Automated Refinement Checking of Concurrent Systems

Sudipta Kundu   Sorin Lerner   Rajesh Gupta

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

August 2007

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.

1 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 [6]. For instance, researchers have advocated the use of SystemC models and their stepwise refinement into interacting hardware and software components [24]. 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 [23, 37]. Communicating Sequential Processes [18] (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. Broadly speaking, refinements are useful because they provide guarantees about the two programs that are in the refinement relation. Many notions of refinement exist, for example trace refinement, failures refinement, or failures/divergence refinement [35]. Each kind of refinement provides its own guarantees. For example, trace refinement preserves safety properties, failures refinement also preserves liveness properties and deadlock freedom, and failures/divergence refinement also preserves livelock freedom [35].

Refinement checking of CSP programs has many applications. As one example, Allen and Garlen have shown how CSP programs can be used as types to describe the interfaces of software components [3]. The refinement relation becomes a sub-typing relation, and refinement checking can then be used to determine if two components, whose interfaces are specified using CSP programs, are compatible.

Another important application of refinement checking appears in the context of refinement-based software or hardware design [38]. The engineer starts with a high-level description of the design, usually called a specification, which is then continually refined into more and more concrete implementations. Checking correctness of these refinement steps has many benefits, including finding bugs in the refinements, while at the same time guaranteeing that properties checked at higher-levels in the design are preserved through the refinement process, without having to recheck them at lower levels. For example, if one checks that a given specification satisfies a safety property, and that an implementation is a correct trace refinement of the specification, then the implementation will also satisfy the safety property.

Refinement checking of CSP programs is an area that has been widely explored. However, as we will explain in further detail throughout the rest of the paper, previous 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 [20], to semi-automated techniques where humans must provide hints to guide a theorem prover in checking refinements [13, 39, 19, 27, 28, 29]. Second, there has been work on fully automated techniques to CSP refinement checking [2, 10], 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 [7, 11, 33, 36], 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 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 [25, 26].

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 [34, 30], theorem proving [12], and relational approaches to reasoning about programs [20, 22]. 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 [20]. 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 [34, 30], 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 theorem prover [12] 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].

The remainder of the paper is organized as follows. Section 2 presents an overview of our approach, and shows how our algorithm works on a simple

$$
\begin{array}{rcl}
\mathit{CSP} & ::= & (\texttt{decl}\ \mathit{PID} = P)* \\
P & ::= & \mathit{PID} \mid v := E \mid P; P \mid P\ \texttt{||}\ P \mid P\ \texttt{|||}\ P \mid \\
 & & e \to P \mid b \to P\ \texttt{[+]}\ b \to P \mid P\ \sqcap\ P \mid P \,\Box\, P \mid \\
 & & P \setminus \{c, \ldots, c\} \\
e & ::= & c\ ?\ v \mid c\ !\ v \\
E & ::= & \text{side-effect free expressions} \\
b & ::= & \text{side-effect free boolean expressions} \\
\mathit{PID} & ::= & \text{process identifiers} \\
c & ::= & \text{channel identifiers} \\
v & ::= & \text{variable identifiers}
\end{array}
$$

Figure 1: CSP grammar

example. Sections 3 and 4 then provide the full details of our algorithm. In particular, Section 3 covers the checking algorithm, which verifies that a given simulation relation is a correct refinement checking relation. Section 4 then presents our inference algorithm, which automatically infers a simulation relation from a specification program and an implementation program. Section 5 describes our experimental results, Section 6 describes related work, and finally Section 7 presents our conclusions and plans for future work.

2 Overview

This section presents an overview of our approach. We start out in Section 2.1 by describing the salient features of CSP required for understanding the examples in this paper, and the representation of CSP programs we will use throughout the rest of the paper. In Section 2.2 we introduce a CSP refinement example that will serve to illustrate our technique. Section 2.3 describes a variety of previous approach and why they don't handle our running example in a fully automated manner. Finally Section 2.4 describes our approach and how it works on our running example.

