Removing Concurrency for Rapid Functional Verification

Stephen Longfield, Jr., Rajit Manohar
School of Electrical and Computer Engineering
Cornell University, Ithaca, NY, 14853, U.S.A.
{slongfield, rajit}@csl.cornell.edu

***Abstract*—VLSI systems are commonly specified using sequential executable functional specifications, but implemented in a highly concurrent manner. Alhough the methods to transform between the sequential specification and concurrent implementation have been well-studied, there are still substantial difficulties in verifying that the concurrent implementation corresponds to the sequential specification after low-level optimization. The majority of methods for doing this verification have focused on strong semantic models for reasoning about systems and their specifications, but these models can add significant unnecessary complexity. In this paper, we explore a weak but effective method for reasoning about implementation relations. We show how a sequential embedding of a concurrent program can be generated, and how that embedding can be used to dramatically reduce the reachable state space of the verification problem while maintaining the semantic model of interest.**

I. Introduction

Verifying that VLSI circuits operate as specified is an important area of research due to the high cost of failure in chip design and the complexity of these verification problems. A recent study determined that, as of 2012, more time was spent in the verification of VLSI systems than in their design, and that the complexity of these problems was growing at a double-exponential rate with an increase in the number of devices on chip [1].

Much of the existing work has focused on improving the scalability of state-based methods, or on ease of application of theorem proving based methods. In this paper, we focus on a semantic model of concurrent systems, *behaviors*, and show how properties of this model can be leveraged to dramatically reduce the size of the functional verification problem. In particular, we show how to generate a sequential embedding of the concurrent implementation which has the same behavior set.

We will be focusing on verifying the *projection* design style [2] in which a sequential specification is tranformed into several smaller, concurrent programs, operating on disjoint parts of the original task. This allows the designer to introduce pipelining and other forms of concurrency while maintaining an exact correspondence to the original specification. This style was originally introduced for reasoning about high-performance VLSI systems, but here we show how properties of the behaviors produced by projection can simplify verification.

Projection has found a significant foothold in the self-timed VLSI community, where it has been used to design many systems, from high performance MIPS microprocessors [3] and FPGAs [4] to portions of a low power GPS baseband processor [5], while providing the theory for the CAD tool Proteus [6], the static token synthesis technique [7], and dataflow synthesis for FPGAs [8].

One limitation of projection is that if optimizations are made after the specification has been transformed into the concurrent representation, the method to augment the original specification in a way that permits these optimizations may be non-obvious. Typically checking that these optimizations do not violate the specification requires expensive state-space verification. Using the method we present, this check consists of comparing the externally observable behavior of two finite-state sequential programs, a much simpler problem. We have created a software implementation of our method, and have found it to be applicable to real-world systems. In addition, we created a method for verifying that the sequential embedding is a valid representation of the original program. The embedding verification algorithm is substantially simpler than the embedding algorithm, meaning the trusted code base can be small.

II. Related Work

A. Concurrent semantic models

When verifying the functionality of concurrent systems, it is important to carefully consider the semantic model used for reasoning about that system. A model which is too strong may classify interchangeable systems as distinct, while a model that is too weak may not provide the mechanisms needed to reason about certain properties. For instance, semantic models that do not distinguish between deadlocking and and successful completion make it impossible to reason about these separately.

A classic model for reasoning about concurrent systems is to model the system and its specification as labeled transition systems, and to check if these models can simulate each other, a method known as bisimulation [9]. This method is very strong, and will distinguish between systems which may have the same observable behavior, but different internal causality. It can be weakened by hiding “internal” transitions; however, it is still a very strong relation, and may distinguish between two otherwise interchangeable systems.

Trace theory is a reasoning method that has been found to avoid some of the problems of bisimulation, particularly when applied to VLSI systems [10]. In this semantic model, the system is modeled as a set of sequences of atomic actions, where each element of the set represents a possible execution of the system. This allows for dramatically different implementations to be compared. However, it commonly suffers from the *state explosion problem*, where increasing the number of processes exponentially increases the size of the trace set, making it difficult to apply this method to large systems.

A possible weakening of trace theory is to use partially ordered events [11]. Using this method, only events which can affect each other are given ordering relations, and no representation is built for the sequencing behavior of independent events. This restricts any analyses that use this semantic model to only reason about actual dependencies, which may dramatically improve the runtime of the analysis. The main downside of this method is that they often require significant processing to identify independent events.

B. Verification of concurrent systems

One of the most successful methods for verifying finite state concurrent systems has been model checking [12]. In this method, safety and liveness properties are encoded in a logic language (typically computation tree logic or linear temporal logic), and checked to see if they are satisfied in the state space of the system, most commonly represented as a Boolean Decision Diagram (BDD). The chief weakness of this method is that the state explosion problem can cause the BDD to be intractable. Recent work in model checking has looked to solve some of these issues with SAT- and SMT-based symbolic bounded model checking [13]. However, these methods still may have difficulty with highly concurrent systems, where the transition relation becomes large [14].

The other successful efforts tend to fall into the class of theorem proving methods. In these methods, a set of invariants is built up, and properties about these are proven to hold within the execution of the system [15]. Theorem proving methods can apply to infinite programs and data structures, but require manual expert work to construct the formulas, which can take up to months of effort.

C. Concurrency reduction

We take inspiration from previous work which has found that reducing the amount of concurrency in systems helps to simplify reasoning about their properties. This has found a moderate amount of success in the concurrent programming community, in particular with a focus on making concurrent programs simpler to debug and in reducing timing side channels [16].

Concurrency reduction also has a classic inspiration from dataflow languages. In static dataflow, if there is a consistent schedule for the graph, it is possible to schedule the execution of that graph statically in finite memory. This is also possible for a subset of dynamic dataflow graphs, and algorithms exist for finding these schedules based on clustering subgraphs [17]. These methods are mostly based on finding consistent probabilities that describe how the system evolves, which if found, give strong guarantees about the memory required to execute the dataflow graph.

