Preventing Glitches and Short Circuits in High-Level Self-Timed Chip Specifications

Stephen Longfield Brittany Nkounkou Rajit Manohar Ross Tate
Cornell University
slongfield@csl.cornell.edu brn25@cornell.edu rajit@csl.cornell.edu ross@cs.cornell.edu

Abstract

Self-timed chip designs are commonly specified in a high-level message-passing language called CHP [21]. This language is closely related to Hoare's CSP [11] except it admits erroneous behavior due to the necessary limitations of efficient hardware implementations. For example, two processes sending on the same channel at the same time causes glitches and short circuits in the physical chip implementation. If a CHP program maintains certain invariants, such as only one process is sending on any given channel at a time, it can guarantee an error-free execution that behaves much like a CSP program would. In this paper, we present an inferable effect system for ensuring that these invariants hold, drawing from model-checking methodologies while exploiting language-usage patterns and domain-specific specializations to achieve efficiency. This analysis is sound, and is even complete for the common subset of CHP programs without data-sensitive synchronization. We have implemented the analysis and demonstrated that it scales to validate even microprocessors.

Categories and Subject Descriptors B.6.3 [*Design Aids*]: Hardware Description Languages; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; D.3.2 [*Programming Languages*]: Concurrent, Distributed, and Parallel Languages

General Terms Algorithms,Verification

Keywords Self-timed chips, AVLSI, QDI, CHP, inferable effect system, synchronization, automata-based model checking

1. Introduction

Information takes time to propagate on a physical chip. Traditionally, chips use a global clock whose tick indicates that there has been enough time for the all information from one stage to propagate to the next stage. An alternative approach is to integrate feedback mechanisms within the chip, communicating when information is available or has been received. This approach is known as *self-timed* chip design, and it addresses some of the major issues currently facing the architecture community.

Self-timed chips are specified in a high-level programming language, such as CHP (Communicating Hardware Processes) [21]. This language is closely related to Hoare's CSP (Communicating Sequential Processes) [11]. For example, $A!$ sends on channel A (where a channel is implemented as a bundle of wires), and $A?$ receives on channel A. The two programs can be run in parallel, as in $A! \parallel A?$, to synchronize via communication.

Behind the scenes, CHP implements these sends and receives with a four-phase handshake. The sender starts sending the data, moving to a hidden state we represent with $A_{¡}$. In this state, the sender is actively driving the voltage on the wires, holding it to the values it wants to communicate. Eventually, the receiver observes that the data is available and starts locally storing that data. Once stored, the receiver moves to a hidden state we represent with $A_{¿}$, indicating that it is actively driving a wire to acknowledge that the message has been handled. Eventually, the sender then receives that acknowledgement and communicates that fact to complete its part of the handshake. Eventually, the receiver recognizes that communication, removes the acknowledgement now that it has been heard, finally completing its part of the handshake.

For this process to work correctly, at any point in time only one process can be attempting to send on a given channel, and at any point in time only one channel can be attempting to receive on a given channel. Thus, unlike in CSP, the CHP program $A! \parallel A? \parallel A! \parallel A?$ is erroneous, causing glitches and short circuits in its physical chip implementation. This is only one of many sources of potential errors that a CHP programmer needs to avoid through careful reasoning about the synchronization of the program. Reasoning about synchronization behavior in a highly concurrent language can be quite a challenge, especially for programmers new to the subtleties of CHP.

In this paper, we present an inferable effect system for the CHP language that ensures that a given program is free of glitches and short circuits. Our effects describe the synchronization behavior of CHP specifications, which we use to guarantee that the conflicting actions that would cause glitches (when a wire moves ambiguously between high and low states) and short circuits (when the power source has a direct path to the ground) never occur. These effects are inferable, meaning we are guaranteed to be able to determine the effect of a specification and check if the specification matches our formal requirements. We have implemented this effect system as an analysis and determined that our effect system is expressive enough to handle the common idioms and large programs such as microprocessors.

After providing a brief overview of self-timed chips and CHP in Section 2, we present the following contributions:

Section 3 A refined state semantics for a critical subset of CHP that improves upon existing work by recognizing errors due to instabilities, and a novel trace semantics that enables compositional reasoning of CHP programs, with a proof of equivalence formally verified in Coq [24].

PLDI'15, June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/10.1145/2737924.2737967

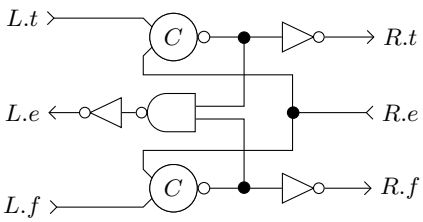


Figure 1. Simple Boolean buffer

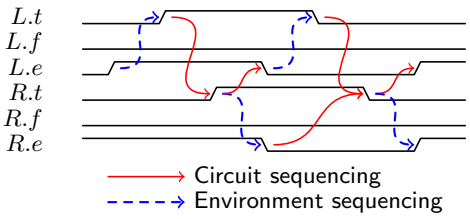


Figure 2. Four-phase handshake

Section 4 An effect system capable of concise sound and complete classification of the externally visible synchronization behavior of a CHP program, and an efficient sound and complete effect-inference algorithm.

Section 5 A discussion of how to soundly extend our analysis to data-carrying channels such that incompleteness occurs only when the uncommon practice of data-sensitive synchronization is used.

Section 6 A demonstration that our analysis can scale to large systems, with strengths that complement existing techniques such as partial-order reduction, as shown by comparison to the SPIN model checker [12].

We establish the context of our work in the literature in Section 7, and then look to lessons learned in Section 8.

2. Background

Before we can describe how to avoid glitches and short circuits in hardware systems, it is important to understand how these systems work. In this section, we give some background on self-timed VLSI, the CHP language that we use to describe the specification of self-timed circuits, and how CHP specifications are translated into hardware.

2.1 Self-Timed VLSI

Very Large Scale Integration (VLSI) refers to circuit systems with at least 100,000 transistors, often many more. The majority of these systems are *synchronous* VLSI systems, where operations execute in lockstep as coordinated by a centralized clock signal. This clock signal is expensive to propagate throughout the chip, sometimes consuming over one third of the total power and requiring complex design considerations to avoid inconsistency [36]. Particularly troublesome is the fact that the clock must be as slow as the worst-case behavior of the slowest subsystem, forcing designers to spend a great deal of time and silicon area optimizing the timing behavior of corner cases instead of focusing on the more frequently exercised cases [7]. As wire delay begins to outweigh gate delay, this strategy is growing increasingly costly [10]. These problems are compounded by the presence of temperature, voltage, and manufacturing-process variations, any of which may alter the delay of gates and wires.

