementation, respectively, then the local condition for matching instructions c?a (in the specification) and c?b (in the implementation) would then be $i_s = j_i$. The equality between the index variables implies that the values being read are the same, and since this fact is always true, we directly add it to the generated local condition, producing $i_s = j_i \wedge a_s = b_i$.

Once the related pairs of locations have been collected, we define for each pair of locations (gl_1, gl_2), a constraint variable $\psi_{(gl_1, gl_2)}$ to represent the state-relating formula that will be computed in the simulation relation for that pair. We then define a set of constraints over these variables to ensure that the would-be simulation relation is a simulation.

There are two kinds of constraints. First, for each pair of locations (gl_1, gl_2) that are related, we want $\psi_{(gl_1, gl_2)}$ to imply that the local condition at those locations hold. For example, $\psi_{(a_7, (b_4, b_9))}$ should imply $sum_s = sum_i$, so that the output values are the same. Such constraints guarantee that the computed simulation 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 (gl_1, gl_2) in states satisfying $\psi_{(gl_1, gl_2)}$, the specification and implementation can execute in parallel to reach another related pair of locations (gl'_1, gl'_2), then $\psi_{(gl'_1,gl'_2)}$ must hold in the resulting states. As shown in Section V, such constraints can be stated over the variables $\psi_{(gl_1,gl_2)}$ and $\psi_{(gl'_1,gl'_2)}$ using the weakest precondition operator (wp). These constraints guarantee that the computed simulation relation is in fact a simulation.

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 simulation relation.

The constraint solving for our example is shown in Table II. Our algorithm first initializes the constraint variables with the local conditions that are required for the visible instructions to be equivalent. Then it chooses any entry from the table, say C, and finds the entries that can reach it (i.e., C and B). Consider the synchronized loop from C to C shown in Fig. 4(a). Our algorithm computes the weakest precondition of the formula at the bottom ($\mathsf{k_s} = \mathsf{n_i}$) over the instructions in the implementation and in the specification, which happens to be $\delta = [(\mathsf{k_s} < 10) \Rightarrow (\mathsf{n_i} < 10) \Rightarrow (\mathsf{k_s} + 1) = \mathsf{t_i}]$. Next, it asks a theorem prover if the condition at the top, i.e., $\mathsf{k_s} = \mathsf{n_i}$ implies δ. Since it does not, our algorithm strengthens the constraint variable at the top with $(\mathsf{k_s} + 1) = \mathsf{t_i}$ which is a stronger condition than δ. A similar pass through Fig. 4(b) strengthens the constraint variable at C with ($\mathsf{sum_s} = \mathsf{sum_i}$). For the other paths, B–C, D–B, and A–B shown in Fig. 4, the theorem prover is able to validate the implication, and as such we do not need to strengthen. Our constraint solving continues in this manner until a fixpoint is reached.

III. Definition of Refinement

We now present a formal description of our approach that builds upon the illustration shown earlier. We assume that the specification and the implementation are single-entry–single-exit programs. We represent each process in the specification and the implementation using a *transition diagram* that describes the control structure of the process in terms of *generalized program locations* and *program transitions*. A generalized program location represents a point of control in the (possibly concurrent) program. A generalized program location is either a node identifier, or a pair of two generalized program locations, representing the state of two processes running in parallel. 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 Σ the 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 Fig. 1 the instruction $\mathsf{i_3}$ has $c(\sigma) = (\sigma(\mathsf{k}) < 10)$ and $f(\sigma) = \sigma$, whereas the instruction $\mathsf{i_2}$ has $c(\sigma) = true$ and $f(\sigma) = \sigma[\mathsf{sum} \mapsto 0]$.

Finally, a transition diagram is defined as follows.

Definition 1 (Transition Diagram): *A transition diagram π is a tuple $(\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, where $\mathcal{L}$ is a finite set of generalized program 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 (gl, i, gl') called transitions, and $\iota \in \mathcal{L}$ is the entry location. We write $gl \xrightarrow{i} gl'$ to denote $(gl, i, gl') \in \rightarrow$. We use $\epsilon \in \mathcal{L}$ to represent the exit location of π.*

Definition 2 (Configuration): *Given a transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, we define a configuration to be a pair $\langle gl, \sigma\rangle$, where $gl \in \mathcal{L}$ and $\sigma \in \Sigma$.*

Definition 3 (Semantic Step): *Given a transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, two configurations $\langle gl, \sigma\rangle$ and $\langle gl', \sigma'\rangle$, and an instruction $i \in \mathcal{I}$, the semantic step relation is defined as follows:*

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

Definition 4 (Execution Sequence): *For a given transition diagram $\pi = (\mathcal{L}, \mathcal{I}, \rightarrow, \iota)$, an execution sequence η starting in configuration $\langle gl_0, \sigma_0\rangle$ is a sequence of configurations such that*

$$\langle gl_0, \sigma_0\rangle \overset{i_1}{\rightsquigarrow} \langle gl_1, \sigma_1\rangle \overset{i_2}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle gl_n, \sigma_n\rangle.$$

We denote by $\mathcal{N}$ the set of all execution sequences. We use the shorthand notation $\eta\langle\pi, gl_0, \sigma_0\rangle$ to represent an execution sequence η starting in configuration $\langle gl_0, \sigma_0\rangle$ in π.

We define ϑ to be the set of *visible instructions*. These are the instructions whose semantics we would like preserved between the specification and implementation. Because our framework is parameterized by the set ϑ of visible instructions, we can apply our framework to various settings. For example, in this discussion we consider visible instructions to be input and output to visible channels. In Section VII-B, however we define visible instructions to be function calls and return statements. 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 σ_1 is equivalent to v_2 in program state σ_2. In the case of channels, two visible instructions are equivalent if they both are inputs, or both outputs on the same channel and their values are the same. In the case of function calls and returns, we say that two function calls are equivalent if the state of globals, the arguments and the address of the called function are the same. Furthermore, we say that two returns are equivalent if the returned value and the state of the globals are the same. This concept of equivalence for visible instruction can be extended to execution sequences as follows.

Definition 5 (Equivalence of Execution Sequences): *Two execution sequences $\eta_1 \in \mathcal{N}$ and $\eta_2 \in \mathcal{N}$ are said to be equivalent, written $\eta_1 \equiv \eta_2$, if the two sequences contain visible instructions that are pairwise equivalent.*

Definition 6 (Refinement of Transition Diagrams): *Given two transition diagrams $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$, we define π_1 to be a refinement of π_2 (written $\pi_1 \sqsubseteq \pi_2$) iff for every $\sigma_1 \in \Sigma$ and $\eta_1\langle\pi_1, \iota_1, \sigma_1\rangle \in \mathcal{N}$ there exists $\sigma_2 \in \Sigma$ and $\eta_2\langle\pi_2, \iota_2, \sigma_2\rangle \in \mathcal{N}$ such that $\eta_1 \equiv \eta_2$.*

IV. Simulation Relation

A *verification relation* between two transition diagrams π_1 and π_2 is a set of triples (gl_1, gl_2, ϕ), where $gl_1 \in \mathcal{L}_1$, $gl_2 \in \mathcal{L}_2$ and ϕ is a predicate over the variables live at locations gl_1 and gl_2. Let the set of such predicates be denoted by $\Phi \overset{def}{=} \Sigma \times \Sigma \rightarrow \mathcal{B}$. We write $\phi(\sigma_1, \sigma_2) = true$ to indicate that ϕ is satisfied in $(\sigma_1, \sigma_2) \in \Sigma \times \Sigma$.

