Automated Refinement Checking of Concurrent Systems

Sudipta Kundu, Sorin Lerner, and Rajesh Gupta

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

***Abstract*— Stepwise refinement is at the core of many approaches to synthesis and optimization of hardware and software systems. For instance, it can be used to build a synthesis approach for digital circuits from high level specifications. It can also be used for post-synthesis modification such as in Engineering Change Orders (ECOs). Therefore, checking if a system, modeled as a set of concurrent processes, is a refinement of another is of tremendous value. In this paper, we focus on concurrent systems modeled as Communicating Sequential Processes (CSP) and show their refinements can be validated using insights from translation validation, automated theorem proving and relational approaches to reasoning about programs. The novelty of our approach is that it handles infinite state spaces in a fully automated manner. We have implemented our refinement checking technique and have applied it to a variety of refinements. We present the details of our algorithm and experimental results. As an example, we were able to automatically check an infinite state space buffer refinement that cannot be checked by current state of the art tools such as FDR. We were also able to check the data part of an industrial case study on the EP2 system.**

I. Introduction

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 – while ensuring system-level functional correctness [5]. For instance, researchers have advocated the use of SystemC models and their stepwise refinement into interacting hardware and software components [22]. At the heart of such proposals is a model of the system consisting of concurrent pieces of functionality, oftentimes expressed as sequential program-like behavior, along with synchronous or asynchronous interactions [21], [32]. Communicating Sequential Processes [15] (CSP) is a calculus for describing such concurrent systems as a set of processes that communicate synchronously over explicitly named channels.

Refinement checking of CSP programs consists of checking that, once a refinement has been performed, the resulting refined design, expressed as a CSP program, is indeed a correct refinement of the original high-level design, also expressed as a CSP program. In the context of hardware development, refinement checking is useful because it can guarantee that the refined design correctly implements the high-level design, and it can also guarantee that certain properties, for example safety properties, are preserved from the high-level design to the refined design [33].

Earlier work on CSP refinement falls into two broad categories. First, there has been research on techniques that require human assistance. This work ranges from completely manual proof techniques, such as Josephs's work on relational approaches to CSP refinement checking [17], to semi-automated techniques where humans must provide hints to guide a theorem prover in checking refinements [12], [34], [16], [25]. Second, there has been work on fully automated techniques to CSP refinement checking [2], [9], the state of the art being embodied in the FDR tool [2]. These techniques essentially perform an exhaustive state space exploration, which places onerous restrictions on the kinds of infinite state spaces they can handle. In particular, they can only handle one kind of infinite state space, namely those arising from data-independent programs. Such programs must treat the data they process as black boxes, and the only operation they can do on data is to copy it. Although there has been work on making the finite state spaces that automated techniques can handle larger and larger [6], [10], [29], [31], there has been little work on automatically handling refinement checking of two concurrent programs whose state spaces are *truly* infinite. This is a serious limitation when attempting to raise the level of abstraction of system models to move beyond the bit-oriented data structures used in RTL specifications.

In this paper we present a new trace refinement checking algorithm for concurrent systems modeled as CSP programs. Our algorithm uses a simulation relation approach to proving refinement. In particular, we automatically establish a simulation relation that states what points in the specification program are related to what points in the implementation program. This simulation relation guarantees that for each trace in the implementation, a related and equivalent trace exists in the specification. Although there has been work on using simulation relations for automatically checking concurrent systems in a fully automated way, their use has been focused on specific low-level hardware refinements [23], [24].

Our algorithm consists of two components. The first component is given a simulation 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 simulation relation just from the specification and the implementation programs. Once the simulation relation is inferred, it can then be checked for correctness using the checking algorithm.

Unlike previous approaches, our approach can automatically check the refinement of infinite state space CSP programs that are not data-independent. This means that we can handle CSP programs that manipulate, inspect, and branch on data ranging over truly infinite domains, for example the set of all integers or all reals. In order to achieve this additional checking power, our algorithm draws insights and techniques from various areas, including translation validation [30], [26], theorem proving [11], and relational approaches to reasoning about programs [17], [20]. As a result, the contribution of our paper can be seen in different lights. One way to characterize our contribution is that we have automated a previously known, but completely manual technique, namely Josephs's relational approach to proving CSP refinements [17]. Another way to characterize our contribution is that we have incorporated an automated theorem proving component to FDR's search technique [2] in order to handle infinite state spaces that are not data-independent. Yet another way to characterize our contribution is that we have generalized translation validation [30], [26], an automated technique for checking that two sequential programs are equivalent, to account for concurrency and for trace containment checking.

To evaluate our approach, we used the Simplify [11] theorem prover to implement our CSP refinement checking algorithms in a validating system called ARCCoS (Automated Refinement Checker for Concurrent Systems). We then used our implementation to check the correctness of a variety of refinements, including parts of an industrial case study of the EP2 system [1].

1-4244-1382-6/07/$25.00 ©2007 IEEE

The remainder of the paper is organized as follows. Section II presents a simple refinement example to guide our discussion. Section III presents previous approaches, while Section IV describes an overview of our approach. Section V then provides the full details of our inference algorithm. Section VI describes experimental results, and finally Section VII presents our conclusions and plans for future work. An accompanying technical report [19] contains more details, including a full description of all our algorithms, and a more comprehensive discussion of related work.