A possible alternative that avoids these problems is *self-timed*, or clockless, designs. By reducing the dependence on timing assumptions, these designs become more resilient to manufacturing variation and require less analog verification after digital verification than similar synchronous designs [29]. Additionally, since there is no clock speed limited by the worst-case behavior, it is possible to achieve average-case performance.

The self-timed family that makes the fewest timing assumptions is delay-insensitive (DI) circuits, which only require gate and wire delay to be finite. Unfortunately, the class of DI circuits is very small, and it is impossible to even use an OR gate in a fully DI manner [23]. A common weakening of DI systems is to allow for some wire forks (i.e. branches in the connectivity graph) to be considered as *isochronic forks* where the difference in delays between the branches of the forks are required to be shorter than the delay of the gates that the branches feed into. This class of circuits is known as *quasi delay-insensitive* (QDI) and has been found to be Turing complete [17].

Since their introduction, QDI circuits have found applications in both low-energy and high-performance systems, including full microprocessors [18], very-low-power floating-point units [31] and GPS processors [34], high-frequency interconnect networks in field-programmable gate arrays (FPGAs) [8], and on-chip networks for neuromorphic circuits [1]. The International Technology Roadmap for Semiconductors estimates that 40% of new designs will contain some self-timed components by 2020, and marks tools for these systems as an important area of research [13].

2.2 CHP

A common language for high-level specification of QDI systems is the Communicating Hardware Processes (CHP) language [37]. It is inspired by Hoare's Communicating Sequential Processes (CSP) [11], modified to be more amenable to hardware design. One such modification is the addition of *probes*: the ability to test for whether another process is ready to communicate on a channel [19]. Another addition is that it allows shared channels (i.e. channels with multiple senders and multiple receivers) instead of restricting channels to having a single sender and a single receiver [20]. This particular feature has been found to be very important in building high-performance hardware systems [18]. Allowing for multiply-shared channels can reduce resource requirements while increasing concurrency.

2.3 QDI Circuits

An example of a QDI circuit is shown in Figure 1, taken from [15]. This circuit represents a stage of a simple FIFO buffer, reading in from a left channel (represented with the signals $L.t$, $L.f$, and $L.e$), and sending out on a right channel (the signals $R.t$, $R.f$, and $R.e$). It is implemented with standard CMOS NAND and NOT gates, as well as Muller consensus elements (C-elements) [27]. Common and fundamental for QDI systems, C-elements copy the values on their inputs to their output when all of the input values agree, and maintain the current value of the output otherwise.

Each of the channels is implemented by a dual-rail enable-based four-phase handshake. Dual rail is a delay-insensitive code [38] that specifies the data representation: one of the wires (either $L.t$ or $L.f$) being high (i.e. on) represents a possible Boolean value, (`true` or `false`, respectively), and when both wires are low (i.e. off), the value is in a *neutral* state (i.e. there is no value).

The communication protocol for the buffer is implemented as a four-phase handshake. An example four-phase handshake is depicted in Figure 2. Causality in the environment is represented with dashed blue lines, and within the circuit with solid red lines.

The execution proceeds as follows: First, the environment waits for the channel to be enabled (i.e. $L.e$ high), then brings the data to

$$
\begin{aligned}
&\text{Channel } A \\
&\text{Program } P ::= C \mid P; P \mid P \parallel P \mid *P \mid \texttt{skip} \\
&\qquad\qquad \mid [G \to P \,[]\, G \to P] \mid [G \to P \mid G \to P] \\
&\text{Communication } C ::= A! \mid A? \\
&\text{Guard } G ::= \overline{A} \wedge \cdots \wedge \overline{A}
\end{aligned}
$$

Figure 3. Syntax of CHP

a valid state by bringing $L.t$ high, encoding a **true** token. As the channel R is enabled (i.e. $R.e$ is high), the circuit brings $R.t$ high. When this occurs, the circuit is done with $L.t$ and acknowledges L by lowering the enable, and the environment does same when it is done with $R.t$. When both of these transitions complete, the circuit sets $R.t$ low and $L.e$ high, and the environment sets $R.e$ high, returning to the initial state. In this execution, actions overlap significantly. However, as long as the ordering required by the causality relations is satisfied, the transition timing can be freely altered.

2.4 Abstraction Requirements

Before manufacture, QDI designs must be compiled from CHP to a network of logic gates. These gates are described by two Boolean predicates, a *pull-up*, which sets the gate's output to a **true** state when satisfied, and a *pull-down*, which sets the gate's output to a **false** state when satisfied. For gate networks to correctly describe the behavior of a QDI silicon implementation, these assignments must be both *stable* and *non-interfering* [21]. As an example, in Figure 1, the C-element connected to $L.t$ has a pull-up of $\neg L.t \wedge \neg R.e$ and a pull-down of $L.t \wedge R.e$.

A gate network is stable if, whenever one of these statements is satisfied, it remains satisfied until the resulting assignment is completed (e.g. if the pull-up is satisfied, it remains satisfied until the output of the gate is set to **true**). This is an execution property that allows gate transitions to be modeled as atomic.

A gate network is non-interfering if it never enters a state where both a pull-up and a pull-down for a node are simultaneously enabled. For some gates, it may be the case that ¬pull-up∨¬pull-down is a tautology, but for others, non-interference will depend on the execution behavior. This is very important because a simultaneous assignment to **true** and **false** corresponds to an on-chip short circuit, which may cause irreparable damage.

2.5 Compilation

The typical synthesis procedure, which we use, is *Martin* synthesis, a series of semantics-preserving transformations to compile a specification into a gate network [21]. In order to produce efficient translations, Martin synthesis is traditionally done by hand, requiring the designer to reason in detail about the synchronization behavior of various components of a highly concurrent program. Thus, an automated technique for reasoning about the synchronization behavior of CHP programs would not only aid in identifying potential errors but also assist the designer in this synthesis process.

We can derive the circuit in Figure 1 by applying Martin synthesis to the CHP specification $*(L?x; R!x)$, which describes a circuit that repeatedly reads from a channel L into a Boolean variable x and then sends that value on a channel R [15]. The specification makes assumptions about the behavior of the environment in order to guarantee the absences of glitches or short circuits when executing the resulting physical device. For example, no other process can read from L at the same time as the buffer. The goal of the inferable effect system presented in this paper is to infer those assumptions as an effect and ensure they are not violated by the surrounding context.