Simulation 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 7 (Cumulative Semantic Step): *Given configurations $\langle gl_0, \sigma_0\rangle$ and $\langle gl_n, \sigma_n\rangle$, and an execution sequence η that contains at least one transition, we define $\rightsquigarrow^+$ as follows:*

$$\langle gl_0, \sigma_0\rangle \overset{\eta}{\rightsquigarrow^+} \langle gl_n, \sigma_n\rangle \quad \textit{iff} \quad \eta = \langle gl_0, \sigma_0\rangle \overset{i_1}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \langle gl_n, \sigma_n\rangle.$$

Definition 8 (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*

$$R(\iota_1, \iota_2, true).$$

$$\forall gl_2, gl_2' \in \mathcal{L}_2, gl_1 \in \mathcal{L}_1, \sigma_1, \sigma_2, \sigma_2' \in \Sigma, \phi \in \Phi, \eta_2 \in \mathcal{N}.$$
$$\begin{bmatrix} \langle gl_2, \sigma_2\rangle \overset{\eta_2}{\rightsquigarrow_2^+} \langle gl_2', \sigma_2'\rangle \wedge \\ R(gl_1, gl_2, \phi) \wedge \phi(\sigma_1, \sigma_2) = true \end{bmatrix} \Rightarrow$$
$$\exists gl_1' \in \mathcal{L}_1, \sigma_1' \in \Sigma, \phi' \in \Phi, \eta_1 \in \mathcal{N}.$$
$$\begin{bmatrix} \langle gl_1, \sigma_1\rangle \overset{\eta_1}{\rightsquigarrow_1^+} \langle gl_1', \sigma_1'\rangle \wedge \\ R(gl_1', gl_2', \phi') \wedge \phi'(\sigma_1', \sigma_2') = true \wedge \eta_1 \equiv \eta_2 \end{bmatrix}.$$

Intuitively, these conditions respectively state that: 1) the entry location of π_1 must be related to the entry location of π_2; and 2) if π_1 and π_2 are in a pair of related configurations, and π_2 can proceed one or more steps producing an execution sequence η_2, then π_1 must also be able to proceed one or more steps, producing a sequence η_1 that is equivalent to η_2, and the two resulting configurations must be related.

The following lemma and theorem connect the above relation with our definition of refinement for transition diagrams (Definition 6).

Lemma 1 (Refinement): *If R is a simulation relation for π_1, π_2, then for each element $(gl_1, gl_2, \psi) \in R$, $\sigma_2 \in \Sigma$, and $\eta_2\langle\pi_2, gl_2, \sigma_2\rangle \in \mathcal{N}$, there exists $\sigma_1 \in \Sigma$, and $\eta_1\langle\pi_1, gl_1, \sigma_1\rangle \in \mathcal{N}$ such that $\eta_1 \equiv \eta_2 \ \wedge \ \psi(\sigma_1, \sigma_2) = true$.*

Theorem 1 (Refinement): *If there exists a simulation relation for π_1, π_2, then $\pi_2 \sqsubseteq \pi_1$.*

The conditions from Definition 8 are used as the base case and the inductive case of a proof by induction showing that π_2 is a refinement of π_1. Thus, a simulation relation is a witness that π_2 is a refinement of π_1.

V. Translation Validation Algorithm

Our translation validation algorithms consists of two parts, checking and inference. To show that a transition diagram is a refinement of another transition diagram, we show there exists a simulation relation. In the following sections, we describe our algorithms for computing a simulation relation.

Given a transition diagram π and a set of locations S, we define the *skipping transition* relation $\hookrightarrow$, which is a version of $\longrightarrow$ that skips over all locations not in S. This transition allows us to focus our attention on only those locations that are in S.

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

$$gl \xhookrightarrow{(w,S)}_{\pi} gl' \quad \textit{iff} \quad \exists gl_1, \cdots, gl_n \in (\mathcal{L} - S) \textit{ such that } gl \xrightarrow{i_0} gl_1 \cdots gl_n \xrightarrow{i_n} gl'.$$

Throughout the rest of this paper, we assume that $\pi_1 = (\mathcal{L}_1, \mathcal{I}_1, \rightarrow_1, \iota_1)$ represents the procedure in the *specification*, and $\pi_2 = (\mathcal{L}_2, \mathcal{I}_2, \rightarrow_2, \iota_2)$ represents the corresponding procedure in the *implementation*. Thus, our goal is to show that π_2 is a refinement of π_1 (i.e., $\pi_2 \sqsubseteq \pi_1$).

A. Checking Algorithm

In this section, we present the details of our algorithm for checking that a verification relation is indeed a correct simulation relation. We let $\mathscr{R} \subseteq \mathcal{L}_1 \times \mathcal{L}_2 \times \Phi$ to be the verification relation that needs to be checked. We first define two sets of locations $\mathcal{P}_1$ and $\mathcal{P}_2$, which are of interest to our algorithm

$$\mathcal{P}_1 = \{gl_1 \mid \exists gl_2, \phi.\ (gl_1, gl_2, \phi) \in \mathscr{R}\}$$
$$\mathcal{P}_2 = \{gl_2 \mid \exists gl_1, \phi.\ (gl_1, gl_2, \phi) \in \mathscr{R}\}.$$

To focus our attention on only those locations in $\mathcal{P}_1$ and $\mathcal{P}_2$, we use the skipping transition relation $\hookrightarrow$. In this section, we use the shorthand notation $gl_1 \xhookrightarrow{w_1}_1 gl_1'$ for $gl_1 \xhookrightarrow{(w_1,\mathcal{P}_1)}_{\pi_1} gl_1'$, and $gl_2 \xhookrightarrow{w_2}_2 gl_2'$ for $gl_2 \xhookrightarrow{(w_2,\mathcal{P}_2)}_{\pi_2} gl_2'$.

Given an entry in $\mathscr{R}$, we then define the *next transition* relation $\twoheadrightarrow$, which traverses the two transition diagrams π_1 and π_2 simultaneously to the next entries reachable from it.