II. Refinement Example

We start by presenting a simple refinement example that we will use to illustrate our approach. This example replaces two distinct communication links with a shared multiplexed communication link. The specification for this example is shown in Figure 1(a) using an internal CFG representation of the concurrent system. The specification states (using the || operator) that two parallel processes are continually reading values from two input channels, respectively called left1 and left2 (the instruction left1?a represents reading a value from the left1 channel and storing it into a). Each process then outputs, on either channel right1 or right2, four times the value read from the input channel (the instruction right1!w represents writing the variable w to channel right1). Reads and writes in CSP are synchronous. Figure 2(a) shows a process communication diagram for this specification, where each box represents a process, and the arrows between processes represent communication channels. In this example, the two processes link1 and link2 represent communication links, for example a wire in a network, or some path in a circuit. In refinement based hardware development, the designer often starts with such a high-level description of a communication link, refining the details of the implementation later on.

In our example, we will refine the two communication links in two ways: (1) the two links will be collapsed into a single shared communication link with multiplexing; and (2) instead of multiplying the communicated value by 4 at once, the communicated value will first be multiplied by 2 on the sender side of the link, and then multiplied again by 2 on the receiver side of the link, producing the required "times 4" effect.

The easiest way to understand the structure of the implementation is to look at its process communication diagram first, which is shown in Figure 2(b). The two communication links from the specification have been collapsed into a single link that consists of two processes (msg sender and msg receiver) that communicate over a channel called msg. At the sender end of this communication link, the values from the original left1 and left2 channels are sent to the msg sender process, which performs the multiplexing required to send all the values on a single link. At the receiver end of the communication link, the msg receiver process demultiplexes incoming values to deliver them to the appropriate channel, either right1 or right2. One additional subtlety of this example is that, in order for the refinement to be correct, an additional link needs to be added for sending acknowledgments back to the sender, so that a new value isn't read from left1 or left2 until the current value has been written out. Otherwise, the implementation will be able to buffer up to three values (one value on each of the sm1, msg and rm1 channels), whereas the specification cannot read any additional values until the current value is written out.

The CFG representation for this implementation is shown in Figure 1(b). The two processes left link1 and left link2 read values from their respective input channels, and send them to the msg sender process via the sm1 and sm2 channels (sm stands for "send message"). Before going on to the next value, left link1 and left link2 wait for an acknowledgment on the ra1 or ra2 channels (ra stands for "receive acknowledgment").

The msg sender process contains a loop with an external choice operator □ at the top of the loop. The external choice operator in CSP chooses *between* two paths based on the surrounding environment. In our case, the immediate successors of the □ node are sm1?x and sm2?y, which means the following: if only the sm1 channel has a message on it, then the sm1?x path is chosen; if only the sm2 channel has a message on it, then the sm2?x path is chosen; if both channels have messages, the choice is made non-deterministically; and if none of the channels have messages, the process waits until a message arrives on either sm1 or sm2. After a choice is made between which channel to get a value from, the msg sender process sends on the msg channel the value multiplied by 2, and then an identifier stating which of the two communication links the current value belongs to.

On the receiver side, the msg receiver process performs the required demultiplexing of values from the msg channel. In particular, msg receiver starts by reading two values from the msg channel into q and i. It then uses the case operator [+] to branch based on the value of i. Depending on whether i contains 1 or 2, msg receiver sends 2 * q to either rm1 or rm2 (rm stands for "receive message"). To complete the forward communication path, the right link1 and right link2 processes read the values from rm1 or rm2 and place them on right1 and right2 respectively.

Once a value has been sent across the communication link, an acknowledgment token (in this case the value 1) is sent back to the sender in a similar fashion.

The channels sm1, sm2, rm1, rm2, sa1, sa2, ra1 and ra2 in the implementation are hidden channels. All events that occur on such channels are hidden (or invisible). Channels that are not hidden are externally visible, and these are the channels that we preserve the behavior of when checking refinement.

III. Approaches to refinement checking

Our goal is to check that for any trace in the implementation, there is an equivalent trace in the specification. One approach for checking trace refinement, used in FDR [2], 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.

Unfortunately, our refinement example from Section II is not finite, because we 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, since both the specification and the implementation are "inspecting" the integers when multiplying 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, and the implementation erroneously sets the output to 2 times the input (instead of 4 times), we would not detect the error. FDR cannot check the refinement of such infinite data-dependent CSP systems (which we shall henceforth call *IddCSP* 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,