2.1 CSP

The grammar from Figure 1 describes the fragment of CSP we will use to illustrate our approach. A CSP program is a set of (possibly mutually recursive) process definitions. The simple cases for a process include the following: a reference to a process by name (PID); an assignment statement ($v := E$); a sequencing of two processes ($P; P$); a parallel composition of two processes ($P\ \texttt{||}\ P$); and a parallel interleave of two processes ($P\ \texttt{|||}\ P$), which is similar to parallel composition, except that the parallel processes cannot communicate.

We now describe some of the more subtle cases. The event prefixing process $e \to P$ is a process that performs some event e and then proceeds with P. An event e can either be reading a value from a channel c into a variable v ($c?v$), or writing a variable v to a channel c ($c!v$). Reads and writes are synchronous.

The case statement $b_1 \rightarrow P_1$ `[+]` $b_2 \rightarrow P_2$ executes P_1 or P_2 based on which of the two boolean conditions b_1 or b_2 is true. If both are true, the choice is made non-deterministically. If neither is true, the statement can't make progress.

The non-deterministic choice process $P_1 \sqcap P_2$ executes either P_1 or P_2 non-deterministically.

The external choice process $P_1 \Box P_2$ executes either P_1 or P_2 depending on what events the surrounding environment is willing to engage in. For example $(c_1?v \rightarrow P_1) \Box (c_2?v \rightarrow P_2)$ will execute the left or right side of the $\Box$ operator depending on what channel (c_1 or c_2) first has a value. If both channels have a value, then the choice is made non-deterministically. The external choice operator resembles the `select` system call in Unix and Linux systems.

The channel hiding process $P \setminus \{c_1, \ldots, c_n\}$ acts like P but hides all the events that occur on channels $c_1, \ldots, c_n$. Channels that are *not* hidden are externally visible, and these are the channels that we preserve the behavior of when checking refinement.

Parallel processes in our version of CSP (and Hoare's original version too) can only communicate through messages on channels. Although there are no explicit shared variables, these can easily be simulated using a process that stores the value of the shared variable, and that services reads and writes to the variable using messages.

As an example, here is a simple CSP program:

$$\begin{array}{l}
\texttt{decl}\ P = (\mathsf{i} := 100; P_1)\,||\,P_2 \\
\texttt{decl}\ P_1 = (\mathsf{i\%2} == 0 \ \rightarrow \mathsf{c!i} \rightarrow \mathsf{e?i} \rightarrow P_1)\ \texttt{[+]} \\
\qquad\quad\ (\mathsf{i\%2} \mathrel{!=} 0 \rightarrow \mathsf{d!i} \rightarrow \mathsf{e?i} \rightarrow P_1) \\
\texttt{decl}\ P_2 = (\mathsf{c?x} \rightarrow \mathsf{s} := \mathsf{x}/2; \mathsf{e!s} \rightarrow P_2)\ \Box \\
\qquad\quad\ (\mathsf{d?x} \rightarrow \mathsf{s} := \mathsf{x} + 1; \mathsf{e!s} \rightarrow P_2)
\end{array}$$

Figure 2 displays this CSP program using our internal Control Flow Graph (CFG) representation after tail recursion elimination has been performed. The example consists of two processes running in parallel. Process P_1 continually outputs i on either channel c or d depending on whether i is even or not. Process P_2 waits until a value is available on channel c or d. Depending on which channel P_2 ends up reading, P_2 either divides the value it just read by 2 or adds 1 to it, and then outputs the resulting value on channel e. Process P_1 reads this value from channel e, stores it in i, and then goes back to the beginning of its loop. The cumulative effect of these two processes is that each time around the loop in P_1, if i is even it is divided by 2, and if it is odd, it is incremented by 1.