Definition 10 (Next Transition): *Given $(gl_1, gl_2, \phi) \in \mathscr{R}$, $(gl_1', gl_2', \psi) \in \mathscr{R}$, $w_1 \in \mathcal{I}_1^*$ and $w_2 \in \mathcal{I}_2^*$, we define $\twoheadrightarrow$ as follows:*

$$(gl_1, gl_2, \phi) \overset{(w_1,w_2)}{\twoheadrightarrow} (gl_1', gl_2', \psi) \quad \textit{iff} \quad gl_1 \xhookrightarrow{w_1}_1 gl_1' \wedge gl_2 \xhookrightarrow{w_2}_2 gl_2'.$$

For the verification relation $\mathscr{R}$ to be a simulation relation we require it to satisfy certain conditions. In particular, we want the conditions to make sure that the entry locations are related, and the exit locations are related. Furthermore, the conditions should make sure that for every path in the implementation there is a corresponding path in the specification (our refinement criterion). These conditions are made precise by the following definition of *well-formed relation*. If the relation $\mathscr{R}$ is *not* well-formed, then our checking algorithm immediately rejects the verification relation $\mathscr{R}$.

Definition 11 (Well-Formed Relation): *We define the relation $\mathscr{R}$ to be well formed if the following holds:*

1) $(\iota_1, \iota_2, true) \in \mathscr{R}$.
2) $\exists \phi \in \Phi.\ (\epsilon_1, \epsilon_2, \phi) \in \mathscr{R}$.
3) $\forall (gl_1, gl_2, \phi) \in \mathscr{R}, gl_2' \in \mathcal{P}_2, w_2 \in \mathcal{I}_2^*$
$$gl_2 \xhookrightarrow{w_2}_2 gl_2' \Rightarrow \exists gl_1' \in \mathcal{P}_1, \psi \in \Phi.\ (gl_1', gl_2', \psi) \in \mathscr{R}.$$

```
1.  function CheckRelation($\mathscr{R}$)
2.    for each $(gl_1, gl_2, \phi) \in \mathscr{R}$ do
3.      for each $(gl_1, gl_2, \phi) \overset{(w_1, w_2)}{\hookrightarrow\!\!\!\rightarrow} (gl'_1, gl'_2, \psi)$ do
4.        if $\neg$IsInfeasible$(w_1, w_2, \phi)$ then
5.          if $\neg$WellPaired$(w_1, w_2, \phi)$ then
6.            Error("Traces are not well formed")
7.          if ATP$(\phi \Rightarrow$ wp$(w_1,$ wp$(w_2, \psi))) \neq$ Valid then
8.            Error("Cannot verify relation entry")

9.  function IsInfeasible$(w_1 \in \mathcal{I}_1^*, w_2 \in \mathcal{I}_2^*, \phi \in \Phi)$ : Bool
10.   return ATP$(\neg$sp$(w_1,$ sp$(w_2, \phi))) =$ Valid
```

Fig. 5. Algorithm for checking a simulation relation.

The checking algorithm is shown in Fig. 5. The CheckRelation procedure takes as input a well-formed relation $\mathscr{R}$, and verifies each entry in the verification relation individually. For each possible entry (line 2), the algorithm iterates through all the next transitions as shown in line 3. In doing this search, infeasible paths are pruned out on line 4.

The IsInfeasible function (lines 9 and 10), checks using an automated theorem prover (ATP) whether or not it is in fact feasible for the specification to follow trace w_1 and the implementation to follow w_2. The trace combination is infeasible if the strongest postconditions (computed using the sp function) with respect to w_2 and then with respect to w_1 is inconsistent. This takes care of pruning within a single program, but also across the specification and the implementation. For a given formula ϕ and trace w, the strongest postcondition sp(w, ϕ) is the strongest formula ψ such that if the instructions in the trace w are executed in sequence starting in a program state satisfying ϕ, then ψ will hold in the resulting program state. The sp computation itself is standard, except for the handling of communication instructions, which are simulated as assignments. When computing sp 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.

Once we have identified that the two sequences w_1 and w_2 may be a feasible combination, we check that they are well formed using the WellPaired predicate (lines 5 and 6). The WellPaired predicate (not shown here) checks that there is at most one visible instruction in the sequences w_1 and w_2. It also checks that the visible instructions are equivalent.

Next, for well formed sequences, we check that if we start at states that satisfy the predicate ϕ and execute w_1 in π_1 and w_2 in π_2 then the resulting states should satisfy the predicate ψ. To do this, we first compute the weakest precondition of ψ with respect to the two traces, and then ask an ATP to show ϕ implies it (line 7). We perform the weakest precondition computation on one trace and then the other. For a given formula ψ and trace w, the weakest precondition wp(w, ψ) is the weakest formula ϕ such that executing the trace w in a state satisfying ϕ leads to a state satisfying ψ. Here again, the wp computation itself is standard, and the order in which we process the two traces does not matter. If at the end of the algorithm there is no error then the verification relation is indeed a simulation relation.

There are additional optimizations we perform that are not explicitly shown in the algorithm from Fig. 5. These are however important in improving the efficiency of our refinement checking process. When exploring the control state (both in the checking and in the inference algorithm), we perform a simple partial order reduction [45] that is very effective in reducing the size of the control state space: if two communication instructions happen in parallel, but they do not depend on each other, and they do not involve externally visible channels, then we only consider one ordering of the two instructions.

B. Inference Algorithm

Since there can be many possible paths through a loop, writing simulation relations by hand can be tedious, time consuming and error prone. We therefore need methods for generating these relations automatically, not just checking them. This in turn also allow us to automate the validation process entirely. Nevertheless, our checking algorithm is useful by itself, in case our inference algorithm is not capable of finding an appropriate relation, and a human wants to provide the relation by hand.

Here again to focus our attention on only those locations for which our approach infers the relation entries, we define two sets of locations $\mathcal{Q}_1$ and $\mathcal{Q}_2$ for the transition diagrams π_1 and π_2 respectively. These include all locations corresponding to visible instructions and also all locations before branch statements. In this section, we do notation abuse by reusing the shorthand $gl_1 \overset{w_1}{\hookrightarrow}_1 gl'_1$ for $gl_1 \overset{(w_1, \mathcal{Q}_1)}{\hookrightarrow}_{\pi_1} gl'_1$, and $gl_2 \overset{w_2}{\hookrightarrow}_2 gl'_2$ for $gl_2 \overset{(w_2, \mathcal{Q}_2)}{\hookrightarrow}_{\pi_2} gl'_2$.

We now define a parallel transition relation $\hookrightarrow\!\!\!\rightarrow$ that essentially traverses the two transition diagrams (specification and implementation) in synchrony, while focusing on only those locations for which our approach infers the relation entries.