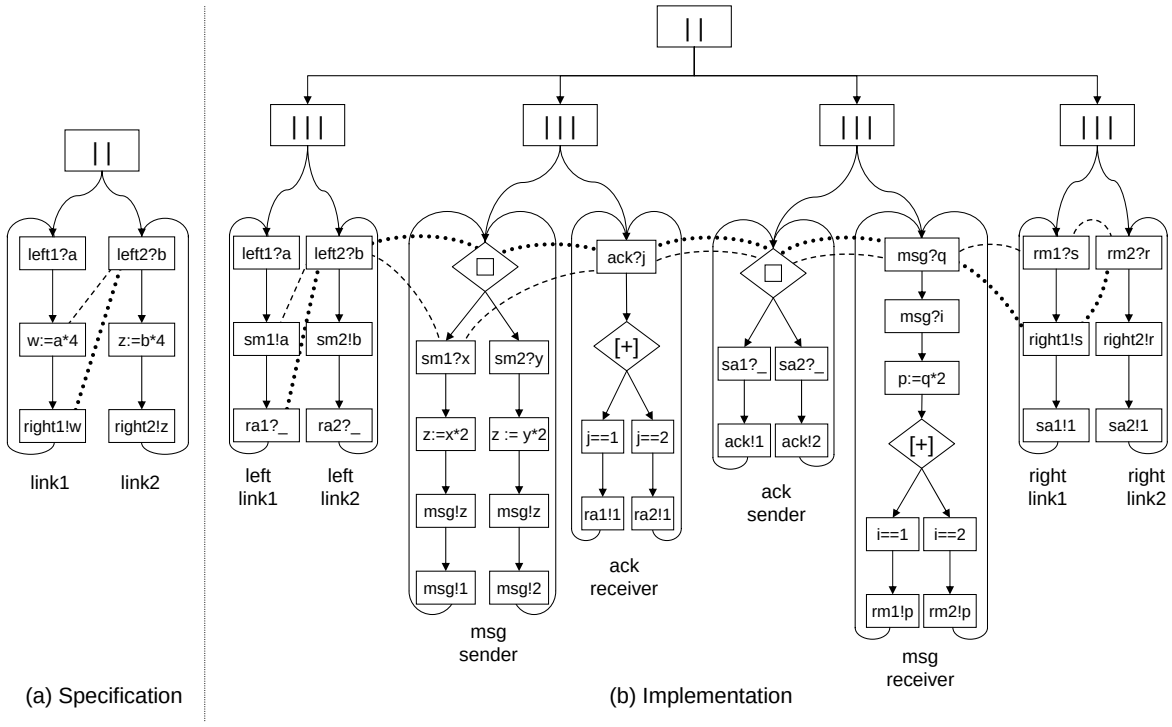

Fig. 1. CFGs for the specification and implementation of our running example

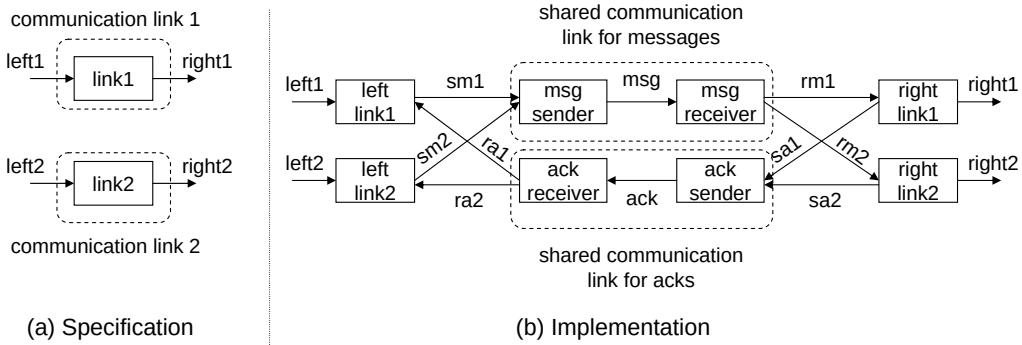

Fig. 2. Process communication diagram for the specification and implementation of our running example

Name	Line type	Condition
A	----------	Spec.a == Impl.a
B	··········	Spec.w == Impl.s

Fig. 3. Sample entries from the simulation relation

despite many techniques that have been developed for checking larger and larger finite state spaces [6], [10], [29], [31], the state space can still grow to a point where automation is impossible. For example, we tried checking the refinement from Section II in FDR using 32-bit integers as values, and the tool had to be stopped because it ran out of memory after several hours (Our algorithm, in contrast, is able to check this example for any sized integers, not just 32-bit integers, in about 4 seconds).

An approach that seems much better suited for *IddCSP* systems is the relational approach of Josephs [17]. Relational approaches are a common tool for reasoning about programs, and they are used in a variety of contexts, including model checking [18], [7], translation validation [30], [26], and reasoning about optimizations once and for all [20], [4]. Josephs presents a relational approach for refinement checking of CSP programs. Intuitively, the idea is to show that there exists a relation R that matches a given program state in the implementation with the corresponding state in the specification. The relation $R \subseteq State_1 \times State_2$ operates over the program states $State_1$ of the specification and the program states $State_2$ of the implementation. If $Start_1$ is the set of start states of the specification, $Start_2$ is the set of start states of the implementation, and $\sigma \rightarrow^e \sigma'$ denotes state σ stepping to state σ' with observable events e, then the following conditions summarize Josephs requirements for a correct refinement:

$$\forall \sigma_2 \in Start_2 \;.\; \exists \sigma_1 \in Start_1 \;.\; R(\sigma_1, \sigma_2)$$
$$\forall \sigma_1 \in State_1, \sigma_2 \in State_2, \sigma_2' \in State_2 \;.$$
$$\sigma_2 \rightarrow^e \sigma_2' \wedge R(\sigma_1, \sigma_2) \Rightarrow$$
$$\exists \sigma_1' \in State_1 \;.\; \sigma_1 \rightarrow^e \sigma_1' \wedge R(\sigma_1', \sigma_2')$$

These conditions respectively state that (1) for each starting state in the implementation, there must be a related state in the specification;

and (2) if the specification and the implementation are in a pair of related states, and the implementation can proceed to produce observable events e, then the specification must also be able to proceed, producing the same events e, and the two resulting states must be related. The above conditions are the base case and the inductive case of a proof by induction showing that the implementation is a trace refinement of the specification. Although this relational approach can handle *IddCSP* systems, Josephs's work was centered on applying the technique by hand, whereas our goal is to design a fully automated technique.