2.2 Refinement Example

We now present 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

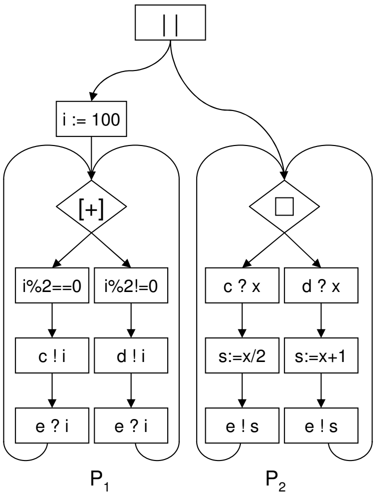

Figure 2: Example CFG representation of a CSP program

using our internal CFG representation in Figure 3(a). We omit the details of the actual CSP code, because the CFG representation is complete, and we believe the CSP code only makes the example harder to follow. The specification states that two parallel processes are continually reading values from two input channels, respectively called left1 and left2. Each process outputs the value $4 * v$ for each input value v that is read from an input channel. The output values are written to two distinct channels, respectively called right1 and right2. Figure 4(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 4(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, where msg stands for message). These two processes communicate over a channel called msg. At the sender end of this communication link, the values from the original

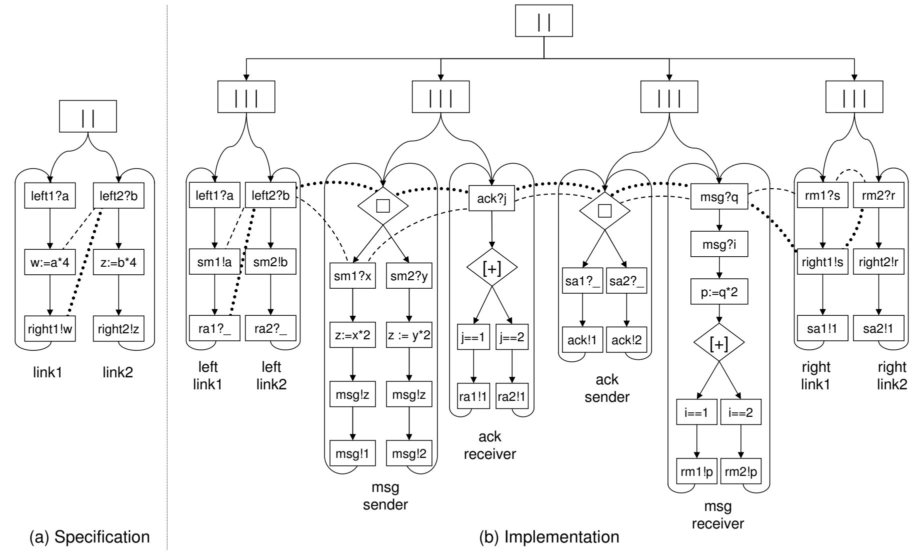

Figure 3: CFGs for the specification and implementation of our running example

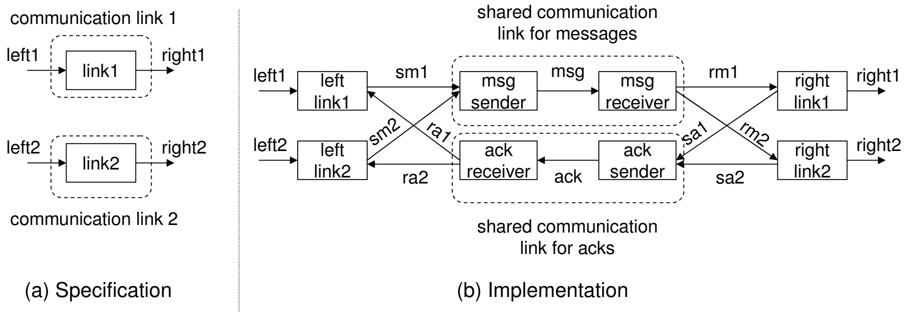

Figure 4: Process communication diagram for the specification and implementation of our running example

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

Figure 5: Sample entries from the simulation relation

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 could not have read any additional values until the current value was written out.

The CFG representation for this implementation is shown in Figure 3(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 value read from these "receive acknowledgment" channels is not used, and so we use an "_" for the variable being read.

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. The immediate successors of the external choice node are used to determine which path to follow. In our case, the two successor nodes 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 two messages on the msg channel: first it sends the value multiplied by 2, and then it sends 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 continually reads two values from the msg channel, and depending on the second value, it sends the first value, multiplied by 2, 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. Acknowledgments are sent from the right link1 and right link2 processes to an ack sender process, which sends acknowledgments for both communication links on a single ack channel. The ack receiver process then decodes the stream of values from this single ack channel to produce an acknowledgment token for either the left link1 or the left link2 process. 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).