3. Semantics

For the sake of concision we present a simplification of CHP that contains all the challenges of full CHP except one: data-sensitive synchronization, which is an uncommon practice that we will discuss in Section 5. The syntax of this sublanguage can be seen in Figure 3.

Informally, these language constructs correspond to:

- $A!$ Dataless send on channel A
- $A?$ Dataless receive on channel A
- $P_1; P_2$ Sequential composition of P_1 and P_2
- $P_1 \parallel P_2$ Parallel composition of P_1 and P_2
- $*P$ Infinite repetition of P
- `skip` Do nothing and continue on
- $[\, G_1 \to P_1$ Branch to P_1 or P_2 once appropriate guard
  $[]\, G_2 \to P_2]$ holds, or error if both hold simultaneously
- $[\, G_1 \to P_1$ Branch to P_1 or P_2 once
  $\mid\, G_2 \to P_2]$ appropriate guard holds

A guard is a conjunction of probes $\bar{A}$ that each hold if a concurrent process is actively sending on channel A. The reason that there are two forms of branching, each inspired by Dijkstra's Guarded Command Language [6], is that the potentially erroneous one (known as deterministic choice) is extremely efficient to implement in hardware, whereas the error-free one (known as non-deterministic choice) is expensive and inevitably suffers from metastability issues [25].

To define the semantics of the language, we will use the additional symbols $A_¡$, $A_¿$, and $\hat{A}$. During the execution of a program, $A!$ represents "ready to send on channel A", and $A_¡$ represents "actively sending on channel A". Similarly, $A?$ represents "ready to receive from channel A", and $A_¿$ represents "actively receiving from A". Lastly, $\hat{A}$ probes whether some concurrent process is actively receiving on A (i.e. $\hat{A}$ will be true if and only if a concurrent process is in the state $A_¿$). For simplicity, we are assuming that all channels are *active-send/passive-receive* channels [21], where a process must be actively sending on a channel before another can begin to receive on it.

3.1 State Semantics

Message-passing systems are often given a semantics where all of the processes communicating on message-passing channels are known. To formalize such a semantics for CHP, we first extend the syntax and then define an operational semantics for the language using this execution syntax, as shown in Figure 4. In this semantics, σ is used as to indicate the current state of channels.

For every channel A in the program, the initial state σ_0 initializes every $\bar{A}$ and $\hat{A}$ to **false**, indicating no process is already actively sending or receiving on any channel. Each step of channel communication asserts some requirement and alters the state. This interaction of requirements and state alterations encodes the four-phase handshake used by the translation to gate networks. Consequently, even though the individual components of the communication execute in parallel, the whole communication will always progress in the following order due to the four-phase handshake:

$$A! \parallel A? \to A_¡ \parallel A? \to A_¡ \parallel A_¿ \to \texttt{skip} \parallel A_¿ \to \texttt{skip} \parallel \texttt{skip}$$

Erroneous Behavior due to Hardware Realities

Note that this representation of the semantics itself implicitly assumes some atomicity. For example, the isochronic-fork assumption we make of the manufacturing process corresponds to the fact that each variable has only one value at a time regardless of its location in the program (i.e. location on the chip). More naïvely, all actions are formalized as a single step, with no mechanism to represent the state of the system or the possible executions that may

Context $E[\cdot] ::= \cdot \mid E; P \mid E \parallel P \mid P \parallel E$
Communication $C += A_{¡} \mid A_{¿}$
Boolean $b ::= \mathbf{true} \mid \mathbf{false}$
Natural Number $n ::= 0 \mid 1 \mid 2 \mid \ldots$

$$\frac{\sigma \models G_i}{\langle [G_1 \to P_1 \mid G_2 \to P_2], \sigma \rangle \to \langle P_i, \sigma \rangle}$$

† σ is a mapping from each $\bar{A}$ and $\hat{A}$ to b

$$\frac{\cdot}{\langle \mathtt{skip}; P, \sigma \rangle \to \langle P, \sigma \rangle} \qquad \dagger\frac{\sigma \models \neg \hat{A}}{\langle A!, \sigma \rangle \to \langle A_{¡}, \sigma[\bar{A} \mapsto \mathbf{true}] \rangle}$$

$$\frac{\cdot}{\langle \mathtt{skip} \parallel P, \sigma \rangle \to \langle P, \sigma \rangle} \qquad \dagger\frac{\sigma \models \hat{A}}{\langle A_{¡}, \sigma \rangle \to \langle \mathtt{skip}, \sigma[\bar{A} \mapsto \mathbf{false}] \rangle}$$

$$\frac{\cdot}{\langle P \parallel \mathtt{skip}, \sigma \rangle \to \langle P, \sigma \rangle} \qquad \dagger\frac{\sigma \models \bar{A}}{\langle A?, \sigma \rangle \to \langle A_{¿}, \sigma[\hat{A} \mapsto \mathbf{true}] \rangle}$$

$$\frac{\langle P, \sigma \rangle \to \langle P', \sigma' \rangle}{\langle E[P], \sigma \rangle \to \langle E[P'], \sigma' \rangle} \qquad \dagger\frac{\sigma \models \neg \bar{A}}{\langle A_{¿}, \sigma \rangle \to \langle \mathtt{skip}, \sigma[\hat{A} \mapsto \mathbf{false}] \rangle}$$

$$\frac{\cdot}{\langle *P, \sigma \rangle \to \langle P; *P, \sigma \rangle} \qquad \frac{\sigma \models G_i}{\langle [G_1 \to P_1 \,[\!]\, G_2 \to P_2], \sigma \rangle \to \langle P_i, \sigma \rangle}$$

† σ_0 maps everything to $\mathbf{false}$

$$\frac{\langle P, \sigma_0 \rangle \to^* \langle P', \sigma' \rangle}{P \xrightarrow{s} P'}$$

Figure 4. Naïve/idealistic state semantics for CHP

$$\frac{\sigma \models G_1 \qquad \sigma \models G_2}{\langle [G_1 \to P_1 \,[\!]\, G_2 \to P_2], \sigma \rangle \to \mathbf{error}}$$

$$\frac{\langle P, \sigma \rangle \to \mathbf{error}}{\langle E[P], \sigma \rangle \to \mathbf{error}} \qquad \frac{\langle P, \sigma_0 \rangle \to^* \mathbf{error}}{P \xrightarrow{s} \mathbf{error}}$$