More generally, there has been little work on checking trace refinement (and other refinements too) of two truly infinite CSP systems in a completely automatic way. Various tools have been developed for reasoning about such CSP systems [12], [34], [16], using a variety of theorem provers [27], [28]. But all these tools are interactive in nature, and they require some sort of human assistance, usually in the form of a proof script that states which theorem proving tactics should be applied to perform the proof.

Although not directly in the context of CSP, there has been work on checking refinement of concurrent systems, for example in the context of the MAGIC tool [8]. However, our approach is different from MAGIC's counter-example driven approach, and it is also considerably simpler. We show that our seemingly simple approach, which was inspired by Necula's work on translation validation [26], in fact works well in practice.

IV. Overview of our approach

Our technique for refinement checking builds on Josephs's relational approach by overcoming the difficulties of automation with a simple division-of-labor approach. In particular, we handle infinite state spaces by splitting the state space into two parts: the control flow state, which is finite, and the dataflow state, which may be infinite. The exploration of the control flow state is done using a specialized algorithm that traverses our internal CFG representation of CSP programs. Along paths discovered by the control flow exploration, the dataflow state is explored using an automated theorem prover. Although this way of splitting the state space has previously been used in reasoning about a given sequential program [13], [14], [3], a given concurrent program [8], a pair of sequential programs [30], [26], or specific low-level hardware refinements [23], [24], its use in reasoning automatically about the refinement of two general purpose infinite state space concurrent programs is novel.

Our approach consists of two parts, which theoretically are independent, but for practical reasons, we have made one part subsume the other. 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 refinement-checking relation. The second part is an inference algorithm that infers a relation given two CSP programs. However, because the inference algorithm does a similar kind of exploration as the checking algorithm, we have 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 a human wants to provide the relation by hand. This section presents an overview of our approach and describes how it works on our running example. Whereas, the next Section V describes the algorithms in details.

A. 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 in our algorithm consists of a set of entries of the form (p_1, p_2, ϕ), where p_1 and p_2 are program points in the specification and implementation respectively, and ϕ is a boolean formula over variables of the specification and implementation. The pair (p_1, p_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. As an example, Figure 1 pictorially shows two simulation relation entries. We use lines to represent the control component of entries in the simulation relation by connecting all the nodes in the CFG that belong to the entry being represented (the actual program point that belongs to the entry is the program point right before the node). The data component of these two entries are given in Figure 3.

The first entry in Figure 1, shown with a dashed line and labeled A in Figure 3, shows the specification just as it finishes reading a value from the left1 channel. The corresponding control state of the implementation shows left link1 as it finishes reading from the left1 channel. The msg sender process at this point has already chosen the sm1?x branch of its $\Box$ operator, since the left link1 process is about to execute a write to the sm1 channel. All other processes in the implementation are at the top of their loops. For this entry, the relevant data invariant is $Spec.a == Impl.a$, which states that the value of a in the specification is equal to the value of a in the implementation. This is because both the specification and the implementation have stored in a the same value from the surrounding environment.

The next entry in the simulation relation is shown in Figure 1 with a dotted line and is labeled B in Figure 3. In running from A to B, the specification executes w := a*4, while the implementation goes through the following steps: (1) left link1 sends the value a over sm1; (2) msg sender reads this value into z, then sends z*2 over msg, and finally returns to the top of its loop; (3) msg receiver reads this z*2 value from msg and sends twice that (in essence z*4) on rm1; (4) right1 reads this z*4 value into s and gets ready to write it to right1; (5) all other processes in the implementation don't step.

The relevant invariant at B is $Spec.w == Impl.s$. Indeed, if we combine the invariant from A (which is $Spec.a == Impl.a$), with the fact that the specification executes w := a*4, and the fact that the cumulative effect of the implementation is to set s to the value a*4, we get that $Spec.w == Impl.s$ holds at B. Furthermore, at B the specification is about to write w to the right1 channel and the implementation is about to write s to the same channel. The invariant $Spec.w == Impl.s$ at B therefore implies that the specification and the implementation will produce the same value on the externally visible right1 channel.

Execution from B can reach back to A, establishing the invariant $Spec.a == Impl.a$, since by the time execution reaches A 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 IV-C).

The A and B entries in the simulation relation represent two loops that run in synchrony, one loop being in the specification and the other being in the implementation. The invariants at A and B can be seen as loop invariants across the specification and the implementation, which guarantee that the two loops produce the same effect on the surrounding environment. The control part of the A and B entries guarantee that the two loops are in fact synchronized.

The A–B synchronized loops are only one of many loop pairs in this example. Nominally, one has to have 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. Because there can be many possible paths through a loop, writing simulation relations by hand is tedious, time consuming and error prone, which points to the need for generating simulation relations automatically, not just checking them.