2.3 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 2.2 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, despite many techniques that have been developed for checking larger and larger finite state spaces [7, 11, 33, 36], the state space can still grow to a point where automation is impossible. For example, we tried checking the refinement from Section 2.2 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 [20]. Relational approaches are a common tool for reasoning about programs, and they are used in a variety of contexts, including model checking [21, 8], translation validation [34, 30], and reasoning about optimizations once and for all [22, 5]. 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:

$$
\begin{array}{l}
\forall \sigma_2 \in \mathit{Start}_2 \ . \ \exists \sigma_1 \in \mathit{Start}_1 \ . \ R(\sigma_1, \sigma_2) \\
\forall \sigma_1 \in \mathit{State}_1, \sigma_2 \in \mathit{State}_2, \sigma_2' \in \mathit{State}_2 \ . \\
\quad\quad \sigma_2 \rightarrow^e \sigma_2' \wedge R(\sigma_1, \sigma_2) \Rightarrow \\
\quad\quad\quad\quad \exists \sigma_1' \in \mathit{State}_1 \ . \ \sigma_1 \rightarrow^e \sigma_1' \wedge R(\sigma_1', \sigma_2')
\end{array}
$$

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 Josephs's approach can handle *IddCSP* systems, automating his approach turns out to be difficult for two reasons. First, the patterns of quantifiers that appear in the above conditions confuse the heuristics of state of the art theorem provers such as Simplify [12], and as a result, it seems unlikely that a theorem prover could prove the above conditions directly, without any human assistance. Second, to use Josephs's relational approach, one has to come up with a candidate relation to begin with, something that Josephs does not address in his work.

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 [13, 39, 19], using a variety of theorem provers [31, 32]. 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 [9]. 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 [30], in fact works well in practice.

2.4 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 [15, 17, 4, 14], a given concurrent program [9], or a pair of sequential programs [34, 30], its use in reasoning about the refinement of two 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, one of which is a specification, and one of which is an implementation. To check that one CSP program is a refinement of another, one therefore runs the inference algorithm to infer a relation, and then one uses the checking algorithm to verify that the resulting relation is indeed a refinement-checking relation. However, because the inference algorithm does a similar kind of exploration as the checking algorithm, this leads to a lot of duplicate work. To address this issue, 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.

2.4.1 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 3 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 5.

The first entry in Figure 3, shown with a dashed line and labeled A in Figure 5, shows the specification just as it finishes reading a value from the left1 channel. The corresponding control state of the implementation has the left link1 process in the same state, just as it finishes reading from the left1 channel. The msg sender process in the implementation at this point has already chosen the sm1?x branch of its external choice □ 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. As Section 2.4.2 will explain in further detail, our algorithm models the environment as a set of separate processes that are running in parallel with the specification and the implementation. For now we elide these additional processes for clarity of exposition.

The next entry in the simulation relation is shown in Figure 3 with a dotted line and is labeled B in Figure 5. 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 2.4.3).

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.

2.4.2 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.

2.4.3 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 3, 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 $\mathit{Spec}.i == \mathit{Impl}.j$. 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 $\mathit{Spec}.i == \mathit{Impl}.j \wedge \mathit{Spec}.a == \mathit{Impl}.b$.

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.

We use weakest preconditions to infer the simulation relation rather than strongest postconditions because the weakest precondition approach is more goal directed: we start with the seed conditions we want to hold, and then only compute the conditions required to establish these seeds. In contrast, doing strongest postconditions in the forward direction would compute everything that is derivable at a given program point, which would cause our algorithm to often compute facts that are not relevant, and as a result diverge. To see the difference, consider for example a specification and an implementation that both have loops incrementing a variable i and outputting each value of i on an externally visible channel. The strongest post-condition approach on this example would basically be tantamount to running the program, and would not terminate. On the other hand, the weakest precondition approach, if seeded with $\mathit{Spec}.i == \mathit{Impl}.i$ (since those are the values sent on the externally visible channel) would appropriately find out that the invariant is $\mathit{Spec}.i == \mathit{Impl}.i$. Our work in fact confirms Necula's finding in his translation validation work [30] that a weakest precondition approach to inferring simulation relations works well in practice.