σ is a mapping from each $\bar{A}$ and $\hat{A}$ to n

$$\frac{\sigma \models \neg \hat{A} \qquad \sigma(\bar{A}) = n}{\langle A!, \sigma \rangle \to \langle A_{¡}, \sigma[\bar{A} \mapsto n+1] \rangle}$$

$$\frac{\sigma \models \hat{A} \qquad \sigma(\bar{A}) > 1}{\langle A_{¡}, \sigma \rangle \to \mathbf{error}} \qquad \frac{\sigma \models \hat{A} \qquad \sigma(\bar{A}) = 1}{\langle A_{¡}, \sigma \rangle \to \langle \mathtt{skip}, \sigma[\bar{A} \mapsto 0] \rangle}$$

σ_0 maps everything to 0

$$\frac{\sigma \models \bar{A} \qquad \sigma(\hat{A}) = n}{\langle A?, \sigma \rangle \to \langle A_{¿}, \sigma[\hat{A} \mapsto n+1] \rangle}$$

$$\frac{\sigma \models \neg \bar{A} \qquad \sigma(\hat{A}) > 1}{\langle A_{¿}, \sigma \rangle \to \mathbf{error}} \qquad \frac{\sigma \models \neg \bar{A} \qquad \sigma(\hat{A}) = 1}{\langle A_{¿}, \sigma \rangle \to \langle \mathtt{skip}, \sigma[\hat{A} \mapsto 0] \rangle}$$

$$\frac{\langle E[P], \sigma \rangle \to \langle E'[P], \sigma' \rangle \qquad \langle P, \sigma \rangle \to \langle P'', \sigma'' \rangle \quad \text{there is no } \sigma''' \text{ such that } \langle P, \sigma' \rangle \to \langle P'', \sigma''' \rangle}{\langle E[P], \sigma \rangle \to \mathbf{error}}$$

Figure 5. Revised/realistic state semantics for CHP

occur between when an action begins to execute and when it completes. This is problematic because the run-time behaviors that need to be avoided to prevent glitches and short circuits all arise from conflicting simultaneous actions.

We can refine our semantics to account for the three kinds of errors that can occur, replacing the rules marked with †. The first kind of error, formalized in the top rules of Figure 5, occurs when both guards of a deterministic choice hold simultaneously. In order to translate this branch efficiently to hardware, the translator relies on an assumption that which branch holds is determined uniquely at run time, an assumption the programmer is supposed to guarantee. Violating this assumption leads to undefined behavior, as summarized by **error**. Note that deterministic choice is preferred over non-deterministic choice because non-deterministic choice is more expensive to implement, impossible to implement without metastability issues [25], and may take an unbounded amount of time to execute [14].

The second kind of error is called interference. Interference corresponds to actual short circuits in the chip, meaning a direct connection from power to ground that causes immediate catastrophe. To account for interference, we need to differentiate between when a state variable is resting at **true** or **false** and when a state variable is being *driven* to **true** or **false**. In particular, interference occurs when one process is driving a variable to **true** while another is driving it to **false**. To keep the presentation concise, we exploit the invariants that a variable is only **true** when it is being actively driven, and that a variable is only ever driven to **false** temporarily (when finishing an $A_{¡}$ or an $A_{¿}$). Thus, we change σ to be a map from variables to a natural number indicating how many processes are actively driving it to **true**, and the variable is **false** if that number is 0. The middle rules of Figure 5 formalize the changes to the †-marked rules of Figure 4 to account for interference.

The third kind of error is called instability. Suppose one part of a chip is waiting for a wire to become true and another part of the chip drives that wire to true and then to false. This is called a glitch, and one way it occurs is when some guard is probing a channel that is actively sending, so the guard's branch can proceed, but then the concurrent sender completes, so suddenly the guard's branch cannot proceed. In an untimed system, it is not clear whether the waiting component observed the temporary truth or not. Worse yet, this can leave dependent wires in an unacceptable middle voltage. CHP programmers need to avoid this kind of instability. We formalize instability with the bottom rule of Figure 5.

The semantics is now rather complicated due to all the possible sources of error. But all this complication is accurate according to the operational semantics by Smith and Zwarico [32], which has been widely accepted by the self-timed VLSI community. The problem is that we are attempting to formalize properties of simultaneous events with a semantic technique not capable of expressing simultaneous actions. From an analysis perspective, this makes it difficult to understand how distinct components of the program affect each other. Furthermore, the use of a global state makes it difficult to reason about a component in isolation. As such, we present our own alternate semantics that more accurately reflects the implementation in hardware, makes it easier to identify interactions, and enables modular reasoning of program components.

3.2 Trace Semantics

Up to this point, the semantics has assumed a closed world, where all communication is done with all senders and receivers completely defined, and each evaluation step is an atomic transformation. However, we have found it useful to consider the semantics of CHP as generating traces, with channel-communication synchronization filtering the set of traces. A trace semantics for CHP has been given before [37], but here we present a much simpler semantics, making it easier to reason about.

Open-World Semantics

In the trace semantics, each transition is given a label as defined by the language given in Figure 6. The structure $L_1 \parallel L_2$ is used to indicate that both L_1 and L_2 occur simultaneously. The $\parallel$ operator is commutative, associative, and has idle as an identity. The label

$$\text{Move } M ::= \mathsf{start} \mid \mathsf{use} \mid \mathsf{finish}$$
$$\text{Step } S ::= M_!^A \mid M_?^A$$
$$\text{Label } L ::= \mathsf{idle} \mid S \mid G \text{ holds} \mid L \parallel L$$

$$\frac{P \text{ does not contain } A_¡ \text{ or } A_¿}{P \xrightarrow{\mathsf{idle}} P} \qquad \frac{\cdot}{A_¡ \xrightarrow{\mathsf{use}_!^A} A_¡} \qquad \frac{\cdot}{A_¿ \xrightarrow{\mathsf{use}_?^A} A_¿}$$

$$\frac{\cdot}{*P \xrightarrow{\mathsf{idle}} P; *P} \qquad \frac{P_1 \xrightarrow{L_1} P_1' \quad P_2 \xrightarrow{L_2} P_2'}{P_1 \parallel P_2 \xrightarrow{L_1 \parallel L_2} P_1' \parallel P_2'} \qquad \frac{P_1 \xrightarrow{L} P_1'}{P_1; P_2 \xrightarrow{L} P_1'; P_2}$$

$$\frac{P \xrightarrow{L} P}{\mathtt{skip} \parallel P \xrightarrow{L} P} \qquad \frac{\cdot}{A! \xrightarrow{\mathsf{start}_!^A} A_¡} \qquad \frac{\cdot}{A_¡ \xrightarrow{\mathsf{finish}_!^A} \mathtt{skip}}$$