Definition 12 (Parallel Transition): *Given* $(gl_1, gl_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$, $(gl'_1, gl'_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$, $w_1 \in \mathcal{I}_1^*$ *and* $w_2 \in \mathcal{I}_2^*$, *we define* $\hookrightarrow\!\!\!\rightarrow$ *as follows:*

$$\begin{aligned}(gl_1, gl_2) \overset{(w_1, w_2)}{\hookrightarrow\!\!\!\rightarrow} (gl'_1, gl'_2) \quad & \textit{iff} \\ gl_1 \overset{w_1}{\hookrightarrow}_1 gl'_1 \ \wedge\ gl_2 \overset{w_2)}{\hookrightarrow}_2 gl'_2 \ \wedge \\ \mathsf{Rel}(w_1, w_2, gl_1, gl_2) \ \wedge\ \mathsf{WellMatched}(w_1, w_2).\end{aligned}$$

We now describe the two predicates Rel and WellMatched used in the above definition. The predicate Rel : $\mathcal{I}^* \times \mathcal{I}^* \times \mathcal{Q}_1 \times \mathcal{Q}_2 \rightarrow \mathcal{B}$ 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 Fig. 1 and the corresponding branch in the implementation. For any two such branches, the 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, Rel(w_1, w_2, gl_1, gl_2) returns true only if the paths w_1 and w_2 follow branches in a correlated way.

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 Fig. 1, since the strongest postcondition involves the input parameter p, the theorem prover is unable to reason about it. However, because the structure of the branch predicate is not changed in the implementation, Rel can conclude that the two branches go in the same direction.

The other predicate WellMatched : $\mathcal{I}^* \times \mathcal{I}^* \rightarrow \mathcal{B}$ prunes some of these pair of transitions if the sequence of instructions are not similar (well-matched). We say two sequences (w_1, w_2) of instructions are well-matched if neither of them contain a visible instruction or they each contains a single visible instruction of the same type; i.e., they are both input or both output on the same channel. Although Rel and WellMatched make guesses about the correlation of branches and visible instructions, the later constraint solving phase of our approach makes sure that these guesses are correct.

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

Definition 13 (Pairs of Interest): *The relation $\mathcal{R} \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$ is defined to be the minimal relation that satisfies the following three properties:*

$$\mathcal{R}(\iota_1, \iota_2)$$
$$\mathcal{R}(\epsilon_1, \epsilon_2)$$
$$\left[\mathcal{R}(gl_1, gl_2) \ \wedge \ (gl_1, gl_2) \xhookrightarrow{(w_1,w_2)} (gl'_1, gl'_2)\right] \Rightarrow \mathcal{R}(gl'_1, gl'_2).$$

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

For our approach to successfully infer a simulation relation, the computed set $\mathcal{R}$ must cover every path in the implementation (our refinement criterion). This condition is made precise by the following definition of *well-formed pairs of interest*. The well-formed condition here is similar to the one described in Definition 11, except that now it is for the pairs of interest relation $\mathcal{R}$. We do not need the first two conditions here as they are satisfied by construction. Here again if the computed set $\mathcal{R}$ is *not* well-formed, then our validation approach immediately rejects the translation from specification to implementation.

Definition 14 (Well-Formed Pairs of Interest): *We define the pairs of interest relation $\mathcal{R}$ to be well formed if the following holds:*

$$\forall (gl_1, gl_2) \in \mathcal{R},\, gl'_2 \in \mathcal{Q}_2,\, w_2 \in \mathcal{I}_2^*$$
$$gl_2 \xhookrightarrow{w_2}_2 gl'_2 \ \Rightarrow \ \exists gl'_1 \in \mathcal{Q}_1.\ (gl'_1, gl'_2) \in \mathcal{R}.$$

We now describe our inference algorithm in terms of constraint solving. In particular, for each $(gl_1, gl_2) \in \mathcal{R}$ we define a constraint variable $\psi_{(gl_1, gl_2)}$ representing the predicate that we want to compute for the simulation entry (gl_1, gl_2). We denote by Ψ the set of all such constraint variables. Using these constraint variables, the final simulation relation will have the form

$$\{(gl_1, gl_2, \psi_{(gl_1,gl_2)}) \mid \mathcal{R}(gl_1, gl_2)\}.$$

To compute the predicates that the constraint variables $\psi_{(gl_1,gl_2)}$ stand for, we define a set of constraints on these variables, and then solve the constraints. The constraints are defined as follows.

Definition 15 (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.*

Definition 16 (Set of Constraints): *The set $\mathcal{C}$ of constraints is defined by*

For each $(gl_1, gl_2) \xhookrightarrow{(w_1,w_2)} (gl'_1, gl'_2)$:

$$\left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{CreateSeed}(w_1, w_2)\right] \in \mathcal{C}$$
$$\left[\psi_{(gl_1,gl_2)} \Rightarrow \mathsf{wp}(w_1, \mathsf{wp}(w_2, \psi_{(gl'_1,gl'_2)}))\right] \in \mathcal{C}.$$

The CreateSeed function above creates for each pair of instruction sequences (w_1, w_2) a formula, which does not refer to any constraint variables. There are two cases, either they are well-matched or they are branches (Definition 12). If the instructions are well-matched, then the formula returned by CreateSeed states that the visible instructions in them are equivalent as defined in Section III; and if they 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. The weakest precondition computation is the same as the one described in Section V-A.

Having created a set of constraints $\mathcal{C}$, our validation approach now solves these constraints using the algorithm in Fig. 6. The algorithm starts by setting each constraint variable to *true* (line 13) and initializing a *worklist* with the set of all constraints (line 14). Next, while the *worklist* is not empty, it removes a constraint from the worklist (line 16), and checks using a theorem prover if it is *Valid* (line 17). If not, then it appropriately strengthens the constraint variable in the left-hand side of the implication for the given constraint (line 20) and adds it to the worklist of all the constraints that have this constraint variable in the right-hand side (lines 21–22).

```
11. function SolveConstraints($\mathcal{C}$)
12.   for each $(gl_1, gl_2) \in \mathcal{R}$ do
13.     $\psi_{(gl_1,gl_2)} := true$
14.   let $worklist := \mathcal{C}$
15.   while $worklist$ not empty do
16.     let $[\psi_{(gl_1,gl_2)} \Rightarrow f(\psi_{(gl'_1,gl'_2)})] := worklist$.Remove
17.     if ATP$(\psi_{(gl_1,gl_2)} \Rightarrow f(\psi_{(gl'_1,gl'_2)})) \neq Valid$ then
18.       if $(gl_1, gl_2) = (\iota_1, \iota_2)$ then
19.         Error("Start Condition not strong enough")
20.       $\psi_{(gl_1,gl_2)} := \psi_{(gl_1,gl_2)} \wedge f(\psi_{(gl'_1,gl'_2)})$
21.       $worklist := worklist \cup$
22.           $\{c \in \mathcal{C} \mid \exists \psi, g \ .\ c = [\psi \Rightarrow g(\psi_{(gl_1,gl_2)})]\}$
```

Fig. 6. Algorithm for solving constraints.

VI. Equivalence of Transition Diagrams

Apart from checking refinements, we also sometimes want to check equivalence between two transition diagrams. In this section, we describe how we can generalize our algorithms to check for equivalence. We first define two transition diagrams to be equivalent as follows:

Definition 17 (Equivalence of Transition Diagrams): *Two transition diagrams π_1 and π_2 are said to be equivalent iff $\pi_1 \sqsubseteq \pi_2$ and $\pi_2 \sqsubseteq \pi_1$.*

We define a *bisimulation relation* using the definition of simulation relation.