3 Checking Algorithm

In this section, we present the details of our algorithm for verifying that a simulation relation is a correct refinement checking relation. The algorithm is shown in Figure 6. We assume that the specification and the implementation programs are global, and so the only parameter to CheckRelation is the relation

```
procedure CheckRelation($R$ : $\mathit{VerificationRelation}$)
    for each $(p_1, p_2, \phi) \in R$ do
        if $\neg$Explore($[p_1], [p_2], \phi, R$) then
            Error(“Trace in Impl not found in Spec”)
    let $(\mathit{Seeds}, \_)$ := ComputeSeeds()
    CheckImplication($R, \mathit{Seeds}$)

function Explore($t_1 : \mathit{Trace}, t_2 : \mathit{Trace}, \phi : \mathit{Formula}$,
                 $R : \mathit{VerificationRelation}$) : $\mathit{Boolean}$
    for each $p_2$ such that $t_2 \rightarrow p_2$ do
        let $\mathit{found} := \mathit{false}$
        for each $p_1$ such that $t_1 \rightarrow p_1$ do
            if $\neg$IsInfeasible($t_1 :: p_1, t_2 :: p_2, \phi$) then
                if $\exists i > 0 \ . \ t_1[i] = p_1 \wedge t_2[i] = p_2$ then
                    Warning(“Loop with no relation entry”)
                elseif $\exists \psi \ . \ (p_1, p_2, \psi) \in R$ then
                    $\mathit{found} := \mathit{true}$
                    PreImpliesPost($\phi, t_1 :: p_1, t_2 :: p_2, \psi$)
                else
                    if Explore($t_1 :: p_1, t_2 :: p_2, \phi, R$) then
                        $\mathit{found} := \mathit{true}$
        if $\neg\mathit{found}$ then
            return $\mathit{false}$
    return $\mathit{true}$

procedure PreImpliesPost($\phi : \mathit{Formula}, t_1 : \mathit{Trace}$,
                         $t_2 : \mathit{Trace}, \psi : \mathit{Formula}$)
    let $\psi' :=$ wp($t_1$, wp($t_2, \psi$))
    if ATP($\phi \Rightarrow \psi'$) $\neq \mathit{Valid}$ then
        Error(“Cannot verify relation entry”)
```

Figure 6: Algorithm for checking a simulation relation

to be checked. The CheckRelation procedure verifies each entry in the simulation relation individually by calling the Explore function. Each entry in the simulation relation is a triple (p_1, p_2, ϕ), where $p_1 \in \mathit{GPP}$ and $p_2 \in \mathit{GPP}$ 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 Explore function uses a "control step" relation $\rightarrow \subseteq \mathit{Trace} \times \mathit{GPP}$ that identifies how our CSP programs transfer control flow. A trace $t \in \mathit{Trace}$ is a sequence of generalized program points $p \in \mathit{GPP}$, not necessarily starting from the beginning of the program. We say that $t \rightarrow p$ iff the next generalized program point that is reached by trace t is p. For $t \in \mathit{Trace}$ and $p \in \mathit{GPP}$, 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 Explore function starts with two traces, t_1 in the specification and t_2 in the implementation. It returns *true* if for all traces t_2' that are extensions of t_2 (that is to say t_2' is t_2 with some additional points at the end of it), there exists a trace t_1' that is an extension of t_1 such that t_1' and t_2' end at a simulation relation entry, and there are no loops in t_1' and/or t_2'. This check intuitively makes sure that for all traces in the implementation, there is a trace in the specification.