$$\frac{P \xrightarrow{L} P}{P \parallel \mathtt{skip} \xrightarrow{L} P} \qquad \frac{\cdot}{A? \xrightarrow{\mathsf{start}_?^A} A_¿} \qquad \frac{\cdot}{A_¿ \xrightarrow{\mathsf{finish}_?^A} \mathtt{skip}}$$

$$\frac{P = [G_1 \to P_1 \,[]\, G_2 \to P_2] \text{ or } P = [G_1 \to P_1 \mid G_2 \to P_2]}{P \xrightarrow{G_i \text{ holds}} P_i}$$

$$\frac{\cdot}{\mathtt{skip}; P \xrightarrow{\mathsf{idle}} P} \qquad \frac{\mathbf{error} \text{ is any program guaranteed to fault}}{[G_1 \to P_1 \,[]\, G_2 \to P_2] \xrightarrow{G_1 \wedge G_2 \text{ holds}} \mathbf{error}}$$

Figure 6. Open-world trace semantics for CHP

L	$\mathbf{pre}(L)$	$\mathbf{post}(L)$	$\mathbf{off}(L)$	$\mathbf{on}(L)$	$\mathbf{drive}(L)$
idle	**true**	**true**	**true**	**true**	**true**
$\mathsf{start}_!^A$	**true**	$\bar{A}$	$\neg\hat{A}$	**true**	$\bar{A}$
$\mathsf{use}_!^A$	$\bar{A}$	$\bar{A}$	**true**	**true**	$\bar{A}$
$\mathsf{finish}_!^A$	$\bar{A}$	**true**	**true**	$\hat{A}$	$\neg\bar{A}$
$\mathsf{start}_?^A$	**true**	$\hat{A}$	**true**	$\bar{A}$	$\hat{A}$
$\mathsf{use}_?^A$	$\hat{A}$	$\hat{A}$	**true**	**true**	$\hat{A}$
$\mathsf{finish}_?^A$	$\hat{A}$	**true**	$\neg\bar{A}$	**true**	$\neg\hat{A}$
G holds	**true**	**true**	**true**	G	**true**
$\parallel$	$\wedge$	$\wedge$	$\wedge$	$\wedge$	$\wedge$

$$\mathbf{compliant}(L) = \bigwedge \begin{matrix} \exists\sigma.\ \sigma \models \mathbf{pre}(L) & \wedge & \sigma \models \mathbf{off}(L) \\ \forall\sigma.\ \sigma \models \mathbf{pre}(L) & \implies & \sigma \models \mathbf{on}(L) \end{matrix}$$

$$\mathbf{noninterferant}(L) = \exists\sigma.\ \sigma \models \mathbf{drive}(L)$$

$$\mathbf{stable}(L) = \bigwedge \begin{matrix} \exists\sigma.\ \sigma \models \mathbf{post}(L) & \wedge & \sigma \models \mathbf{off}(L) \\ \forall\sigma.\ \sigma \models \mathbf{post}(L) & \implies & \sigma \models \mathbf{on}(L) \end{matrix}$$

$$\mathbf{faulty}(L) = \neg\mathbf{noninterferant}(L) \vee \neg\mathbf{stable}(L)$$

$$\frac{P \xrightarrow{L} P' \quad \mathbf{compliant}(L) \quad \mathbf{noninterferant}(L)}{P \xrightarrow{\mathsf{t}} P'} \qquad \frac{P \xrightarrow{L} P' \quad \mathbf{compliant}(L) \quad \mathbf{faulty}(L)}{P \xrightarrow{\mathsf{t}} \mathbf{error}}$$

Figure 7. Closed-world trace semantics for CHP

is used to indicate what is guaranteed to be true of the global state for that label to even be possible, what condition must hold of the global state for the transition to actually occur, and what is necessarily true of the global state after the transition. In this way, they are analogous to the checks and updates on σ in the state semantics.

Figure 6 gives the trace semantics for the language, which generates traces of possible executions. The key components are the channel actions. The transition from $A!$ to $A_¡$ is labeled with $\mathsf{start}_!^A$ to indicate that is has started sending on channel A. The transition from $A_¡$ to itself is labeled with $\mathsf{use}_!^A$ to indicate that while idling in this state the process is actively sending on channel A. The transition from $A_¡$ to `skip` is labeled with $\mathsf{finish}_!^A$ to indicate that it has finished sending on channel A. As for branches, they use the label G holds to indicate they can progress into a branch only if its guard holds.

Each of these labels encodes invariants that must be true of the global state before and after that transition for it to be available (let alone actually taken), which are formalized by the functions **pre** and **post**. For example, $\mathsf{use}_!^A$ is only available if the program before and after has a process in the $A_¡$ state, indicating that the channel A is being sent on and therefore $\bar{A}$ must be true both before and after the label. Each label also encodes requirements that must be true of the system as a whole for the entire system to transition over that label, which is formalized by the functions **off** and **on**. For example, $\mathsf{start}_!^A$ can only be transitioned over by the system if no other component of the system is actively receiving on channel A, i.e. $\hat{A}$ is **off**, whereas $\mathsf{start}_?^A$ can only be transitioned over by the system if some other component of the system is actively sending on channel A, i.e. $\bar{A}$ is **on**. Lastly, each label encodes what force it is imposing upon the system, which is formalized by the function **drive**. For example, $\mathsf{use}_?^A$ actively drives $\hat{A}$ to **true**, whereas $\mathsf{finish}_?^A$ actively drives $\hat{A}$ to **false**. The definitions of these descriptive functions are shown in Figure 7.

Closed-World Semantics

The trace semantics provides an open-world semantics for CHP. Each transition is labeled with the requirements that must hold of the entire system for the transition to actually be taken. Yet at some point we want to close the world so that we can determine how a whole program progresses without needing to ask conditions of a non-existent larger environment. We present such a semantics in Figure 7.

This semantics utilizes three classification functions on transition labels: **compliant**, **noninterferant**, and **stable**. The **compliant** classification indicates whether a transition can be taken: are the requirements described by the label necessarily true of the whole system described by the label. The **noninterferant** classification indicates whether the transition avoids any short circuits: no component of the system is being driven to both **true** and **false** simultaneously. The **stable** classification indicates whether the transition avoids undefined behaviors: the requirements that necessarily held beforehand still hold afterward. The left rule indicates that a transition label that can be taken can progress to a valid program provided it does not cause interference. The right rule indicates that a transition label that can be taken can progress to an erroneous state if it causes interference or instability. Note that a compliant transition label that causes instability (but not interference) can progress to both a valid program and an erroneous state, indicating the non-deterministic nature of instability.