III. Background

A. CHP language

In this paper, we use the Communicating Hardware Processes (CHP) language, a derivative of Hoare's Communicating Sequential Processses (CSP) [18], to represent both the high-level sequential and low-level concurrent specifications of systems. This language is Turing-complete [19], which allows us to simplify our reasoning and presentation without sacrificing our ability to generalize this method to more complex specication languages, such as VerilogCSP [20]. Here we informally present the CHP language. A formal trace-tree based semantics can be found in [21]. Unless otherwise specified, variables are represented by lowercase letters, communication channels are represented by uppercase letters from the beginning of the alphabet, and process or program fragments are represented by uppercase letters from the second half of the alphabet.

Each CHP process is built out of smaller processes, built up according to the following constructions:

- **Assignment**: $x := E$. This statement means "assign the value of expression E to x."
- **Communication**: $A!E$ is a statement meaning "send the value of expression E over channel A," and $B?x$ means "receive a value over channel B and store it in variable x." Both send and receive are blocking, enabling them to be used as both synchronization and data-communication primitives.
- **Choice**: $[G_1 \rightarrow P_1 [\!] \ldots [\!] G_n \rightarrow P_n]$, where each G_i is a Boolean expression (guard), and each P_i is a program fragment. This statement is executed by waiting for exactly one of the guards to be true, and then executing the associated fragment. If the guards are not mutually exclusive, a thin bar ($|$) is used instead of the thick bar ($[\!]$) to indicate a non-deterministic choice.
- **Probe**: The Boolean $\overline{A}$ is true if and only if a communication on channel A can complete without suspending [22]. Probes are only allowed to occur in the guards of choice statements.
- **Repetition**: $*[P]$ infinitely repeats statement P.
- **Sequential Composition**: $P; Q$.
- **Parallel Composition**: $P \parallel Q$.

Several CHP processes can be transformed into a full CHP program by composing them together in parallel. The execution of this program is assumed to be *weakly fair*, meaning that every continuously enabled action will eventually be given a chance to execute.

B. Behaviors

Many methods for reasoning about equivalence of concurrent systems are based in some way on the concept of traces—sequences of atomic actions [10]. For reasoning about concurrent message-passing systems, these traces typically overspecify, particularly in the way that they represent reachable interleavings.

For example, consider the following CHP programs:

$$\begin{aligned} P0 &\triangleq *[A_0!\textbf{true}; B!\textbf{false}] \parallel *[A_0?x; A_1!x] \parallel *[A_1?a] \parallel *[B?b] \\ P1 &\triangleq *[A_0!\textbf{true}] \parallel *[B!\textbf{false}] \parallel *[A_0?x; A_1!x] \parallel *[A_1?a; B?b] \end{aligned}$$

The program $P0$ is made of four concurrent sub-programs, or *processes*, one which alternates between sending **true** on channel A_0 and sending **false** on B; another which reads a value in from A_0 and sends that value out on A_1; and two processes which read from channels, A_1 and B, and write into variables, a and b. The program $P1$ is similar, however, instead of sequencing sends on A_0 and B, it sequences receives on A_1 and B.

In both of the programs, an infinite number of communications will take place, and the channels A_0 and A_1 will be continuously communicating **true** tokens, and channel B will be communicating **false** tokens. Additionally, in both programs, the difference in the count of the tokens passing through channels A_0, A_1, and B will be bounded. In many of the metrics that designers are concerned with, these programs act identically. However, trace-theoretically, they are distinct due to the reachable interleavings of actions. In $P1$, the nth communication on B cannot occur until after the nth communication on A_1, but in $P0$, it may occur before.

To reason about the equality properties that refer only to data values computed, it is possible to use *behaviors* [23]. The behavior of a trace is represented by a list of communication action sequences for each channel and a list of *decision points* (the outcome of a non-deterministic choice). As our examples do not contain any non-deterministic choice statements, the behaviors of programs $P0$ and $P1$ will just be sequences of channel actions. In this particular case, there is only one behavior for each program, and the two programs' behavior sets are unary and equal:

$$\mathcal{B}(P0) = \mathcal{B}(P1) = \begin{cases} A_0 & \mapsto \quad \textbf{true}, \textbf{true}, \textbf{true}, \ldots \\ A_1 & \mapsto \quad \textbf{true}, \textbf{true}, \textbf{true}, \ldots \\ B & \mapsto \quad \textbf{false}, \textbf{false}, \textbf{false}, \ldots \end{cases}$$

This method for reasoning about systems is relatively weak by design, which is advantageous when reasoning about program equivalence with respect to an interface. However, there are certain properties which cannot be expressed using behaviors, most notably mutual exclusion. For example, consider the following two processes, each of which contains a non-critical section (which can be interleaved in any way) and a critical section (which must be mutually exclusive):

$$\begin{gathered} *[NCS_1; CS_1] \\ \parallel *[NCS_2; CS_2] \end{gathered}$$

Behaviors cannot be used to observe the property that the two processes access their critical sections, CS_i, in an exclusive manner since they can only observe the sequence of values on channels.

C. *Slack Elasticity*

The examples $P0$ and $P1$ in the previous section are part of a special class of CHP programs known as *slack elastic* programs [23]. This class of programs has the useful property that there will only be a single behavior in the behavior set. These programs get their name from the fact that slack (i.e. buffering) can be added to any channel without altering the behavior. This property has been found to hold for a wide variety of specifications, and to be very useful when building complex designs, as it can allow additional performance-enhancing buffers to be added late in the design without altering the correctness [24].

Conveniently, all CHP programs that contain neither probes, which would allow processes to reason about interleavings, nor any shared resources, which would require reasoning about mutual exclusion, are slack elastic [2]. This check is purely syntactic, meaning it can be established very easily. This is a sufficient but not necessary requirement for slack elasticity, as it is possible for programs to contain probes and shared resources, but to still only have a single behavior.