Definition 18 (Bisimulation Relation): *A verification relation R is a bisimulation relation for π_1, π_2 iff R is a simulation relation for π_1, π_2 and $R^{-1} = \{(gl_2, gl_1, \phi) \mid R(gl_1, gl_2, \phi)\}$ is a simulation relation for π_2, π_1.*

The following theorem connects the above relation with our definition of equivalence for transition diagrams.

Theorem 2 (Equivalence): *If there exists a bisimulation relation for π_1, π_2, then π_1 and π_2 are equivalent.*

Like simulation relation, a bisimulation relation is a witness that two transition diagrams are equivalent. Therefore, to check if the specification is equivalent to the implementation our algorithms now have to show that there exists a bisimulation relation between them. We can use both our checking and inference algorithms for this purpose with just slight modifications.

For the checking algorithm, we only have to strengthen the definition of well-formed relation (Definition 11) with this fourth condition

$$\forall (gl_1, gl_2, \phi) \in \mathscr{R},\ gl_1' \in \mathcal{P}_1,\ w_1 \in \mathcal{I}_1^*$$
$$gl_1 \xhookrightarrow{w_1}_1 gl_1' \ \Rightarrow\ \exists gl_2' \in \mathcal{P}_2, \psi \in \Phi.\ (gl_1', gl_2', \psi) \in \mathscr{R}.$$

Similarly, for the inference algorithm, we only have to strengthen the definition of well-formed pairs of interest (Definition 14) with this condition

$$\forall (gl_1, gl_2) \in \mathcal{R},\ gl_1' \in \mathcal{Q}_1,\ w_1 \in \mathcal{I}_1^*$$
$$gl_1 \xhookrightarrow{w_1}_1 gl_1' \ \Rightarrow\ \exists gl_2' \in \mathcal{Q}_2.\ (gl_1', gl_2') \in \mathcal{R}.$$

VII. Evaluation

We implemented our algorithms in a tool called Surya using the Simplify ATP [12]. We have used Surya to validate programs in two different settings. First, we used it to *automatically* check refinements of various concurrent programs, written in CSP. Next, we used Surya to validate the result of the HLS framework Spark.

A. Automatic Refinement Checking of CSP Programs

For refinements our goal is to infer a simulation relation (if possible). The visible instructions in this case are input and output on visible channels. We wrote a variety of CSP refinements, and checked them for correctness automatically. The refinements that we checked are shown in Table III, along with the number of parallel threads, the number of instructions, the number of simulation relation entries, the number of calls to the theorem prover, and the time required to automatically infer and check them. Apart from the theorem prover calls discussed in this paper, we also use the theorem prover to reduce the size of the formulas used in our algorithms. The number of calls to the theorem prover mentioned in Table III include *all* these calls.

TABLE III
Timings for the Refinement Examples Checked Using Our Tool

Description	T	I	SRE	TP	Time (mins)
1. Simple buffer	7	29	3	14	00.00
2. Simple vending machine	2	20	9	32	00.00
3. Cyclic scheduler	6	65	157	11 082	00.49
4. Student tracking system	3	63	12	115	00.01
5. 1 comm link	11	54	3	14	00.01
6. 2 parallel comm links	18	105	37	486	00.04
7. 3 parallel comm links	25	144	45	1861	00.21
8. 4 parallel comm links	32	186	124	7228	01.11
9. 5 parallel comm links	39	228	315	24 348	02.32
10. 6 parallel comm links	46	270	762	74 991	08.29
11. 7 parallel comm links	53	312	1785	217 131	37.28
12. SystemC refinement	8	39	3	14	00.00
13. EP2 system	3	173	208	5648	01.47

T: number of parallel threads; I: number of instructions; SRE: number of simulation relation entries; TP: Number of theorem prover calls.

The first 11 refinements were inspired from examples that come with the failures-divergence refinement (FDR) tool [39]. FDR is a state-of-the-art tool to check CSP refinements. The approach that FDR uses for checking refinement is to perform an exhaustive search of the implementation-specification combined state space. Although in its pure form this approach only works for finite state systems, there is one way in which it can be extended to infinite systems. In particular, if an infinite state system treats all the data it manipulates as black boxes, then one can use skolemization and simply check the refinement for one possible value. Such systems are called *data-independent*, and FDR can check the refinement of these systems using the skolemization trick, even if they are infinite [48].

Unfortunately, for high-level programs, there are many refinement examples that are not finite, because they do not specify the bit-width of integers (in particular, we want the refinement to work for any integer size). Nor are the processes data-independent, as they manipulate the data during the refinement process. In particular, our example from Fig. 1 is neither finite nor data-independent, since both the specification and the implementation are "inspecting" the variables when manipulating them. Indeed, it would not at all be safe to simply check the refinement for any one particular value, since, if we happen to pick 0 for p, and the implementation erroneously sets the output to four times the input (instead of two times), we would not detect the error. FDR cannot check the refinement of such infinite data-dependent CSP systems, except by restricting them to a finite subset first, for example by picking a bit-width for the integers, and then doing an exhaustive search. Not only would such an approach not prove the refinement for any bit-width, but furthermore, despite many techniques that have been developed for checking larger and larger finite state spaces [6], [11], [45], [49], the state space can still grow to a point where automation is impossible. For example, we tried checking the refinement example '2 parallel comm links' from Table III in FDR using 32-bit integers as

TABLE IV
SPARK BENCHMARKS SUCCESSFULLY CHECKED

Benchmarks	No. of Bisimulation Relation Entries	No. of Calls to Theorem Prover	Time (s)
1. Incrementer	6	9	00.52
2. Integer-sum	6	20	00.81
3. Array-sum	6	24	00.83
4. Diffeq	7	41	01.68
5. Waka	11	79	02.61
6. Pipelining	12	75	02.30
7. Rotor	14	71	02.57
8. Parker	26	281	05.23
9. S2r	27	570	26.73
10. Findmin8	29	787	14.86

values, and the tool had to be stopped because it ran out of memory after several hours (our tool, in contrast, is able to check this example for any sized integers, not just 32-bit integers, in about 4 s).

We implemented generalizations of these 11 FDR examples to make them data-dependent and operate over infinite domains. We were able to check these generalized refinements that FDR would not be able to check.

The 12th refinement in the list is a hardware refinement example taken from a SystemC book [21]. This example models the refinement of an abstract first-in-first-out (FIFO) communication channel to an implementation that uses a standard FIFO hardware channel, along with logic to make the hardware channel correctly implement the abstract communication channel.

In the 13th refinement from Table III, we checked part of the EP2 system [1], which is a new industrial standard for electronic payments. We followed the implementation of the data part of the EP2 system found in a recent paper on CSP-PROVER [27]. The EP2 system states how various components, including service centers, credit card holders, and terminals, interact.