For each possible successor program point p_2 in the implementation, the algorithm iterates through each successor program point p_1 in the specification to find some p_1 that leads to a simulation relation. In doing this search, infeasible paths are pruned out on line 12 (the details of the IsInfeasible function will be given when we cover the inference algorithm in Section 4). Furthermore, if a loop is found in the traces (lines 13-14), then a warning message is printed. Strictly speaking it is not an error to find a loop that has no cutting relation – it just means that the algorithm will have to find another path in the specification that accounts for the current path in the implementation, and if no such path is found, the error will occur on line 4.

If an entry in the simulation relation is found with condition ψ (lines 15-17), then we use the PreImpliesPost procedure to check that the condition ϕ at the beginning of the traces implies ψ at the end. The PreImpliesPost procedure computes the weakest precondition ψ' of ψ with respect to the two traces (line 26), and then asks an automated theorem prover (ATP) to show $\phi \Rightarrow \psi'$ (line 27). We perform the weakest precondition computation on one trace and then the other. When computing the weakest precondition with respect to one trace, we treat all variables from the other trace as constants. The two traces 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. 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 itself is standard, except for the handling of communication events, which are simulated as assignments.

```
29. function InferRelation() : $VerificationRelation$
30.     let ($Seeds$, $Traces$) := ComputeSeeds()
31.     return PropagateSeeds($Seeds$, $Traces$)
```

Figure 7: Inference algorithm

Going back to the Explore function, if no simulation entry is found (lines 19-20), we recursively call explore on the new traces. If the recursive call returns *true*, we've met our obligation, and we can set the current *found* variable to *true*. If for a given p_2, there was no p_1 found, then Explore returns *false* (lines 21-22), otherwise it returns *true* (line 23).

There are additional optimizations we perform that are not explicitly shown in the algorithm from Figure 6. When exploring the control state (both in the checking and in the inference algorithm), we perform a simple partial order reduction [33] 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 last step in the CheckRelation function is to make sure that the simulation relation R in fact implies the conditions required to preserve visible events. This is done by first using the ComputeSeeds function to compute a relation *Seeds* that states which events in the specification and the implementation must match, and what conditions must hold at those matching points. The ComputeSeeds function is used for performing inference aswell, and it will be explained in detail in Section 4.1 (ComputeSeeds returns a pair, and here we do not use the second element of the pair). We then call the CheckImplication function (not shown here) to check using a theorem prover that each entry in *Seeds* has a corresponding entry in R that implies it.

4 Inference Algorithm

Our inference algorithm, shown in Figure 7, runs in two steps. Here again, the specification and implementation programs are assumed to be global, and so the InferRelation function takes no arguments and returns a simulation relation. The InferRelation function first performs a forward pass using the ComputeSeeds function (line 30) 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. Using the computed seeds and traces, InferRelation then performs a backward pass using the PropagateSeeds function (line 31) 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 8 and 10 respectively, and we explain each in turn.

```
function ComputeSeeds() :
      $((GPP \times GPP) \rightarrow Formula) \times set[Trace]$
    let $Seeds :=$ new map of type $GPP \times GPP \rightarrow Formula$
              which returns $MissingFormula$ for uninitialized entries
    let $Traces := \emptyset$
    let $worklist :=$ new worklist of $GPP \times GPP$
    let $M :=$ new map of type $GPP \times GPP \rightarrow Formula$
              which returns $false$ for uninitialized entries
    let $C :=$ new map of type $GPP \times GPP \rightarrow Ints$
              which returns 0 for uninitialized entries
    $M(\iota_1, \iota_2) := true$
    $worklist$.Add$(\iota_1, \iota_2)$
    while $worklist$ not empty do
        let $(p_1, p_2) := worklist$.Remove
        let $\phi := M(p_1, p_2)$
        let $T_1 :=$ FindEvents$([p_1])$
        let $T_2 :=$ FindEvents$([p_2])$
        for each $t_2 \in T_2$ do
            let $found := false$
            for each $t_1 \in T_1$ do
                if $\neg$IsInfeasible$(t_1, t_2, \phi)$ then
                    if Channel(Event$(t_1)$) $\neq$
                       Channel(Event$(t_2)$) then
                         Error("Channels don't match")
                    $found := true$
                    $Traces := Traces \cup \{t_1, t_2\}$
                    let $lp_1 :=$ LastGPP$(t_1)$
                    let $lp_2 :=$ LastGPP$(t_2)$
                    $Seeds(lp_1, lp_2) :=$
                         CreateSeed(Event$(t_1)$, Event$(t_2)$)
                    let $\psi :=$ sp$(t_1,$ sp$(t_2, \phi))$
                    let $\delta := M(lp_1, lp_2)$
                    if ATP$(\psi \Rightarrow \delta) \neq Valid$ then
                        let $c := C(lp_1, lp_2)$
                        if $c \geq limit$ then
                            $M(lp_1, lp_2) := true$
                        else
                            $C(lp_1, lp_2) := c + 1$
                            $M(lp_1, lp_2) := \psi \vee \delta$
                        $worklist$.Add$(lp_1, lp_2)$
            if $\neg found$ then
                Error("Trace in Impl not found in Spec")
    return $(Seeds, Traces)$
```