The three classification functions are defined in Figure 7. **compliant** uses $\exists$ for **off** to capture the fact that variables are off unless forced on as indicated by **pre**. And, **compliant** uses $\forall$ for **on** to capture the fact that variables are on only if forced on as indicated by **pre**. The same reasoning applies to **stable**. Lastly, **drive** uses $\exists$ to indicate that there are no contradictions. For clarity, we use the classification function **faulty** to indicate whether a transition label can progress to an erroneous state.

To see how errors might arise, consider the following:

$$A! \parallel A! \parallel A?$$

One partial error-free trace for this program is the following:

$$A! \parallel A! \parallel A? \to A! \parallel A_¡ \parallel A? \to A! \parallel A_¡ \parallel A_¿ \to A! \parallel \texttt{skip} \parallel A_¿$$

This is essentially what would happen in CSP; two processes begin communicating on a channel and no other process uses the channel until the communication is finished. However, in CHP we are not so fortunate, and the following branch is another partial trace for the program, but this time erroneous:

$$A! \parallel A! \parallel A? \to A! \parallel A_¡ \parallel A? \to A_¡ \parallel A_¡ \parallel A_¿ \to A_¡ \parallel \texttt{skip} \parallel A_¿$$

In this trace, the final label is $\neg\mathbf{noninterferant}$: finishing the second $A_¡$ drives $\bar{A}$ to $\mathbf{false}$ but the first $A_¡$ being in progress is also driving $\bar{A}$ to $\mathbf{true}$. Furthermore, the second label is $\neg\mathbf{stable}$: $\hat{A}$ is initially $\mathbf{false}$, permitting the first $A!$ to start, but the $A?$ transitioning to $A_¿$ changes $\hat{A}$ to $\mathbf{true}$, thereby making the starting condition for $A!$ unstable.

3.3 Equivalence of Semantics

We have presented two semantics, each capturing different perspectives of the CHP language and its underlying hardware realities. The state semantics more directly realizes that a chip is a state machine. The trace semantics more directly realizes that a chip is a concurrent machine. Thus, an equivalence of these two semantics significantly increases confidence that they accurately model CHP and the hardware.

Our trace semantics is actually slightly more precise than the state semantics because of simultaneous actions. In particular, there are traces possible in the hardware only if two events happen at the same time. However, these traces only exist for potentially erroneous programs. For example, two channels can start sending simultaneously, be acknowledged, and then finish simultaneously, and no instability or interference occurs, but any interleaving of the events would be erroneous. Thus, the state semantics is a sound approximation of the hardware, and the trace semantics identifies only additional transitions that happen to work by complete luck. The following theorems precisely formalize these intuitions:

Theorem. *For any CHP program P not containing data-carrying channels or any $A_¡$ or $A_¿$, the following two properties hold:*

$$P \xrightarrow{\mathsf{s}} \mathbf{error} \implies P \xrightarrow{\mathsf{t}}{}^{*} \mathbf{error}$$

$$\forall P'.\ P \xrightarrow{\mathsf{s}}{}^{*} P' \implies P \xrightarrow{\mathsf{t}}{}^{*} P'$$

Theorem. *For any CHP program P not containing data-carrying channels or any $A_¡$ or $A_¿$, the following two properties hold:*

$$P \xrightarrow{\mathsf{t}}{}^{*} \mathbf{error} \implies P \xrightarrow{\mathsf{s}} \mathbf{error}$$

$$\forall P'.\ P \xrightarrow{\mathsf{t}}{}^{*} P' \implies \begin{array}{c} P \xrightarrow{\mathsf{s}} P' \\ \textit{or} \\ P \xrightarrow{\mathsf{t}}{}^{*} \mathbf{error} \textit{ and } P \xrightarrow{\mathsf{s}} \mathbf{error} \end{array}$$

Due to the complexity of the semantics, we proved these theorems in Coq to reinforce confidence in their truth [16]. The key insight is that σ in the state semantics is uniquely determined by P and corresponds to $\mathbf{pre}(L)$ for any trace label L possible from P. And even though there are some subtle differences between which executions the two semantics accept, the following formally verified corollary illustrates that both semantics agree on whether or not a program is erroneous:

Corollary. *For any CHP program P not containing data-carrying channels or any $A_¡$ or $A_¿$, the following property holds:*

$$P \xrightarrow{\mathsf{s}} \mathbf{error} \iff P \xrightarrow{\mathsf{t}}{}^{*} \mathbf{error}$$

This result means that we now have a concise, composable, and extensible formalization of the semantics of CHP and the run-time errors we need to prevent. Next, we exploit these properties to design an inferable effect system guaranteed to identify those run-time errors.

4. Inferable Effect System

We can use our newfound trace semantics to design an inferable compositional effect system classifying the synchronization behavior of CHP programs. Compositionality is the property that each subcomponent can be analyzed independently, offering a description of that subcomponent without needing to know its context. When subcomponents are combined, they interact with each other, and each form of composition defines how the descriptions of the subcomponents should be combined together to reflect this interaction. Effects are a description of a program that is independent of the types of its communication channels [35]. Since synchronization in CHP is independent of the types of data channels, synchronization behavior is a kind of effect. Our goal, then, is to design an inferable effect system, where inferability means that we have an algorithm for determining the most precise effect of any CHP program that is expressible and verifiable by our effect system.

4.1 Automata as Effects

The effect (i.e. the synchronization interactions on various channels) of a CHP program can be completely described by a nondeterministic finite automaton whose edges are labeled with trace labels L. The possible traces of a program P correspond to the paths of the NFA ε describing its effect. For example, if the effect of P_1 is described by the NFA ε_1, and similarly P_2 by ε_2, then the effect of $P_1; P_2$ is described by $\varepsilon_1; \varepsilon_2$, denoting the sequential composition of automata where the accepting states of ε_1 are connected to the initial states of ε_2. Any trace of $P_1; P_2$ is a trace of P_1 followed by a trace of P_2, just like how any path of $\varepsilon_1; \varepsilon_2$ is a path of ε_1 followed by a path of ε_2.