D. *Projection*

If a system is known to be slack elastic, it is possible to project programs onto disjoint sets of variables and channels to make simpler programs without affecting the behavior set. By removing unneeded synchronization, this process can increase the concurrency in the system and improve performance. This technique is known as *projection* [2]. For example, if we took the program

$*[A?x; L?y; B!x; R!y]$

and projected it onto the disjoint sets: $\{A?, B!, x\}$ and $\{L?, R!, y\}$, we get two processes. These processes can be composed in parallel to form a program that has the same behavior set as the original:

$*[A?x; B!x] \,\|\, *[L?y; R!y]$

This may seem counterintuitive at first—the projected program does not have the same control flow as the original program, and it is possible for the two processes that make up the program to become vastly desynchronized. For instance, in the original program, the 3rd communication on L always follows the 3rd communication on A. In the projected program, there may be thousands of communications on L before the first communication on A.

However, when correctness is separated from the interleaving behavior of a system, and only the data dependencies between channels are considered (i.e. A feeding to B via x, and L to R via y), it can easily be seen that these programs are equivalent. In other words, projection preserves behaviors.

A common use of projection is for pipelining a specification. For instance, the program:

$$*[L?x; R!g(f(x))]$$

is equivalent to:

$$*[L?x; y := f(x); R!g(y)]$$

which can be transformed by replacing the assignment with channel actions using the communication axiom [18]:

$$*[L?x; (R'!f(x) \,\|\, R'?y); R!g(y)]$$

and then projected onto the sets $\{L?, R'!, x\}$ and $\{R'?, R!, y\}$:

$$*[L?x; R'!f(x)] \,\|\, *[R'?y; R!g(y)]$$

and thereby introducing pipelining. This does not perfectly preserve the behavior, as it has added a new channel R', which was not in the original behavior. However, the *externally-facing* channels L and R, which the environment uses to interface with this program, will have the same behavior. Because the system is not *closed*, i.e., because there are externally-facing channels, we cannot give a closed-form description of the behavior set. Instead, we describe the behavior of the output channel, R, as a function of the input channel, L and the iteration count, i:

$$\mathcal{B}(Original) = \mathcal{B}(Projected) = \forall i.R[i] \mapsto g(f(L[i]))$$

E. *Deadlock prevention*

The method we present requires an assumption of deadlock freedom, which may be difficult to guarantee in some cases. However, there are many existing static analysis algorithms for detecting if deadlocks can occur. In particular, we are interested in algorithms for ensuring deadlock freedom of finite-state distributed systems. These algorithms are commonly referred to as deadlock prevention algorithms, and many methods are known, several of which extend existing static analysis techniques [25,26]. It has also been found that these techniques can be more efficient when opportunities to abstract away data can be identified, for instance in systems which can be represented as Petri nets [27].

IV. Deprojection

As a slack elastic program only has a single behavior, all traces reachable by that program will map onto the same behavior. This means that a single trace through the program can act as a "witness" for the behavior set. In essence, for slack elastic programs, the existence of a correct execution means that all executions are correct, for correctness properties which can be expressed on behaviors, such as functional equality with respect to an interface.

In this section, we present an algorithm for *deprojection*, which takes a known deadlock free, concurrent, slack elastic program and generates a sequential program which will have the same behavior set on externally-facing channels. This is similar to finding a total ordering on actions that is consistent with the partial ordering implied by the original processes and the synchronization behavior of the channel communications. The deprojected program can then be used in analyses which would have previously run on the unmodified program. For many slack elastic programs and their properties of interest, this effectively solves the state explosion problem.

The algorithm is built upon a symbolic execution of the program, which we refer to as the *breadcrumb* algorithm for the CHP language, to draw analogy between leaving a trail of breadcrumbs while navigating a maze, and generating a deprojected program through symbolic execution. This symbolic execution is looking to find a path through the program which executes all possible statements, analogous to building a complete map of a maze.

The algorithm keeps track of the position in the program using a vector of program counters (PC), representing the state of each process in the program. At every step, the algorithm attempts to explore a new part of the program, until it has found a loop back to a previously explored position, such that the path passes through every reachable instruction. In the case of branching execution, i.e., choice, the deprojection algorithm speculatively explores each branch. If it finds a deadlock—analogous to a dead end in a maze—it backtracks, and tries to take the other option. This use of a deadlock guarantee lets the symbolic execution proceed without maintaining any of the state information which would be needed to evaluate the conditions of the choice.

```
 1: procedure DEPROJECT($program, pc\_vec\_seen, choice$):
 2:   $breadcrumbs \leftarrow [\,]$
 3:   $pc\_vec \leftarrow$ get_pc_vec($program$)
 4:   repeat
 5:     $instr \leftarrow$ get_next_instr($program$)
 6:     if is_assign($instr$) then
 7:       $breadcrumbs$.append($instr$)
 8:     else if is_external_comm($instr$) then
 9:       $breadcrumbs$.append($instr$)
10:     else if is_internal_comm(instr) then
11:       $instr2 \leftarrow$ get_comm($program, instr$)
12:       if $instr2$ = Null then
13:         replace_instr($instr$)
14:         continue
15:       else
16:         $b \leftarrow$ make_assign($instr, instr2$)
17:         $breadcrumbs$.append($b$)
18:       end if
19:     else if is_choice(instr) then
20:       if $instr \in choice$ then
21:         return Fail
22:       end if
23:       $(p_0, p_1) \leftarrow$ get_choices($instr$)
24:       $c' \leftarrow choice \cup \{instr\}$
25:       $pvs \leftarrow pc\_vec\_seen$
26:       $(b_0, pcv_0) \leftarrow$ deproject($program[p \mapsto p_0], pvs, c'$)
27:       $(b_1, pcv_1) \leftarrow$ deproject($program[p \mapsto p_1], pvs, c'$)
28:       if $b_0 =$ Deadlock $\wedge\, b_1 =$ Deadlock then
29:         return Deadlock
30:       else if $b_0 =$ Deadlock then
31:         $breadcrumbs$.append($b_1$)
32:       else if $b_1 =$ Deadlock then
33:         $breadcrumbs$.append($b_0$)
34:       else
35:         $(b'_0, b'_1) \leftarrow$ align($b_0, b_1, pcv_0, pcv_1$)
36:         $b \leftarrow$ make_choice($instr, b'_0, b'_1$))
37:         $breadcrumbs$.append($b$)
38:       end if
39:     else if $instr$ = Null then
40:       return Deadlock
41:     end if
42:     $pc\_vec\_seen \leftarrow pc\_vec\_seen \cup \{pc\_vec\}$
43:     $pc\_vec \leftarrow$ get_pc_vec($program$)
44:   until $pc\_vec \in pc\_vec\_seen \wedge$ has_successor($pc\_vec$)
45:   return $breadcrumbs, pc\_vec$
46: end procedure
```

Fig. 1. Psuedocode for deprojection algorithm

The most significant limitation of the method we present is that we do not allow for systems where the same choice needs to be taken twice to reach a previously seen position. Systems built by projection from a sequential specification will fit within this limitation, but there are others that may not.