Figure 8: Algorithm for computing seeds

```
75. function IsInfeasible($t_1 : \mathit{Trace}, t_2 : \mathit{Trace}$,
76.     $\phi : \mathit{Formula}) : \mathit{Boolean}$
77.     let $\psi_1 := \mathsf{sp}(t_1, \phi)$
78.     let $\psi_2 := \mathsf{sp}(t_2, \phi)$
79.     return $\mathsf{ATP}(\neg(\psi_1 \wedge \psi_2)) = \mathit{Valid}$

80. function FindEvents($t : \mathit{Trace}) : \mathit{set}[\mathit{Trace}]$
81.     if VisibleEventOccurs($t$) then
82.         return $\{t\}$
83.     else
84.         return $\bigcup_{t' \in \{t::p \mid t \to p\}} \mathsf{FindEvents}(t')$
```

Figure 9: Auxiliary functions

4.1 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 34-35) 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 36) of discovered traces.
- A worklist (line 37) of generalized program point pairs.
- A map M (lines 38-39) 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 40-41) from generalized program point pairs to an integer describing how many times each pair has been analyzed. 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, 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*, the most conservative information, which guarantees that it will never be analyzed again.

The ComputeSeeds algorithm 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 (see Figure 9), 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 a theorem prover 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 53-55). 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 ↔ ext?b) and (ext!c ↔ 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 2.4.3 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 62). 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, 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. Here again, 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 64), then the strongest postcondition is added as a disjunct to δ (line 70). 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 66), where *limit* is a global parameter to our algorithm. Af-

```
85.  function PropagateSeeds(
86.          $Seeds : (GPP \times GPP) \rightarrow Formula$,
87.          $Traces : set[Trace]) : VerificationRelation$
88.    let $M :=$ new map of type $GPP \times GPP \rightarrow Formula$
89.             which returns $true$ for uninitialized entries
90.    let $worklist :=$ new worklist of $GPP \times GPP$
91.    for each $(p_1, p_2)$ such that $Seeds(p_1, p_2) \neq$
92.                                  $MissingFormula$ do
93.      $M(p_1, p_2) := Seeds(p_1, p_2)$
94.      $worklist.\mathsf{Add}(p_1, p_2)$
95.    while $worklist$ not empty do
96.      let $(p_1, p_2) := worklist.\mathsf{Remove}$
97.      let $\psi := M(p_1, p_2)$
98.      let $T_1 :=$ set of traces in $Traces$ that end at $p_1$
99.      let $T_2 :=$ set of traces in $Traces$ that end at $p_2$
100.     for each $t_1 \in T_1,\ t_2 \in T_2$ do
101.       let $\phi := \mathsf{wp}(t_1, \mathsf{wp}(t_2, \psi))$
102.       let $fp_1 := \mathsf{FirstGPP}(t_1)$
103.       let $fp_2 := \mathsf{FirstGPP}(t_2)$
104.       let $\delta := M(fp_1, fp_2)$
105.       if $\mathsf{ATP}(\delta \Rightarrow \phi) \neq Valid$ then
106.         if $(fp_1, fp_2) = (\iota_1, \iota_2)$ then
107.           Error("Start Condition not strong enough")
108.         $M(fp_1, fp_2) := \delta \wedge \phi$
109.         $worklist.\mathsf{Add}(fp_1, fp_2)$
110.   return $M$
```

Figure 10: Algorithm for propagating seeds

ter the limit is reached for a program point pair, its entry in M is set to *true* (line 67), the most conservative information, which guarantees that it will never be analyzed again.