Every CHP construct has a simple corresponding NFA construct. For example, `skip` corresponds to the NFA with one state that is both initial and accepting and has a self loop labeled idle. And, if the effect of P_1 is described by the NFA ε_1, and similarly P_2 by ε_2, then the effect of $P_1 \parallel P_2$ is described by $\varepsilon_1 \times \varepsilon_2$, denoting the product automaton whose states correspond to pairs of states from ε_1 and ε_2 and whose edges correspond to pairs of edges from ε_1 and ε_2 (combining the labels using the $\parallel$ operator on labels). For repetition, we take the NFA ε for P and connect all its accepting states to its initial states *and* make them no longer be accepting since $*P$ denotes *infinite* repetition of P. For non-deterministic choice $[G_1 \to P_1 \mid G_2 \to P_2]$, we take the disjoint union of the automata for P_1 and P_2, preserving edge labels and node acceptance, and add an initial node with edges, labeled G_1 holds or G_2 holds as appropriate, to the formerly initial nodes. For deterministic choice, we also add an edge, labeled $G_1 \wedge G_2$ holds, to an error state, indicating that the program fails if both guards hold. Finally, for the communications $A!$ and $A?$ we construct the appropriate three-state automaton with start and finish transitions.

After constructing the automaton ε describing the effect of a CHP program P, one can analyze that automaton to determine whether any errors are possible in P. One explores the automaton like a graph, starting at the initial states, only traversing edges whose labels are $\mathbf{compliant}$ since only those edges meet the requirements for the execution to progress. If one reaches an edge that is also $\mathbf{faulty}$, then the program P can error, as exhibited by the trace corresponding to the path taken to reach the $\mathbf{faulty}$ edge. If no such $\mathbf{faulty}$ edge is ever reached after exploring the entire graph, then the program P is error free, since every trace

of P is exhibited in the NFA ε. Thus a CHP program P without data-carrying channels is error free if and only if no **faulty** and **compliant** edges are reachable via **compliant** edges in the inferred effect ε. Note that this proof relies heavily on the fact that our trace semantics accurately reflects the hardware implementation, as justified by the equivalence to the stateful semantics that is already accepted to be accurate.

4.2 Optimization via Specialization

So far what we have presented is the automaton approach used in model checking [5]. The unfortunate reality of this approach, though, is that the size of the automaton inferred for a CHP program is exponential with respect to the size of the CHP program due to the fact that parallelism in CHP is represented by the product of automata. Consequently, such an approach can only realistically be applied to very small programs. We address this by specializing the process to CHP and its common usage patterns.

A CHP channel is almost always used across only a few concurrent subcomponents of a CHP program. Furthermore, many trace labels L have the property that $\neg\mathbf{compliant}(L)$ implies $\neg\mathbf{compliant}(L \parallel L')$ for all trace labels L' not referencing some channel A. For example, the label $\mathsf{start}_?^A$ will never be compliant alongside any label L' that does not reference A; it needs to be alongside at least $\mathsf{use}_!^A$. This means that as we construct the automaton ε representing some subcomponent P, if we know that no other subcomponents potentially executing in parallel with P reference a channel A, then we can eagerly remove all edges of ε with a label L that has the above property, and doing so will not reduce the precision of the analysis. Furthermore, removing those edges may result in unreachable states, which can also be removed without losing precision.

We implement this optimization by first determining where in the program channels are used exclusively, meaning that a subcomponent can be analyzed with the assumption that no concurrent subcomponent will operate on a particular channel. This is done by simple static analysis of variable usage. Second, we infer the effect automaton compositionally using the process described above, but whenever we need to infer the effect of a parallel program $P_1 \parallel P_2$ that we know has exclusive access to some channel A (or more generally a set of channels), then we construct the filtered result of $\varepsilon_1 \times \varepsilon_2$ lazily. We start at the initial states, then add edges only if they can become **compliant** in the presence of some label not using A, and then add states only if they become reachable via such edges. Afterwards, we remove from all labels the portions operating on the channel A, since exclusivity of A guarantees these portions will have no effect on the final result, and if this results in a label comprised of only idles and uses then we determine whether we can safely merge the connected nodes using standard local analyses for automata. In short, we eagerly filter a lazily-built naïve-product automaton, and then we merge appropriate nodes in the final result.

There is also an optimization exploiting the limited use of probes. Although the trace of the program $A!$ is comprised of a $\mathsf{start}_!^A$ followed eventually by a $\mathsf{finish}_!^A$, this separation of intermediate states is only visible if some concurrent program probes the channel A, meaning it contains a probe $\bar{A}$. So, in our preprocessing we also determine which channels are probed somewhere in the program. If the channel A is never probed, then the programs $A!$ and $A?$ are represented by two-, rather than three-, state automata with a single transition labeled $A!$ or $A?$ respectively.

Each of these optimizations completely preserves precision, thereby maintaining soundness and completeness of our analysis. Through these optimizations we can in practice accomplish exponential savings in the size of the automata constructed during inference and in the number of states and edges considered during those constructions, as illustrated by the following examples.

4.3 Examples

In this section we give a few simple programs and their inferred effects. We depict initial nodes with an incoming edge and accepting nodes with an outgoing edge. We do not display self loops labeled idle or use for the sake of clarity.

Sequenced Computation

$$(I?; A!) \parallel (A?; B!) \parallel (B?; O!)$$

This program is made of three concurrently executing processes communicating over two exclusive channels A and B. Each channel is only sent on or read from by a single process. Each process reads from and sends on one channel. While on its own this is not an example of a realistic program, it illustrates a common pattern in CHP, though typically between the read and write of each process there would be some computation on the data. The pattern illustrated here is three modular computations sequenced by their channel communications, resulting in one larger sequential process reading from channel I and then sending on channel O after processing. Assuming no other process uses A or B or probes I or O, the effect we infer is the three-state automaton $\rightarrow \bullet \xrightarrow{I?} \bullet \xrightarrow{O!} \bullet \rightarrow$, accurately identifying the program's sequential nature while abstracting its internal details.

The effect is inferred through the following big steps:

1. $I?; A!$ is inferred to have effect $I?; A!$
2. $A?; B!$ is inferred to have effect $A?; B!$
3. The filter of $(I?; A!) \times (A?; B!)$ is computed to simply $\rightarrow \bullet \xrightarrow{I?} \bullet \xrightarrow{B!} \bullet \rightarrow$, exploiting exclusivity of A
4. $B?; O!$ is inferred to have effect $B?; O!$
5. The filter of $\left(\rightarrow \bullet \xrightarrow{I?} \bullet \xrightarrow{B!} \bullet \rightarrow\right) \times (B?; O!)$ is computed to $\rightarrow \bullet \xrightarrow{I?} \bullet \xrightarrow{O!} \bullet \rightarrow$, exploiting exclusivity of B

Note that, by exploiting exclusivity, during this process we never compute an automaton with more than 3 states, whereas otherwise we would first compute an automaton with 9 states and then finish with an automaton with 27 states. Thus our technique avoids significant state explosion even for small programs.