A. Algorithm

The pseudocode for the deprojecton algorithm is shown in Figure 1. The rest of this subsection gives an informal overview of the algorithm. This description references line numbers and function names from Figure 1 when relevant. The next two subsections give two example applications of this algorithm, again referencing these line numbers and function names.

This algorithm is defined recursively, and should be initialized with the full program as the argument to $program$, and empty sets for pc_vec_seen and $choice$.

The algorithm first saves the current state of the program by getting a vector of program counters, one for each of the parallel-executing processes in the program (**get_pc_vec** on line 3), then enters a repeat-until loop. The first step of this loop is to choose an instruction to execute, based on the current state of the speculative execution, using the function **get_next_instr** (line 5). In concurrent systems, there will likely be many available next instructions; if possible, **get_next_instr** will select one that has not been executed previously in the symbolic execution. If this is not possible, it will pick the least recently-explored instruction available.

If the selected instruction is an assignment or an external communication, it immediately drops a breadcrumb (lines 6-9). If it is an internal communication, and the other side of the communication is ready to execute, the assignment breadcrumb generated from this communication is dropped (lines 10-18), where the function **get_comm** searches the program for the ready-to-execute complementary side of an internal communication. If no action is available, (i.e., every process has reached a halt condition or some process is deadlocked) the algorithm returns a deadlock (lines 39-40).

In the case of a choice, both paths are speculatively executed by running a deprojection with the choice path substituted for the current process (lines 23-27). As we are limiting ourselves to programs whose environment guarantees that the control values are deadlock free, if one branch of the choice results in a deadlock, we assume that the other branch is the only one which can execute, and so only place the breadcrumbs from that branch (lines 28-33). If both paths have a deadlock free execution, the algorithm post-processes them to align the breadcrumb executions, and then creates a choice, which it adds to the current list of breadcrumbs (lines 35-37).

Deadlock freedom must be guaranteed by another analysis, outside of the scope of this paper. During deprojection, we do not keep track of the data values that are used to evaluate selection, and so a speculative execution of a choice may lead to an erroneous deadlock. This deadlock condition must be detected during the speculative execution. We do this by keeping track of which channels have been made unavailable by speculative execution, and then detect deadlock when an unavailable channel action is required for forward progress, using classic resource-tracking deadlock detection methods [25]. For the sake of conciseness, resource management and deadlock detection are not shown in Figure 1.

After processing the current instruction, the state, represented by a program counter vector pc_vec, is added onto the list of previously seen states and a new pc_vec is generated (lines 42 and 43).

This process repeats until the speculative execution reaches a state it has seen before, where each one of the processes in that program counter vector has an immediate successor, which may wrap around in the case of loops (i.e. the first state of the program is the successor of the last state, if there is a loop). This is accomplished by a check for **has_successor** in the exit condition of the loop on line 44. This check is necessary to guarantee *weak fairness*: that the speculative execution will not halt in a state where an instruction is enabled, but has never been explored.

B. Simple example

As an example, consider the simple program:
$*[A?a;\, B?b;\, C!(a \wedge b)] \parallel *[C?c;\, D!\neg c]$
This program has three *external* channels, A, B, and D, as well as an *internal* channel, C. The deprojection of the program is shown in

	Program state	Breadcrumbs	pc_vec_seen	Loop iteration
	$*[_0\mathbf{A?a};_1 B?b;_2 C!(a \wedge b)]$ $\| *[_0\mathbf{C?c};_1 D!\neg c]$		$\{\,\}$	0
▷	$*[_0 A?a;_1\mathbf{B?b};_2 C!(a \wedge b)]$ $\| *[_0\mathbf{C?c};_1 D!\neg c]$	$A?a$	$\{\langle 0,0\rangle\}$	1
▷	$*[_0 A?a;_1 B?b;_2\mathbf{C!(a \wedge b)}]$ $\| *[_0\mathbf{C?c};_1 D!\neg c]$	$A?a;\, B?b$	$\{\langle 0,0\rangle; \langle 1,0\rangle\}$	2
▷	$*[_0\mathbf{A?a};_1 B?b;_2 C!(a \wedge b)]$ $\| *[_0 C?c;\, _1\mathbf{D!\neg c}]$	$A?a;\, B?b;\, c := a \wedge b$	$\{\langle 0,0\rangle; \langle 1,0\rangle; \langle 2,0\rangle\}$	3
▷	$*[_0\mathbf{A?a};_1 B?b;_2 C!(a \wedge b)]$ $\| *[_0\mathbf{C?c};_1 D!\neg c]$	$A?a;\, B?b;\, c := a \wedge b;$ $D!\neg c$	$\{\langle 0,0\rangle; \langle 1,0\rangle; \langle 2,0\rangle; \langle 0,1\rangle\}$	4
▷	$*[_0\mathbf{A?a};_1 B?b;_2 C!(a \wedge b)]$ $\| *[_0\mathbf{C?c};_1 D!\neg c]$	$*[A?a;\, B?b;\, c := a \wedge b;$ $D!\neg c]$	$\{\langle 0,0\rangle; \langle 1,0\rangle; \langle 2,0\rangle; \langle 0,1\rangle\}$	

Fig. 2. Simple deprojection

Figure 2. The current value of pc_vec is represented by bolding and coloring dark blue the instructions which may execute, and the PC of each instruction is given as a prepended subscript. The behavior set for the program can be described by:

$$\mathcal{B} = \begin{cases} \forall i.C[i] & \mapsto \quad A[i] \wedge B[i] \\ \forall j.D[j] & \mapsto \quad \neg C[j] \end{cases}$$

In the first loop iteration, two instructions are ready to execute: the external communication on A, and the internal receive on C. These two instructions are both labeled with PC 0, and so the PC vector associated with this state is $\langle 0,0\rangle$. The function **get_next_instr** may choose to execute either, however, if it picks $C?c$, the check on line 11 will not find a complementary communication. Therefore, only the execution of $A?a$ can make it into the breadcrumbs in the first iteration of the loop, and we add the PC vector $\langle 0,0\rangle$ to pc_vec_seen. Similarly, in the second iteration, only the execution of $B?b$ can create a breadcrumb and add $\langle 1,0\rangle$ to pc_vec_seen.