In all of the above examples, we check for trace subset refinement (see Definition 6). Since trace subset refinement preserves safety properties, we can also conclude that the implementation has all the safety properties of the specification.

We also have a large test suite of hand-written *incorrect* refinements that we run our tool on, to make sure that our tool indeed detects these as incorrect refinements.

B. SPARK*: HLS Framework*

SPARK is a C-to-VHDL parallelizing HLS framework that employs a set of compiler, parallelizing compiler, and synthesis transformations to improve the quality of HLS results. Fig. 7 shows an overview of the SPARK HLS framework. What makes SPARK an excellent candidate for experimenting is not only the easy availability of source code but also the fact that it uses a single intermediate representation (IR), called hierarchical task graphs [18]. 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 subexpression 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.

Our tool SURYA takes as input the IR program that is produced by the parser, and the IR program right before resource binding (see Fig. 7), and verifies that the two are equivalent by showing that there exist a bisimulation relation. Our tool therefore validates the entire HLS process of SPARK, except for parsing, resource binding and code generation. Note that SURYA is around 7500 lines of C++ code, whereas SPARKs implementation excluding the parser consists of over 125 000 lines of C++ code. Thus, with around 15 times less effort compared to SPARKs implementation we can build a framework that validates its synthesis process.

We tested our tool on 12 benchmarks obtained from SPARKs test suite. Of these benchmarks, ten passed and two failed. The benchmarks that were successfully checked are shown in Table IV, along with the number of bisimulation relation entries, the number of calls to the theorem prover, and the time required to check each benchmark. All these benchmarks are single threaded. For the ones that passed, our tool was able to quickly find the bisimulation relation, taking on average around 6 s per procedure, and a maximum of 27 s 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. We note that both the bugs are, in retrospect, typical of the errors in such a code consisting of complex compiler transformations. The fact that these bugs were found in a well-used HLS framework indicates the usefulness 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* [53]. The only non-structure-preserving transformation that SPARK performs is loop unrolling, but in our examples this transformation did not trigger.

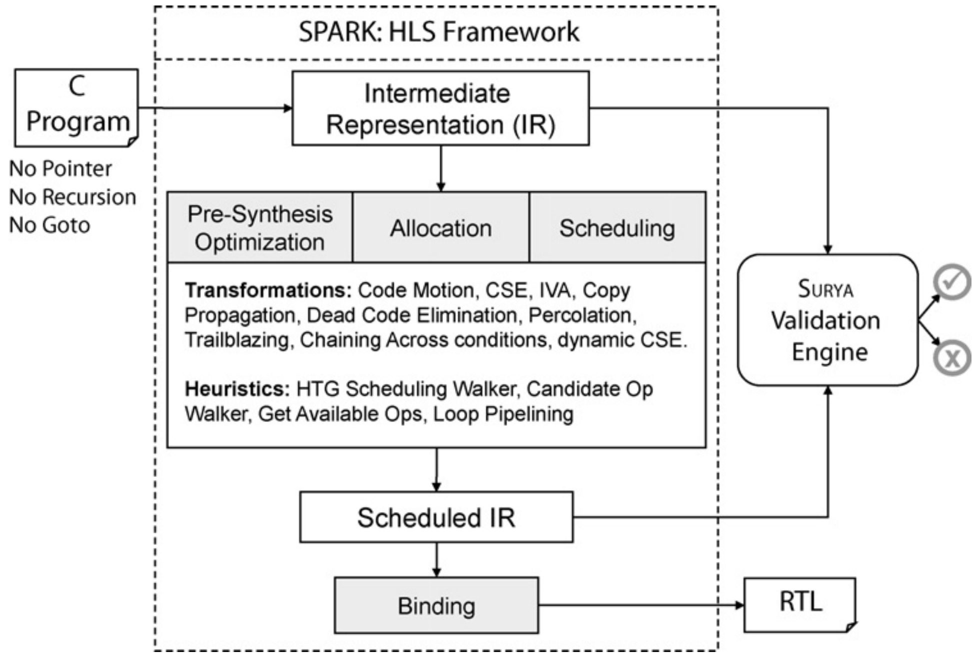
SPARK: HLS Framework
C Program
No Pointer
No Recursion
No Goto
Intermediate Representation (IR)
Pre-Synthesis Optimization
Allocation
Scheduling
Transformations: Code Motion, CSE, IVA, Copy Propagation, Dead Code Elimination, Percolation, Trailblazing, Chaining Across conditions, dynamic CSE.
Heuristics: HTG Scheduling Walker, Candidate Op Walker, Get Available Ops, Loop Pipelining
Scheduled IR
Binding
RTL
SURYA Validation Engine

Fig. 7. Overview of the SPARK framework along with SURYA.

VIII. RELATED WORK

Our work is related to translation validation [19], [33], [42], [46], [47], [53], [54], relational approaches to reasoning about programs [5], [7], [15], [28], [35], CSP refinement checking [13], [27], [39], [51], and HLS verification [2], [14], [31], [34], [41]. We now discuss each area in more detail.

A. Translation Validation

Our inference algorithm was inspired by Necula's translation validation algorithm for inferring simulation relations that prove equivalence of sequential programs [42]. Necula's approach collects a set of constraints in a forward scan of the two programs, and then solves these constraints using a specialized solver and expression simplifier. Unlike Necula's approach, our algorithm must take into account statements running parallel, since hardware is inherently concurrent and 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*). This allows us to easily substitute the current Simplify theorem prover with another one.

B. Relational Approaches

Relational approaches are a common tool for reasoning about programs, and they have been used for a variety of verification tasks, including model checking [7], [15], translation validation [42], [46], and reasoning about optimizations once and for all [5], [35]. In this context, our work is inspired by Josephs's approach [28] for proving refinements. However, Josephs proved refinements by hand, whereas our tool is fully automated.

C. CSP Refinement Checking

There has been a long line of work on reasoning about refinement of CSP programs. Our searching algorithm through the control state of the program is similar to FDRs searching technique [39], which exhaustively explores the state space. However, as mentioned previously, our tool can handle infinite state spaces that do not trivially reduce using skolemization to finite state spaces. We achieve this by capturing the possibly infinite state space of data using formulas and using a theorem prover to reason about these formulas. Although, this technique is well known and has been used in dataflow analysis [16], [20], model checking [4], [8], [9], [24], and translation validation [42], [46]. The use of this technique in the context of checking CSP refinements appears to be novel.

Various interactive theorem provers have been extended with the ability to reason about CSP programs. As one example, Dutertre and Schneider [13] reasoned about communication protocols expressed as CSP programs using the PVS theorem prover [43]. As another example, Tej and Wolff [51] have used the Isabelle theorem prover [44] to encode the semantics of CSP programs. Isabelle has also been used by Isobe and Roggenbach to develop a tool called CSP-PROVER [27] for proving properties of CSP programs. All these uses of interactive theorem provers follow a common high-level approach: the semantics of CSP is usually encoded using the native logic of the interactive theorem prover, and then a set of tactics are defined for reasoning about this semantics. Users of the system can then write proof scripts that use these tactics, along with built-in tactics from the theorem prover, to prove properties about particular CSP programs. Our approach does not have the same level of formal underpinnings as these interactive theorem proving approaches. However, our approach is fully automated, whereas these interactive theorem proving approaches all require some amount of human intervention.