4.2 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 91-94). 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

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

Table 1: Refinement examples checked using our tool (P stands for "Number of Process Description", T stands for "Number of parallel Threads", I stands for "Number of Instructions", PO stands for "partial order reduction", DNT stands for "did not terminate")

that end at p_1 and p_2, the algorithm computes the weakest precondition of ψ with respect to the two traces (lines 98-101). The algorithm then appropriately strengthens the formula stored at the beginning of the two traces using the computed weakest precondition (lines 102-108). 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.

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 [30] 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.

5 Evaluation

To evaluate our algorithm we implemented it using the Simplify theorem prover [12] in a verification system called ARCCoS. We then wrote a variety of CSP refinements, and checked them for correctness automatically. The refinements that we checked are shown in Table 1, 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 2.2. 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.

The 12th refinement in the list is a hardware refinement example taken from a SystemC book [16]. This example models the refinement of an abstract FIFO communication channel to an implementation that uses a standard FIFO hardware channel, along with logic to make the hardware channel correctly implement the abstract communication channel.

In the 13th refinement from Table 1, 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 [19]. The EP2 system states how various components, including service centers, credit card holders, and terminals, interact.

In all of the above examples, since trace subset refinement preserves safety properties, we can conclude that the implementation has all the safety properties of the specification.

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

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 2.2, 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 □ in msg sender. Our tool found these mistakes, and we were able to rectify them.

6 Related Work

As mentioned in the introduction, there has been a long line of work on reasoning about refinement of CSP programs. Our relational checking algorithm was inspired by Josephs's approach [20] for proving refinements. However, Josephs proved refinements by hand, whereas our tool is fully automated. Our searching algorithm through the control state of the program is similar to FDR's searching technique [2], which exhaustively explores the state space. However, as mentioned previously, our tool can handle infinite state spaces that do not trivially reduce using skolemization to finite state spaces.

Various interactive theorem provers have been extended with the ability to reason about CSP programs. As one example, Dutertre and Schneider [13] reasoned about communication protocols expressed as CSP programs using the PVS theorem prover [31]. As another example, Tej and Wolff [39] have used the Isabelle theorem prover [32] to encode the semantics of CSP programs. Isabelle has also been used by Isobe and Roggenbach to develop a tool called CSP-Prover [19] for proving properties of CSP programs. All these uses of interactive theorem provers follow a common high-level approach: the semantics of CSP is usually encoded using the native logic of the interactive theorem prover, and then a set of tactics are defined for reasoning about this semantics. Users of the system can then write proof scripts that use these tactics, along with built-in tactics from the theorem prover, to prove properties about particular CSP programs. Our approach does not have the same level of formal underpinnings as these interactive theorem proving approaches. However, our approach is fully automated, whereas these interactive theorem proving approaches all require some amount of human intervention.

Our inference algorithm was inspired by Necula's translation validation work [30], and bears similarities with Necula's algorithm for inferring simulation relations that prove equivalence of sequential programs. Necula's algorithm collects a set of constraints in a forward scan of the two programs, and then solves these constraints using a specialized solver and expression simplifier. Unlike Necula's approach, our algorithm is expressed in terms of calls to a general theorem prover, rather than using specialized solvers and simplifiers. Our algorithm is also more modular, in the sense that the theorem proving part of the algorithm has been modularized into a component with a very simple interface (it takes a formula and returns *Valid* or *Invalid*).

7 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] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

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

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, June 2002.

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

[16] T. Grötker. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] K. L. McMillan. A compositional rule for hardware design. In *CAV 97*, 1997.

[28] K. L. McMillan. Verification of an implementation of tomasulos algorithm by compositional model checking. In *CAV 98*, 1998.

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

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

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

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

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

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

[35] A. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[36] 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.

[37] 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.

[38] 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.

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