B. Checking Algorithm

Given a simulation relation, our checking algorithm checks each entry in the relation individually. For each entry (p_1, p_2, ϕ), it finds all other entries that are reachable from (p_1, p_2), without going through any intermediary entries. For each such entry (p_1', p_2', ψ), we check using a theorem prover that if (1) ϕ holds at p_1 and p_2, (2) the specification executes from p_1 to p_1' and (3) the implementation executes from p_2 to p_2', then ψ will hold at p_1' and p_2'.

In our example, the traces in the implementation and the specification from A to B are as follows (where communication events have been transformed into assignments and the original communication events are in brackets):

$Spec.a == Impl.a$		
Specification	Implementation	
	x := a	(sm1!a ↔ sm1?x)
	z := x * 2	
	q := z	(msg!z ↔ msg?q)
w := a * 4	i := 1	(msg!1 ↔ msg?i)
	p := q * 2	
	[+]	
	i == 1	
	s := p	(rm1!p ↔ rm1?s)
$Spec.w == Impl.s$		

Our algorithm asks a theorem prover to show that if $Spec.a == Impl.a$ holds before the two traces, then $Spec.w == Impl.s$ holds after the traces.

If there were multiple paths from A to B, our algorithm checks all of them. Furthermore, although we don't show it here, there are multiple next relation entries that are reachable from A, and our algorithm checks all of them.

C. Inference Algorithm

Our inference algorithm works in two steps. First it does a forward pass over the specification and implementation to find externally visible events that need to be matched in the specification and the implementation for the refinement to be correct. In the example from Figure 1, our algorithm finds that there are two input events and two output events that must be matched (the specification events left1?a, left2?b, right1!w and right2!z should match, respectively, with the implementation events left1?a, left2?b, right1!s and right2!r).

This forward pass also finds the local conditions that must hold for these events to match. For events 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 events right1!w and right1!s would be $Spec.w == Impl.s$, and for the events right2!z and right2!r, it would be $Spec.z == Impl.r$.

For events 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 automatically generated 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 events c?a (in the specification) and c?b (in the implementation) would then be $Spec.i == Impl.j$. 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 $Spec.i == Impl.j \wedge Spec.a == Impl.b$.

Once the first pass of our algorithm has found all matching events, and has seeded all matching events with local conditions, the second pass or our algorithm propagates these local conditions backward through the specification and implementation in parallel, using weakest preconditions. The final conditions computed by this weakest-precondition propagation make up the simulation relation. Because of loops, we must iterate to a fixed point, and although in general this procedure may not terminate, in practice it can quickly find the required simulation relation.

V. Algorithms

Since the inference algorithm subsumes the checking algorithm when a simulation relation is found (which is the case for all the refinement example we ran), and since the checking algorithm is simpler than the inference algorithm, we present only our inference algorithm here.

Our inference algorithm, shown on lines 1-3 of Figure 4, runs in two steps. We assume that the specification and implementation programs are global, and so the InferRelation function takes no arguments and returns a simulation relation. The simulation relation is a map and each entry in it is a triple (p_1, p_2, ϕ), where $p_1 \in GPP$ and $p_2 \in GPP$ are generalized program points in the specification and implementation respectively. A generalized program point represents the control state of a CSP program. It can either be a node identifier, indicating that the given node is about to execute, or it can be a pair of two generalized program points, representing the state of two processes running in parallel.

The InferRelation function first performs a forward pass using the ComputeSeeds function (line 2) to find the points in the specification and the implementation that match, and the seed conditions that must hold at those points. At the same time ComputeSeeds finds the traces between points of interests in the specification and the implementation. A trace is a sequence of generalized program points, not necessarily starting from the beginning of the program. Using the computed seeds and traces, InferRelation then performs a backward pass using the PropagateSeeds function (line 3) to propagate the seed conditions throughout the two programs. The resulting propagated conditions constitutes the simulation relation.

The ComputeSeeds and PropagateSeeds functions are shown in Figures 4 and 5 respectively, and we explain each in turn.

A. Computing Seeds

The ComputeSeeds function performs a forward pass over the specification and implementation programs in synchrony to find externally visible events that must match. The algorithm maintains the following information:

- A map *Seeds* (lines 6-8) from generalized program point pairs (one program point in the specification and one in the implementation) to formulas. This map keeps track of discovered seeds.
- A set *Traces* (line 9) of discovered traces.
- A worklist (line 10) of generalized program point pairs.

```
1.  function InferRelation() : VerificationRelation
2.      let (Seeds, Traces) := ComputeSeeds()
3.      return PropagateSeeds(Seeds, Traces)

4.  function ComputeSeeds() :
5.          ((GPP × GPP) → Formula) × set[Trace]
6.      let Seeds := new map of type GPP × GPP → Formula
7.                   which returns MissingFormula for
8.                   uninitialized entries
9.      let Traces := ∅
10.     let worklist := new worklist of GPP × GPP
11.     let M := new map of type GPP × GPP → Formula
12.              which returns false for uninitialized entries
13.     let C := new map of type GPP × GPP → Ints
14.              which returns 0 for uninitialized entries
15.     M(ι1, ι2) := true
16.     worklist.Add(ι1, ι2)
17.     while worklist not empty do
18.         let (p1, p2) := worklist.Remove
19.         let φ := M(p1, p2)
20.         let T1 := FindEvents([p1])
21.         let T2 := FindEvents([p2])
22.         for each t2 ∈ T2 do
23.             let found := false
24.             for each t1 ∈ T1 do
25.                 if ¬IsInfeasible(t1, t2, φ) then
26.                     if Channel(Event(t1)) ≠
27.                        Channel(Event(t2)) then
28.                         Error("Channels don't match")
29.                     found := true
30.                     Traces := Traces ∪ {t1, t2}
31.                     let lp1 := LastGPP(t1)
32.                     let lp2 := LastGPP(t2)
33.                     Seeds(lp1, lp2) :=
34.                         CreateSeed(Event(t1), Event(t2))
35.                     let ψ := sp(t1, sp(t2, φ))
36.                     let δ := M(lp1, lp2)
37.                     if ATP(ψ ⇒ δ) ≠ Valid then
38.                         let c := C(lp1, lp2)
39.                         if c ≥ limit then
40.                             M(lp1, lp2) := true
41.                         else
42.                             C(lp1, lp2) := c + 1
43.                             M(lp1, lp2) := ψ ∨ δ
44.                         worklist.Add(lp1, lp2)
45.             if ¬found then
46.                 Error("Trace in Impl not found in Spec")
47.     return (Seeds, Traces)

48. function IsInfeasible(t1 : Trace, t2 : Trace,
49.                       φ : Formula) : Boolean
50.     let ψ1 := sp(t1, φ)
51.     let ψ2 := sp(t2, φ)
52.     return ATP(¬(ψ1 ∧ ψ2)) = Valid

53. function FindEvents(t : Trace) : set[Trace]
54.     if VisibleEventOccurs(t) then
55.         return {t}
56.     else
57.         return ⋃_{t' ∈ {t::p | t→p}} FindEvents(t')
```

Fig. 4. Inference algorithm, and algorithm for computing seeds

- A map M (lines 11-12) from generalized program point pairs to a boolean formula approximating the set of the states that can appear at those program points. These formulas will be computed using strongest postconditions from the beginning of the specification and implementation programs, and will be used to find branch correlations between the implementation and the specification. The value returned by M for uninitialized entries is $false$, the most optimistic information.
- A map C (lines 13-14) from generalized program point pairs to an integer describing how many times each pair has been analyzed.

The ComputeSeeds algorithm uses a "control step" relation $\rightarrow \subseteq Trace \times GPP$ that identifies how our CSP programs transfer control flow. We say that $t \rightarrow p$ iff the next generalized program point that is reached by trace t is p. For $t \in Trace$ and $p \in GPP$, we also use $t :: p$ for the trace t with p appended to it, and $[p]$ for the trace of length 1 that only contains p.

The ComputeSeeds function starts by setting the value of M at the initial program points of the specification (ι_1) and the implementation (ι_2) to $true$, and adds (ι_1, ι_2) to the worklist. While the worklist is not empty, a generalized program pair (p_1, p_2) is removed from the worklist. Using an auxiliary function FindEvents, we find the set of all traces T_1 that start at p_1 and end at a communication event that is externally visible, and similarly for T_2 (A communication event is externally visible if it occurs on a channel that is not hidden). For each $t_1 \in T_1$, and $t_2 \in T_2$, we check whether or not it is in fact feasible for the specification to follow t_1 and the implementation to follow t_2. The trace combination is infeasible if the strongest postconditions ψ_1 and ψ_2 of the two traces are inconsistent, which can be checked by asking an automated theorem prover (ATP) to show $\neg(\psi_1 \wedge \psi_2)$. This takes care of pruning within a single CSP program, but also across the specification and implementation.

Once we've identified that the two traces t_1 and t_2 may be a feasible combination, we check that the events occurring at the end of these two traces are on the same channel (lines 26-28). If they are not, then we have found an externally visible event that occurs in the implementation but not in the specification, and we flag this as an error. If the events at the end of t_1 and t_2 occur on the same channel, then we augment the $Traces$ set with t_1 and t_2, and then we set the seed condition for the end points of the traces. The seed condition is computed by the CreateSeed function, not shown here. This function takes two communication events that involve an externally visible channel ext. Assuming the two events are (ext!a $\leftrightarrow$ ext?b) and (ext!c $\leftrightarrow$ ext?d), the CreateSeed function first checks if the communication involves reading from an automatically generated environment process, in other words if the ext!a and ext!c instructions belong to some environment processes. If so, then the generated seed is $Spec.i == Impl.j \wedge Spec.b == Impl.d$, where i and j are the index variables for the automatically generated environment processes in the specification and implementation respectively (see Section IV-C for a description of index variables for environment processes). If the communication does not involve reading from an environment process, then the seed is $Spec.a == Impl.c$.