Our tool checks one particular property of CSP programs, namely trace subset refinement. This kind of refinement only preserves safety properties. Algorithms and tools exist for checking other kinds of refinements. For example, CSP-PROVER [27] can check refinements using a failures semantics that preserves liveness properties and deadlock freedom (in addition to safety properties). The FDR [39] tool can also check refinements in a failures/divergence model, which can also preserve livelock freedom.

D. HLS Verification

Techniques like correctness-preserving transformations [14], formal assertions [41], symbolic simulation [3], and relational approaches for functional equivalence of finite state machines with datapaths [29]–[32] have been used to validate the scheduling step of HLS. In contrast to these approaches our algorithm can handle most of the transformations currently being used for HLS in an uniform and modular framework. Our approach is also able to automatically infer the equivalence between the specification and the implementation without relying on any hints from the HLS tool. Moreover, most of 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 SPARKs HLS process. Also, in work that is complementary to ours, model checking was used to validate the binding step of HLS [2], which is the only internal step of SPARK that our tool does not validate.

IX. CONCLUSION AND FUTURE WORK

We have presented an automated algorithm for translation validation of the HLS process. The proposed algorithm is implemented in a validation system called SURYA and we demonstrated its effectiveness through its application in two different settings. The innovation in our work lies in showing that translation validation approaches work well in the application domain of HLS.

Our experiments with SPARK showed that 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. Our work also solves the critical problem of handling more sophisticated datatypes than finite bit-width enumeration types associated with typical RTL code and thus enables stepwise refinement of system designs expressed using high-level languages. Moving forward, we plan 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 [26] programs.

ACKNOWLEDGMENT

The authors would like to thank A. Hu, F. Doucet, and the anonymous reviewers for their invaluable feedback on earlier drafts of this paper.

REFERENCES

[1] *EP2: Electronic Payment 2* [Online]. Available: http://www.eftpos2000.ch

[2] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama, "Verification of RTL generated from scheduled behavior in a high-level synthesis flow," in *Proc. IEEE/Assoc. Comput. Machinery Int. Conf. Comput.-Aided Design*, 1998, pp. 517–524.

[3] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya, "Verification of scheduling in the presence of loops using uninterpreted symbolic simulation," in *Proc. IEEE Int. Conf. Comput. Design*, 1999, pp. 458–466.

[4] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani, "Automatic predicate abstraction of C programs," in *Proc. Assoc. Comput. Machinery Special Interest Group Programm. Languages (SIGPLAN) Conf. Programm. Language Design Implement.*, Jun. 2001, pp. 203–213.

[5] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proc. 31st Assoc. Comput. Machinery Symp. Principles Programm. Languages*, Jan. 2004, pp. 14–25.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proc. 5th Ann. IEEE Symp. Logic Comput. Sci.*, 1990, pp. 1–33.

[7] D. Bustan and O. Grumberg, "Simulation-based minimization," in *Proc. Int. Conf. Automated Deduction*, LNCS 1831. 2000, pp. 255–270.

[8] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha, "Concurrent software verification with states, events and deadlocks," *Formal Aspects Comput. J.*, vol. 17, no. 4, pp. 461–483, Dec. 2005.

[9] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 388–402, Jun. 2004.

[10] K. M. Chandy, *Parallel Program Design: A Foundation*. Boston, MA: Addison-Wesley Longman, 1988.

[11] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *Computer Hardware Description Languages and their Applications*, D. Agnew, L. Claesen, and R. Camposano, Eds. Amsterdam, The Netherlands: Elsevier, 1993, pp. 87–100.

[12] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking," *J. Assoc. Comput. Machinery*, vol. 52, no. 3, pp. 365–473, May 2005.

[13] B. Dutertre and S. Schneider, "Using a PVS embedding of CSP to verify authentication protocols," in *Proc. 10th Int. Conf. Theorem Proving Higher Order Logics*, 1997, pp. 121–136.

[14] H. Eveking, H. Hinrichsen, and G. Ritter, "Automatic verification of scheduling results in high-level synthesis," in *Proc. Conf. Design, Automat. Test Eur.*, 1999, p. 12.

[15] K. Fisler and M. Y. Vardi, "Bisimulation and model checking," in *Proc. 10th Conf. Correct Hardware Design Verification Methods*, Sep. 1999, pp. 338–341.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proc. Assoc. Comput. Machinery Special Interest Group Programm. Languages (SIGPLAN) Conf. Programm. Language Design Implement.*, Jun. 2002, pp. 234–245.

[17] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer, 1992.

[18] M. Girkar and C. D. Polychronopoulos, "Automatic extraction of functional parallelism from ordinary programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[19] B. Goldberg, L. Zuck, and C. Barrett, "Into the loops: Practical issues in translation validation for optimizing compilers," *Electron. Notes Theor. Comput. Sci.*, vol. 132, no. 1, pp. 53–71, May 2005.

[20] S. Graf and H. Saidi, "Construction of abstract state graphs of infinite systems with PVS," in *Proc. Int. Conf. Comput. Aided Verif.*, Jun. 1997, pp. 72–83

[21] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, 2002.

[22] R. Gupta and F. Brewer, "High-Level Synthesis: A Retrospective," in *High-Level Synthesis from Algorithm to Digital Circuit*. Berlin, Germany: Springer, 2008, pp. 13–28.

[23] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. Int. Conf. Very-Large-Scale Integr. Design*, 2003, p. 461.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy abstraction," in *Proc. 29th Assoc. Comput. Machinery Symp. Principles Programm. Languages*, Jan. 2002, pp. 58–70.

[25] C. A. R. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice Hall, 1985.

[26] *IEEE Standard 1666 SystemC Language Reference Manual*, Open SystemC Initiative, 2005 [Online]. Available: http://www.systemc.org

[27] Y. Isobe and M. Roggenbach, "A generic theorem prover of CSP refinement," in *Proc. 11th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, LNCS 1503. Apr. 2005, pp. 103–123.

[28] M. B. Josephs, "A state-based approach to communicating processes," *Distrib. Comput.*, vol. 3, no. 1, pp. 9–18, Mar. 1988.