In the third loop iteration, the internal communication on C is ready, one side sending $a \wedge b$, and the other receiving into c. This is an internal communication, and so will not occur in the behavior set when limited to only the external communications, however for the deprojected program to represent the same semantics as the original, the deprojection must drop a breadcrumb with equivalent semantics. In this case, we can use the synchronous channel definition of assignment to drop the assignment $c := a \wedge b$ [18]. This is constructed by the **make_assign** function, in line 16 of the algorithm. Leaving this state adds the PC vector $\langle 2,0\rangle$ to pc_vec_seen.

In the fourth loop iteration, again two instructions are ready to execute: an external communication on A, and an external communication on D. If we execute A a second time, the deprojection will contain two instances of the $A?a$, though the first will be outside of the repetition. As this is less compact, we execute D, dropping the $D!\neg c$ breadcrumb. This is accomplished by **get_next_instr** preferring to select not-previously-executed instructions. Leaving this state adds the PC vector $\langle 0,1\rangle$ to pc_vec_seen.

After the fourth loop iteration, the PC vector is in a state it has reached before, $\langle 0,0\rangle$. As we have also passed through the states $\langle 0,1\rangle$ and $\langle 1,0\rangle$, both processes have successors in elements of pc_vec_seen, satisfying **has_successor**, and allowing the loop to terminate. After loop termination, we have a sequence of breadcrumbs that brings us back to the starting state, and so we can insert an infinite loop surrounding the entire program, creating the deprojection:

$*[A?a;\, B?b;\, c := (a \wedge b);\, D!\neg c]$

This deprojection does not contain any communication on internal channels, having replaced the only instance of internal communication with an assignment. From the deprojection, we can derive the behavior function:

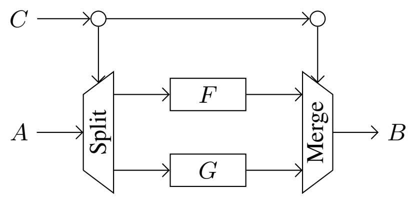

Fig. 3. Split/merge pipeline

$$\mathcal{B} = \forall i.D[i] \mapsto \neg(A[i] \wedge B[i])$$

which is similar to the original behavior description, with substitution used to remove the channel C. Note that, in this case, we were able to place the infinite loop around the full set of breadcrumbs, but in other cases, it may only enclose a subset of the breadcrumbs. The exact placement of this loop construction is determined by the pc_vec value returned on line 45.

Through brute-force state space enumeration, we can find 12 reachable states in the execution of the original program. The deprojected form of this simple program only has 5 reachable states: the state where it is receiving $A?a$, where it is receiving $B?b$, where it is assigning $c := (a \wedge b)$, where it is sending $D!\neg c$ and the state between when one iteration of the loop ends and the next begins. In total, this is a reduction by a factor of 2.4 from the original.

C. Split/merge example

A more realistic example is the split/merge pipeline shown in Figure 3. The control input for both the split and merge of this program comes from the external channel C, meaning that a static analysis cannot know which paths will be taken at runtime.

However, we are assuming that the programs are deadlock free. This means that any set of choices which results in a deadlock is not part of the reachable space. These deadlocks can be identified entirely without knowledge of the concrete data of the program by analysis of resource requirements and restrictions. When these deadlocks are found, we can backtrack to the most recently executed choice, and try the other option. If both options are deadlocking, we backtrack to the previous choice. In the pseudocode in Figure 1, this is accomplished with the recursion on lines 26 and 27, and then the deadlock checks on lines 28 through 33.

If the pipeline in Figure 3 is made out of processes as follows:

C Copy $\triangleq *[C?c;\, C_0!c, C_1!c]$
C Buf $\triangleq *[C_1?c_0;\, C_2!c_0]$
Split $\triangleq *[C_0?c_1, A?x;\, [c_1 \rightarrow L_0!x \,[\!]\, \neg c_1 \rightarrow R_0!x]]$
F $\triangleq *[L_0?x_0;\, L_1!f(x_0)]$
G $\triangleq *[R_0?x_1;\, R_1!g(x_1)]$
Merge $\triangleq *[C_2?c_2;\, [c_2 \rightarrow L_1?x_2 \,[\!]\, \neg c_2 \rightarrow R_1?x_2];\, B!x_2]$

it can be deprojected into the program:

$*[C?c;\, c_0 := c;\, A?x;\, c_1 := c;\, c_2 := c_0$
$\quad [c_1 \rightarrow x_0 := x;\, x_2 := f(x_0);\, B!x_2$
$\quad [\!]\, \neg c_1 \rightarrow x_1 := x;\, x_2 := g(x_1);\, B!x_2]]$

This deprojection will encounter a deadlock state when it speculatively executes the "top" of the split and the "bottom" of the merge simultaneously. In this case, the F pipeline stage will be attempting to send on the channel L_1, however, the speculation of the merge will have hidden the receive on this channel. This combination of a requirement for a resource, and a restriction on a resource leads to a deadlock, and so the speculative execution on the top of the split and the bottom of the merge will not be included in the breadcrumbs.