The rest of the loop computes the strongest postcondition of the two traces t_1 and t_2, and appropriately weakens the formula that approximates the run time state at the end of the two traces. The first step is to compute the strongest postcondition with respect to the trace t_2 and then with respect to the trace t_1 using the sp function (line 35). For a given formula ϕ and trace t, the strongest postcondition $\mathsf{sp}(t, \phi)$ is the strongest formula ψ such that the if the statements in the trace t are executed in sequence starting in a program state satisfying ϕ,

```
58. function PropagateSeeds(
59.         Seeds : (GPP × GPP) → Formula,
60.         Traces : set[Trace]) : VerificationRelation
61.     let M := new map of type GPP × GPP → Formula
62.         which returns true for uninitialized entries
63.     let worklist := new worklist of GPP × GPP
64.     for each (p1, p2) such that Seeds(p1, p2) ≠
65.                                 MissingFormula do
66.         M(p1, p2) := Seeds(p1, p2)
67.         worklist.Add(p1, p2)
68.     while worklist not empty do
69.         let (p1, p2) := worklist.Remove
70.         let ψ := M(p1, p2)
71.         let T1 := set of traces in Traces that end at p1
72.         let T2 := set of traces in Traces that end at p2
73.         for each t1 ∈ T1, t2 ∈ T2 do
74.             let φ := wp(t1, wp(t2, ψ))
75.             let fp1 := FirstGPP(t1)
76.             let fp2 := FirstGPP(t2)
77.             let δ := M(fp1, fp2)
78.             if ATP(δ ⇒ φ) ≠ Valid then
79.                 if (fp1, fp2) = (ι1, ι2) then
80.                     Error("Start Condition not strong enough")
81.                 M(fp1, fp2) := δ ∧ φ
82.                 worklist.Add(fp1, fp2)
83.     return M
```

Fig. 5. Algorithm for propagating seeds

then ψ will hold in the resulting program state. The sp computation itself is standard, except for the handling of communication events, which are simulated as assignments. When computing the strongest postcondition with respect to one trace, we treat all variables from the other trace as constants. The two traces t_1 and t_2 operate over different variables because our internal representation qualifies each variable reference with the program it belongs to, either *Spec* or *Impl*. As a result, the order in which we process the two traces does not matter.

Once the strongest postcondition has been computed, if the formula δ stored in M corresponding to the last GPP of the two traces is not implied by the newly computed strongest postcondition (line 37), then the strongest postcondition is added as a disjunct to δ (line 43). Because propagating strongest postconditions through loops may lead to an infinite chain of formulas at the loop entry, each weaker than the previous, we analyze each program point pair at most *limit* times (line 39), where *limit* is a global parameter to our algorithm. After the limit is reached for a program point pair, its entry in M is set to *true* (line 40), the most conservative information, which guarantees that it will never be analyzed again.

B. Propagating Seeds

The PropagateSeeds algorithm propagates the previously computed seed conditions backward through the specification and implementation programs. The algorithm maintains a map M from generalized program points to the currently computed formulas. When a fixed point is reached, the map M is the simulation relation that is returned.

The algorithm starts by initializing M with the seeds, and adding the seeded program points to a worklist (lines 64-67). While the worklist is not empty, the algorithm removes a generalized program point pair (p_1, p_2) from the list, and reads into ψ the currently computed formula for that pair. For all traces t_1 and t_2 that were found in the previous forward pass (the ComputeSeeds pass), and that end at p_1 and p_2, the algorithm computes the weakest precondition of ψ with respect to the two traces (lines 71-74). For a given formula ψ and trace t, the weakest precondition $\mathsf{wp}(t, \psi)$ is the weakest formula ϕ such that executing the trace t in a state satisfying ϕ leads to a state satisfying ψ. The wp computation is standard, except for the handling of communication events, which are simulated as assignments. Here again, the order in which we process the two traces does not matter because they operate over different variables. Once the weakest precondition has been computed, the algorithm then appropriately strengthens the formula stored at the beginning of the two traces using the computed weakest precondition ϕ (lines 75-81). In particular, if the formula δ stored at the beginning of the two traces does not imply the newly computed weakest precondition, then the weakest precondition is added as a conjunct to δ.

There are additional optimizations we perform that are not explicitly shown in the algorithm. When exploring the control state, we perform a simple partial order reduction [29] that is very effective in reducing the size of the control state space: if two communication events 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 events.

The approach of deriving seeds in a forward pass, and then propagating the seeds in a backward pass using weakest preconditions was inspired by Necula's translation validation work [26] for checking the equivalence of two sequential programs. The intuition behind why such a simple approach works well in practice is that the control flow of the specification and the implementation are often similar, and the relation required to show equivalence are usually simple, involving only linear equalities of variables.

Description	#p	#t	#i	time (PO) min:sec	time (no PO) min:sec
1. Simple buffer	7	7	29	00:00	00:00
2. Simple vending machine	2	2	20	00:00	00:00
3. Cyclic scheduler	11	6	65	00:49	01:01
4. Student tracking system	11	3	63	00:01	00:01
5. 1 comm link	13	11	54	00:01	00:01
6. 2 parallel comm links	22	18	105	00:04	01:28
7. 3 parallel comm links	27	25	144	00:21	514:52
8. 4 parallel comm links	32	32	186	01:11	DNT
9. 5 parallel comm links	37	39	228	02:32	DNT
10. 6 parallel comm links	42	46	270	08:29	DNT
11. 7 parallel comm links	47	53	312	37:28	DNT
12. SystemC refinement	8	8	39	00:00	00:00
13. EP2 system	7	3	173	01:47	01:51

#p - Number of processes
#t - Number of threads
#i - Number of instructions
PO - Partial Order reduction
DNT - Did Not Terminate

TABLE I
REFINEMENT EXAMPLES CHECKED USING OUR TOOL

VI. Evaluation

To evaluate our algorithm we implemented it using the Simplify theorem prover [11] in a verification system called ARCCoS. We then wrote a variety of refinements, and checked them for correctness automatically. The refinements that we checked are shown in Table I, along with the number of processes for each example, the number of parallel threads, the number of instructions, the time required to check each example using partial order reduction (PO), and the time required without partial order reduction (no PO). The first 11 refinements were inspired from examples that come with the FDR tool [2]. The 6th example in this list, named "2 parallel comm links" is the example presented in Section II. We also 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.