[29] C. Karfa, C. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," in *Proc. IEEE Int. Symp. Quality Electron. Design*, 2006, pp. 71–78.
[30] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An equivalence-checking method for scheduling verification in high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 3, pp. 556–569, Mar. 2008.
[31] Y. Kim, S. Kopuri, and N. Mansouri, "Automated formal verification of scheduling process using finite state machines with datapath (FSMD)," in *Proc. 5th Int. Symp. Quality Electron. Design*, 2004, pp. 110–115.
[32] Y. Kim and N. Mansouri, "Automated formal verification of scheduling with speculative code motions," in *Proc. 18th Assoc. Comput. Machinery Great Lakes Symp. Very-Large-Scale Integr.*, 2008, pp. 95–100.
[33] S. Kundu, S. Lerner, and R. Gupta, "Automated refinement checking of concurrent systems," in *Proc. 2007 IEEE/Assoc. Comput. Machinery Int. Conf. Comput.-Aided Design*, 2007, pp. 318–325.
[34] S. Kundu, S. Lerner, and R. Gupta, "Validating high-level synthesis," in *Proc. 20th Int. Conf. Comput.-Aided Verificat.*, 2008, pp. 459–472.
[35] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Proving correctness of compiler optimizations by temporal logic," in *Proc. 29th Assoc. Comput. Machinery Symp. Principles Programm. Languages*, Jan. 2002, pp. 283–294.
[36] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. 1988 Assoc. Comput. Machinery SIGPLAN Special Interest Group Programm. Languages (SIGPLAN) Conf. Programm. Language Design Implement.*, Jun. 1988, pp. 318–328.
[37] E. A. Lee and A. L. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.
[38] Y.-L. Lin, "Recent developments in high-level synthesis," *Assoc. Comput. Machinery Trans. Design Automat. Electron. Syst.*, vol. 2, no. 1, pp. 2–21, 1997.
[39] *Failures-Divergence Refinement: FDR2 User Manual*, Formal Systems (Europe) Ltd., Oxford, U.K., Jun. 2005.
[40] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[41] 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," *Formal Methods Syst. Design*, vol. 19, no. 3, pp. 237–273, 2001.
[42] G. C. Necula, "Translation validation for an optimizing compiler," in *Proc. Assoc. Comput. Machinery Special Interest Group Programm. Languages (SIGPLAN) Conf. Programm. Language Design Implement.*, Jun. 2000. pp. 83–94.
[43] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. "PVS: Combining specification, proof checking, and model checking," in *Proc. Comput.-Aided Verificat. (CAV)*, LNCS 1102. Jul.–Aug. 1996, pp. 411–414.
[44] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, LNCS 828. Berlin, Germany: Springer-Verlag, 1994.
[45] D. Peled, "Ten years of partial order reduction," in *Proc. Int. Conf. Comput.-Aided Verificat.*, Jun. 1998, pp. 17–28.
[46] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Proc. 4th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, LNCS 1384. 1998, pp. 151–166.
[47] M. Rinard and D. Marinov, "Credible compilation," in *Proc. FLoC Workshop Run-Time Result Verificat.*, Jul. 1999.
[48] A. Robinson and A. Voronkov, Eds., *Handbook of Automated Reasoning*. Amsterdam, The Netherlands: Elsevier, 2001.
[49] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood, "Hierarchical compression for model-checking CSP or how to check 1020 dining philosophers for deadlock," in *Proc. 1st Int. Workshop Tools Algorithms Construct. Anal. Syst.*, 1995, pp. 133–152.
[50] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe [formal system design]," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 1, pp. 17–32, Jan. 2004.
[51] H. Tej and B. Wolff, "A corrected failure divergence model for CSP in Isabelle/HOL," in *Proc. 4th Int. Symp. Formal Methods Eur. Ind. Applicat. Strengthened Foundations Formal Methods*, 1997, pp. 318–337.
[52] R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*. Boston, MA: Kluwer, 1991.
[53] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. "VOC: A methodology for the translation validation of optimizing compilers," *J. Univ. Comput. Sci.*, vol. 9, no. 3, pp. 223–247, Mar. 2003.
[54] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. "Translation and run-time validation of loop transformations," *Formal Methods Syst. Design*, vol. 27, no. 3, pp. 335–360, 2005.

Sudipta Kundu received the B.S. and M.S. degrees in mathematics and computing from the Indian Institute of Technology (IIT) Kharagpur, Kharagpur, India, in 2002 and 2004, respectively, and the Ph.D. degree in computer engineering from the University of California (UC) San Diego, La Jolla, in 2009.

He is currently a Senior Research and Development Engineer with Synopsys, Inc., Hillsboro, OR. He worked on this project as a graduate student at UC. His current research interests include high-level verification, equivalence checking, automatic verification of system-level designs, and compiler verification. Earlier he worked on high-level synthesis, embedded operating systems, and heterogeneous home networks.

Dr. Kundu received the prestigious silver medal for being ranked first in the department at the IIT Kharagpur.

Sorin Lerner received the B.Eng. degree in computer engineering from McGill University, Montreal, Canada, the M.S. degree from the University of Washington, Seattle, and the Ph.D. degree in computer science from the University of Washington, in 1999, 2001, and 2006, respectively.

While in graduate school, he interned several times at Microsoft Research. He is currently an Assistant Professor with the Department of Computer Science and Engineering, UC San Diego, San Diego. His current research interests include programming languages and program analysis techniques for making software systems easier to write, maintain and understand, including static program analysis, domain specific languages, compilation, formal methods, and automated theorem proving.

Rajesh K. Gupta (M'83–F'04) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology Kanpur, Kanpur, India, in 1984, the M.S. degree in electrical engineering and computer science from the UC Berkeley, Berkeley, in 1986, and the Ph.D. degree in electrical engineering from Stanford University, Palo Alto, CA, in 1994.

He was previously with Computer Science Faculty, University of Illinois at Urbana-Champaign (UIUC), Champaign, and UC Irvine, Irvine. Prior to UIUC, he was with Intel Corporation, Santa Clara, CA, where he worked as a member of design teams for three generations of microprocessor devices with design experience from Bi/CMOS to high-speed GaAs devices. He is currently a QUALCOMM Chair Professor with the Department of Computer Science and Engineering, UC San Diego. His current research interests include the energy and thermally efficient large-scale systems and distributed processing in sensor networks. His recent contributions include SystemC modeling and SPARK parallelizing high-level synthesis, both of which are publicly available and have been incorporated into industrial practice.

Dr. Gupta lead the Defense Advanced Research Projects Agency-supported Adaptive Memory Reconfiguration Management project which demonstrated methods to optimize movement and placement of application data across the memory hierarchy. His ongoing efforts include energy-efficient datacenters (National Science Foundation (NSF)-supported Project GreenLight) and large scale computing using memory-coherent algorithmic accelerators and nonvolatile storage systems (Department of Defense-supported Project NV-DISC). In recent years, he and his students received the Best Paper Award at the 2008 IEEE/Association for Computing Machinery (ACM) Distributed Computing in Sensor Systems, and the Best Demonstration Award at the 2005 IEEE/ACM Information Processing in Sensor Network Platforms and Tools (SPOTS). He is a recipient of the NSF CAREER Award and the Achievement Awards at Intel Corporation. He currently serves as the Editor-in-Chief of the IEEE EMBEDDED SYSTEMS LETTERS.