Note that for speculative execution of choice, both choice clauses have to reach the same PC vector before rejoining. For example, if the speculative execution where c_1 is true re-executes the external communication on C, it will produce to the breadcrumbs:

$x_0 := x;\, x_2 := f(x_0);\, B!x_2;\, C?c$

If the speculative execution where c_1 is false proceeds as above, it produces the breadcrumbs:

$x_1 := x;\, x_2 := g(x_1);\, B!x_2$

These need to be realigned before they can be merged into a single selection statement. This is done using the **align** function (line 36 of Figure 1). This function will typically append instructions from the set of breadcrumbs that reached further in the execution onto the one that did not, until the two are in the same state. In this case, that means adding the $C?c$ breadcrumb to the false state, leading to the final deprojection:

$C?c;\, *[c_0 := c;\, A?x;\, c_1 := c;\, c_2 := c_0$
$\quad [c_1 \rightarrow x_0 := x;\, x_2 := f(x_0);\, B!x_2;\, C?c$
$\quad [\!]\, \neg c_1 \rightarrow x_1 := x;\, x_2 := g(x_1);\, B!x_2;\, C?c]]$

We can clean up the original deprojection using copy propagation and dead variable removal [28], common compiler optimizations which preserve behaviors on externally facing channels:

$*[C?c;\, A?x;\, [c \rightarrow B!f(x) \,[\!]\, \neg c \rightarrow B!g(x)]]$

In the original program, there were over 1,200 reachable states. After deprojection, there are only 13 reachable states, and after optimization, only 6. In total, this method reduces the reachable state space of the simple split/merge by a factor of roughly two hundred.

In this case, the selection is *properly nested*, meaning that every n-way split is joined by an n-way merge. This relatively common scenario can be very efficient when augmented with a heuristic: only speculatively execute merges if an input is ready. With this heuristic, this deprojection does not spend time in bad speculative executions, and proceeds in time linear with respect to the total number of instructions.

D. Incompleteness

There are cases where speculative execution of selection statements may diverge, for instance:

$*[C?c;\, [c \rightarrow A! \,[\!]\, c \rightarrow B!]]$
$\quad \| *[A?;\, D!;\, A?E!;] \,\|\, *[B?;\, F!;\, B?;\, G!]$

has a choice based on the externally-sourced value from C, which will select between two processes: one of which alternates between sending on external channels D and E, and the other on F and G. That is to say, after the result of a single choice, there is no path back to the starting PC vector.

If the speculative execution were to put down a second choice breadcrumb, split the speculative execution, and continue the search for a previously seen state, it will diverge, as this program does not have a well-formed deprojection. To avoid these divergences, we limit undetermined choice to only execute a single time in the deprojection algorithm. This is accomplished by adding it to the *choice* set whenever we speculative execute a choice (line 24), and then checking that set before further speculative execution (lines 20 and 21).

This is a reasonable solution, because duplicated undetermined choice is rare in real world systems. Particularly, as these forms of choice cannot exist in a single-threaded program, they will not be introduced by projection. The issue highlighted by this example is that our algorithm can fail if the correct operation of the program requires an additional constraint on the data values supplied by the environment beyond those needed for deadlock freedom.

E. Correctness

For correctness, we will first show that the algorithm satisfies two important properties, *halting*, and *weak fairness*, and then we will describe an algorithm for *reprojection*, which allows us to certify that the deprojected program has a behavior set identical to the original.

1) Halting: The deprojection algorithm will always halt.

It will halt under two conditions:

(a) If it reaches a point where no further progress is available
(b) If a deprojection is discovered

The first case will happen if there are no instructions which can execute, or if we require nested speculative execution of a choice.

In the case that this does not happen, the program will halt if it enters a state it has seen before where all of the processes in the program have successors.

First, note that pc_vec_seen has a finite upper bound on its size. pc_vec is a vector with length equal to the number of processes in the system. Each one of the elements in this vector has a finite number of possible values, equal to the number of possible states in this process. Therefore, the maximum number of elements that can be in the pc_vec_seen set is the product of the number of states in each of the processes. Also note that, when this list is full, every value in the set either satisfies the **has_successor** function or is in a state where no instructions can execute. Therefore, if the program enters the state where the pc_vec_seen set is full, it will halt and produce a deprojection.

Now we show that if we do not halt otherwise, the pc_vec_seen set will monotonically grow towards this full set.

Line 42 is the only line which modifies pc_vec_seen, adding the value of pc_vec with a set union. If pc_vec was not in the set, this operation will grow the pc_vec_seen. We break the case where it was in the set into the two sub-cases: where pc_vec satisfies **has_successor**, and where it does not.

If it pc_vec satisfies **has_successor**, then the algorithm would have halted on the previous iteration.

If pc_vec does not satisfy **has_successor**, and it is in pc_vec_seen, then by the definitions of pc_vec and **has_successor** and the construction of pc_vec_seen, there must be at least one process which has never transitioned from its current state.

By the definition of **get_next_instr** when possible a instruction which has never transitioned will execute in the next cycle, and if it is an assignment, external communication, or choice, it will immediately affect pc_vec, and grow pc_vec_seen. If an instruction which has never transitioned is an internal communication, the complement of this may not be ready to execute; however, it still will be given the highest priority for the next loop iteration. If the algorithm does not detect resource-starvation deadlock, exactly one of the other processes will contain the complementary communication, as slack elastic CHP programs do not allow for resource sharing. When it becomes ready to execute, the value of pc_vec will be different than when it was first detected since the other parts of the program will have advanced. However, this value it will not be in pc_vec_seen, as **has_successor** failing on this process implies no value in pc_vec_seen has the successor's value in this process's position. Therefore, its execution will also grow pc_vec_seen.

2) Weak fairness: The evaluation was *weakly fair*, in that if the algorithm halts after discovering a deprojection, then every instruction that was enabled in the symbolic evaluation was evaluated.

It suffices to show that all of the enabled actions in the final state have been previously evaluated, as any other previously enabled actions must have been evaluated if they are no longer in the set of enabled actions at the final state. As the termination condition check on **has_successor** will fail if any of the enabled actions have not been evaluated, in a successful deprojection all enabled actions in the final state will have been evaluated. Therefore, the breadcrumb algorithm is weakly fair.