In the 13th refinement from Table I, 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 TACAS 05 paper on CSP-PROVER [16]. The EP2 system states how various components, including service centers, credit card holders, and terminals, interact.

Aside from providing refinement guarantees, our tool was also useful in finding subtle bugs in our original implementation of some refinements. For example, in the refinement presented in Section II, we originally did not implement an acknowledgment link, which made the refinement incorrect. In this same refinement, we also mistakenly used parallel composition $||$ instead of external choice $\Box$ in msg sender. Our tool found these mistakes, and we were able to rectify them.

VII. CONCLUSION AND FUTURE WORK

We have presented an automated algorithm for checking trace refinement of concurrent systems modeled as CSP programs. The proposed refinement checking algorithm is implemented in a validation system called ARCCoS and we demonstrated its effectiveness through a variety of examples.

We have expanded the class of CSP programs that FDR can handle with variables that include unbounded integers. FDR cannot handle these programs since it requires variables to have fixed bit widths. Even though some of our examples were taken from FDR's test suite, these examples were changed by converting the data type of the variables to unbounded integers. Once we do that, these programs cannot be handled by FDR. In this sense, our contribution is to incorporate an automated theorem proving capability to FDR's search technique in order to handle infinite state spaces that are data-dependent.

Our work 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. Our ongoing effort is on building a validation system that automatically checks SystemC refinements through translation validation for their use in synthesis environments.

REFERENCES

[1] EP2. `www.eftpos2000.ch`.

[2] Failures-divergence refinement: FDR2 user manual. Formal Systems (Europe) Ltd., Oxford, England, June 2005.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings PLDI 2001*, June 2001.

[4] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2004*, January 2004.

[5] A. Benveniste, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Heterogeneous reactive systems modeling and correct-by-construction deployment, 2003.

[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 *Proceedings of LICS 1990*, 1990.

[7] Doran Bustan and Orna Grumberg. Simulation based minimization. In David A. McAllester, editor, *CADE 2000*, volume 1831 of *LNCS*, pages 255–270. Springer Verlag, 2000.

[8] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events and deadlocks. *Formal Aspects of Computing Journal*, 17(4):461–483, December 2005.

[9] E. M. Clarke and David E. Long Orna Grumberg. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency, Reflections and Perspectives*, volume 803 of *LNCS*. Springer Verlag, 1994.

[10] C.N. Ip and D.L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[11] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the Association for Computing Machinery*, 52(3):365–473, May 2005.

[12] B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In *TPHOL 97*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.

[13] Susanne Graf and Hassen Saidi. Construction of abstract state graphs of infinite systems with PVS. In *CAV 97*, June 1997.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *POPL 2002*, January 2002.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[16] Yoshinao Isobe and Markus Roggenbach. A generic theorem prover of CSP refinement. In *TACAS '05*, volume 1503 of *Lecture Notes in Computer Science (LNCS)*, pages 103–123. Springer-Verlag, April 2005.

[17] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3(1):9–18, March 1988.

[18] Moshe Y. Vardi Kathi Fisler. Bisimulation and model checking. In *Proceedings of the 10th Conference on Correct Hardware Design and Verification Methods*, Bad Herrenalb Germany CA, September 1999.

[19] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. Automated refinement checking of concurrent systems. Technical report, University of California, San Diego, 2007. `http://mesl.ucsd.edu/pubs/iccad07-tr.pdf`.

[20] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL 2002*, January 2002.

[21] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[22] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 70–75, New York, NY, USA, 1997. ACM Press.

[23] Panagiotis Manolios, Kedar S. Namjoshi, and Robert Summers. Linking theorem proving and model-checking with well-founded bisimulation. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 369–379, London, UK, 1999. Springer-Verlag.

[24] Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic verification of safety and liveness for xscale-like processor models using web refinements. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10168, Washington, DC, USA, 2004. IEEE Computer Society.

[25] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[26] George C. Necula. Translation validation for an optimizing compiler. In *PLDI 2000*, June 2000.

[27] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE 92*. Springer-Verlag, 1992.

[28] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecure Notes in Computer Science*. Springer Verlag, 1994.

[29] D. Peled. Ten years of partial order reduction. In *CAV 98*, June 1998.

[30] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166, 1998.

[31] 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 10^{20} dining philosophers for deadlock. In *TACAS '95*, 1995.

[32] Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(1):17–32, 2004.

[33] J. P. Talpin, P. L. Guernic, S. K. Shukla, F. Doucet, and R. Gupta. Formal refinement checking in a system-level design methodology. *Fundamenta Informaticae*, 62(2):243–273, 2004.

[34] H. Tej and B.Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In *FME 97*, 1997.