3) Reprojection: To ensure that the deprojected program has the exact same behavior set as the original program, we can reproject the program back onto the sets of variables that made up the original processes. If the set of processes we get is exactly equal to the original set, then the deprojection will have the same behavior set by the projection theorem [2]. This is a simple process, and so can be implemented concisely, making the trusted code base very small.

As an example, consider the deprojection of the split/merge from section IV-C:

$$\begin{aligned}&*[C?c;\, c_0 := c;\, A?x;\, c_1 := c;\, c_2 := c_0\\ &\quad [c_1 \rightarrow x_0 := x;\, x_2 := f(x_0);\, B!x_2\\ &\quad [\!] \neg c_1 \rightarrow x_1 := x;\, x_2 := g(x_1);\, B!x_2]]\end{aligned}$$

with some additional bookkeeping in the deprojection algorithm to keep track of which channels are removed when an assignment statement is created (i.e., in the **make_assign** function from Figure 1) or when deadlock removes a guard from choice, this can be projected onto the sets of channel actions and variables that originally made up the component processes:

$$\{C?, C_0!, C_1!, c\}\ \{C_1?, C_2!, c_0\}\ \{L_0?, L_1!, x_0\}\ \{R_0?, R_1!, x_1\}$$
$$\{C_0?, A?, L_0!, R_0!, c_1, x\}\qquad \{C_2?, L_1?, R_1?, B!, c_2, x_2\}$$

to get the set of processes:

$$\begin{array}{ll} *[C?c;\, C_0!c,\, C_1!c] & *[C_1?c_0;\, C_2!c_0] \\ *[L_0?x_0;\, L_1!f(x_0)] & *[R_0?x_1;\, R_1!g(x_1)] \\ \multicolumn{2}{l}{*[C_0?c_1, A?x;\, [c_1 \rightarrow L_0!x [\!] \neg c_1 \rightarrow R_0!x]]} \\ \multicolumn{2}{l}{*[C_2?c_2;\, [c_2 \rightarrow L_1?x_2 [\!]\ c_2 \rightarrow R_1?x_2];\, B!x_2]} \end{array}$$

This set exactly corresponds to the set of processes that originally described the split/merge, proving that the deprojection has the same behavior set on the channels C, A, and B.

V. Evaluation

A. Implementation

The analysis described in this document was implemented in roughly 3300 lines of OCaml. Of this, 75 are used for reprojection.

The implementation differs in some ways from the directly described algorithm, in that it first runs a type inference method, inspired by Damas-Hindley-Milner type inference in ML [29] to determine if a channel is internal or external. Additionally, while the presentation of the algorithm in Figure 1 uses a list of instructions to represent the breadcrumbs, representing them as a graph made alignment after speculative choice simpler.

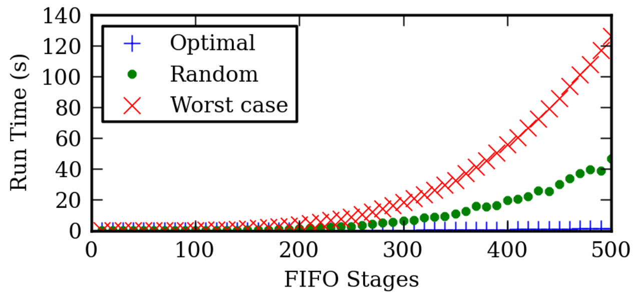

Fig. 4. Ordering sensitivity in FIFO deprojection

B. Sensitivity to ordering

There are many valid deprojections of a single program, and the deprojection generated depends on the order in which the program is explored. This does not affect the correctness of the deprojection, but it may affect the run time. In particular, as internal communication must defer until the complementary side of the communication is ready, if the processes are explored in the "wrong" order, there may be significant additions to the run time.

As an example, consider the deprojection of a multistage first-in-first-out (FIFO) buffer, made out of several single-stage FIFO elements. If the elements are explored in the order they communicate with each other, then the execution time will be roughly linear with the number of processes. However, if they are explored in inverse order, the execution time will be cubic as the failed search for the complementary communication action may take linear time, and every other process will attempt to step before the next search.

Measured run time that shows this property on a variety of sizes of FIFOs is presented in Figure 4. The deprojection was run on FIFOs ranging from 10 to 500 elements long, in increments of 10, where the deprojection explored the FIFO elements in optimal order (in-order), worst case order (inverse order), or random order. This experiment was run on an eight-core 2.80 GHz Xeon with 16 GB of RAM. Each data point is the average of running the experiment 10 times. Even in the worst case, it does not take more than a few minutes to deproject a 500 stage FIFO, which compares very favorably to the time it would take to explore the possible 2^{500} states.

C. Testing

The algorithm was run on a variety of benchmarks, both synthetic ones, as well as cases taken from real designs. These include the writeback unit of a self-timed MIPS microprocessor [2], numerically controlled oscillators (NCO) and counters from a GPS [5], the output of data- and token-flow synthesis methods [7,8] and split/merge pipelines similar to the one shown in Figure 3, but with longer pipelines. Performance metrics from this evaluation are presented in Table I. All of the tests run were reprojection equivalent to the original programs. These were collected on the same machine as in the previous subsection.

The number of states listed is only the number of different control flow states; it does not consider assignments to variables beyond how they affect this flow. In some cases (e.g., the NCOs), the control flow is significantly affected by the data, bloating the number of reachable states. In others, such as an accumulator synthesized using the static token method, they are independent and the state space is smaller.

To compare to existing partial order reductions techniques, we also present data collected from the SPIN model checker [30].

TABLE I
TEST RESULTS

	Deprojection				SPIN partial order reduction		
Test case	# states before	# states after	Ratio	Time	# states before^1	# states after	Ratio
Static token-synthesized accumulator [7]	320*	15	21	2.7 ms	2,900*	648	4.45
Pipelined merge, 4-way [8]	936*	17	55	3.5 ms	15,248*	2,764	5.51
MIPS writeback unit [2]	1,648*	31	53	12.7 ms	119,956*	48,798	2.45
Pipelined split, 4-way [8]	1,680*	17	98	2.9 ms	179,168*	8,046	22.27
FIFO, ten stage	3,840*	11	389	1.8 ms	655,872*	209,203	3.13
Pipelined counter, 8 bit [5]	4,479,000†	230	19,473	142.9 ms	$1.14 \times 10^{13\dagger}$	8,321,214	1.37×10^6
Split/merge pipeline, 2 stage	8,304*	14	593	2.7 ms	125,726	32,079	3.92
Split/merge pipeline, 5 stage	$9.85 \times 10^{7\dagger}$	24	4.1×10^6	4.5 ms	$8.67{\times}10^{8\dagger}$	3.57×10^8	24.29
Split/merge pipeline, 10 stage	$8.4 \times 10^{11\dagger}$	36	2.3×10^{10}	7.7 ms	$5.44{\times}10^{14\dagger}$	●	●
Pipelined NCO, 8 bit [5]	$1.27 \times 10^{10\dagger}$	48	2.64×10^8	28.8 ms	$3.47 \times 10^{11\dagger}$	647,911	535,567
Pipelined NCO, 16 bit [5]	$5.45 \times 10^{18\dagger}$	98	5.56×10^{16}	410.8 ms	$1.56 \times 10^{21\dagger}$	6.84×10^7	2.27×10^{13}
Pipelined NCO, 32 bit [5]	$1.08 \times 10^{36\dagger}$	202	5.34×10^{33}	7,174.0 ms	$2.96 \times 10^{30\dagger}$	●	●

* From state space exploration † Estimated ^{1} SPIN uses a more complex model of communication, increasing state count ● Overran 100 GB limit

VI. CONCLUSION

In this paper, we have shown that using a weaker semantic model of concurrency—behaviors—allows for a solution to the state explosion problem by allowing a sequential embedding of the program to act as a stand in for the program for verification. We have created an algorithm for finding this sequential embedding for the language CHP, shown how the correctness of this embedding can be efficiently verified, and found it to be very effective at reducing the complexity of verification problems on real hardware specifications.

The methods presented have some limitations, in particular that we restrict undetermined choice to be only single-undetermined, and that we focus syntactically slack elastic programs. Future work will include looking into analyses for establishing slack elasticity for a wider set of programs, as well as static analyses to determine when systems have finitely-undetermined choice.

ACKNOWLEDGEMENTS

This research was supported in part by NSF grant CCF-1065307

REFERENCES

[1] H. D. Foster, "Why the design productivity gap never happened," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 581–584.

[2] R. Manohar, T. kwan Lee, and A. J. Martin, "Projection: A synthesis technique for concurrent systems," in *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999.

[3] A. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee, "The design of an asynchronous mips r3000 microprocessor," in *Advanced Research in VLSI, 1997.*, sep 1997, pp. 164 –181.

[4] D. Fang, J. Teifel, and R. Manohar, "A high-performance asynchronous fpga: Test results," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005.* IEEE, 2005, pp. 271–272.

[5] B. Tang, S. Longfield, S. Bhave, and R. Manohar, "A low power asynchronous gps baseband processor," in *Asynchronous Circuits and Systems (ASYNC), 2012*, may 2012, pp. 33 –40.

[6] P. Beerel, G. Dimou, and A. Lines, "Proteus: An asic flow for ghz asynchronous designs," *Design Test of Computers, IEEE*, vol. 28, no. 5, pp. 36–51, 2011.

[7] J. Teifel and R. Manohar, "Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *In Proceedings of International Symposium on Asynchronous Circuits and Systems*, 2004, pp. 17–27.

[8] S. Peng, D. Fang, J. Teifel, and R. Manohar, "Automated synthesis for asynchronous fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 2005, pp. 163–173.

[9] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[10] J. L. A. van de Snepscheut, *Trace Theory and VLSI Design*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1985, vol. 200.

[11] P. Wolper and P. Godefroid, "Partial-order methods for temporal verification," in *CONCUR'93*. Springer, 1993, pp. 233–246.

[12] E. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[13] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of bdds," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM, 1999, pp. 317–320.

[14] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Computer Aided Verification*. Springer, 2005, pp. 82–97.

[15] J. Y. Halpern and M. Y. Vardi, "Model checking vs. theorem proving: a manifesto," *Artificial Intelligence and Mathematical Theory of Computation. Academic Press, Inc*, vol. 212, pp. 151–176, 1991.

[16] T. Bergan, J. Devietti, N. Hunt, and L. Ceze, "The deterministic execution hammer: How well does it actually pound nails," in *Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[17] J. Buck and E. Lee, "The token flow model," in *Data Flow Workshop*, 1992.

[18] C. Hoare, "Communicating sequential processes," in *Communications of the ACM*, vol. 21, no. 8, August 1978, pp. 666–677.

[19] R. Manohar and A. J. Martin, "Quasi-delay-insensitive circuits are turing-complete," in *Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1996.

[20] A. Saifhashemi and H. Pedram, "Verilog hdl, powered by pli: a suitable framework for describing and modeling asynchronous circuits at all levels of abstraction," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 330–333.

[21] M. van der Goot, "Semantics of VLSI synthesis," Ph.D. dissertation, California Institute of Technology, May 1995.

[22] A. J. Martin, "The probe: An addition to communication primitives," *Information Processing Letters*, vol. 20, no. 3, pp. 125–130, 1985.

[23] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proceedings of the Fourth International Conference on the Mathematics of Program Construction, Lecture Notes in Computer Science 1422*. Springer-Verlag, 1998, pp. 272–285.

[24] G. Gill, V. Gupta, and M. Singh, "Performance estimation and slack matching for pipelined asynchronous architectures with choice," in *IEEE/ACM International Conference on Computer-Aided Design, 2008. ICCAD 2008.* IEEE, 2008, pp. 449–456.

[25] A. K. Elmagarmid, "A survey of distributed deadlock detection algorithms," *ACM Sigmod Record*, vol. 15, no. 3, pp. 37–45, 1986.

[26] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 161–180, 1996.

[27] Z. Li, M. Zhou, and N. Wu, "A survey and comparison of petri net-based deadlock prevention policies for flexible manufacturing systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 38, no. 2, pp. 173–188, 2008.

[28] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[29] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1982, pp. 207–212.

[30] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.