Length-3 Buffer

$$*(I?; A!) \parallel *(A?; B!) \parallel *(B?; O!)$$

This time each process repeats ad infinitum. Consequently, this program now implements a length-3 buffer. That is, the environment sends data on I that is asynchronously propagated through the program until it is sent on channel O, and three pieces of data can exist in this pipeline at a time.

Assuming no other process uses A or B, we infer the effect through the following steps:

1. $*(I?; A!)$ is inferred to have effect $*(I?; A!)$
2. $*(A?; B!)$ is inferred to have effect $*(A?; B!)$
3. The filter of $*(I?; A!) \times *(A?; B!)$ is, exploiting exclusivity of A, computed to
$$\rightarrow \bullet \underset{B!}{\overset{I?}{\rightleftarrows}} \circlearrowleft_{I? \parallel B!} \underset{B!}{\overset{I?}{\rightleftarrows}} \bullet$$
4. $*(B?; O!)$ is inferred to have effect $*(B?; O!)$
5. The filtered product of the above two automata is, exploiting exclusivity of B, computed to
$$\rightarrow \bullet \underset{O!}{\overset{I?}{\rightleftarrows}} \circlearrowleft_{I? \parallel O!} \underset{O!}{\overset{I?}{\rightleftarrows}} \circlearrowleft_{I? \parallel O!} \underset{O!}{\overset{I?}{\rightleftarrows}} \bullet$$

Note that there are four states, corresponding to the fact that up to three pieces of data can be in transition at a time.

Unsafe Channel Reuse

A channel roughly corresponds to a bundle of wires on the chip. In the more general setting, one can send data on a channel, and if said data is large then the corresponding bundle is large. To conserve resources, one might try to reuse the same large channel. The program below does so by using channels R_1 and R_2 to identify when the data is ready to be received, and then conceptually sending this large amount of data over the shared channel O.

$$(R_1?; O!) \,\|\, (R_2?; O!)$$

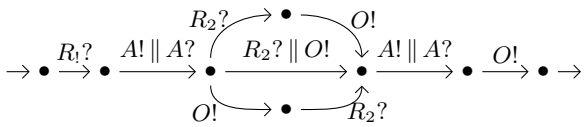
Given that there are no exclusive channels, the effect we infer for this program is the full product automaton but with an **error**-labeled edge where there were simultaneous sends on O. This indicates that an error can occur if the environment fails to use R_1 and R_2 appropriately.

Safe Channel Reuse

Suppose one does not want safety to rely on the environment being implemented correctly. Instead one is willing to assume receiver 2 can wait for receiver 1 to have received its data. However, one would like to at least acknowledge that receiver 2 is ready even if data is still being sent to receiver 1. Then one could use the following program, where A is an exclusive channel.

$$(R_1?; A!; R_2?; A?; O!) \,\|\, (A?; O!; A!)$$

Normally this would result in a 24-node 53-edge (plus 24-self-loop) automaton, but we instead generate the following 8-node 9-edge (plus 8-self-loop) automaton:



In fact, some nodes would be merged because $(A! \,\|\, A?)$-labeled edges become idles, leaving an only 6-state 7-edge (plus 6-self-loop) automaton. Note that, although in building this automaton we would have to consider 10 extra edges, there are still 34 edges we never consider due to building the automaton lazily. Thus, exploiting exclusivity both reduces the size of the automaton built and the time required to build it. The final effect can even be processed and communicated to the programmer intuitively as $R_1?; (O! \times R_2?); O!$, indicating that R_2 can be acknowledged while data is being sent to receiver 1 and there are no faults or even deadlocks.

Uncoordinated Channel Reuse

One can exploit probes to enable recipients to go in either order:

$$[\bar{R}_1 \to R_1?; (O! \,\|\, R_2?); O! \mid \bar{R}_2 \to R_2?; (O! \,\|\, R_1?); O!]$$

Since the receivers send on R_i to indicate they are ready, the program can probe the two channels simultaneously and act on whichever is sent on first. Assuming no one probes O, and using $+$ to denote the coproduct of automata (essentially constructing their disjoint union), the effect we infer is the following:

$$\begin{array}{c}(\bar{R}_1 \;\mathsf{holds}; \mathsf{start}_?^{R_1}; \mathsf{finish}_?^{R_1}; (O! \times (\mathsf{start}_?^{R_2}; \mathsf{finish}_?^{R_2})); O!) \\ + \\ (\bar{R}_2 \;\mathsf{holds}; \mathsf{start}_?^{R_2}; \mathsf{finish}_?^{R_2}; (O! \times (\mathsf{start}_?^{R_1}; \mathsf{finish}_?^{R_1})); O!)\end{array}$$

If instead we had used deterministic choice $[]$, then there would be another case $\bar{R}_1 \wedge \bar{R}_2$ $\mathsf{holds}; \mathbf{error}$ indicating the environment must ensure that both receivers are not ready at the same time.

5. Data-Carrying Channels

So far, all communication that has been described is dataless. However, real systems require communication of data. To do this, data variables x are introduced, sends $A!E$ are extended with expressions, receives $A?x$ are extended to indicate which variable to receive into, and guards are permitted to reference data variables. This extension makes CHP Turing complete [17].

5.1 Semantics

We discuss this extension less formally, since most of the formalities are invisible from the perspective of effect inference. Semantically, each variable x holds a value. When a data-carrying receive $A?x$ starts, the value of the expression in the corresponding data-carrying send-in-progress $A_¡E$ is written to x. If two processes write to a variable simultaneously, that causes interference. If the value of the expression E changes while the send is in progress, that causes instability. Also, due to the intermediate stages that take place at the hardware level, if an expression currently being sent or a guard currently being waited upon depend on a variable that is being written to, that causes instability as well even if the value of the expression or the guard is the same for the old and new values of the variable.

5.2 Effect Inference

In the effect system, we abstract the values of variables, erring on the side of safety. For example, any guard G using a variable is approximated (erring towards **true**) with a guard using just probes. We must also identify interferences and instabilities that may be caused by misuse of variables. So, we extend L to include $\mathsf{read}\ x$ and $\mathsf{write}\ x$. There is an interference whenever two $\mathsf{write}\ x$s occur concurrently, and an instability whenever a $\mathsf{read}\ x$ occurs concurrently with a $\mathsf{write}\ x$. We optimize for exclusivity of variables, much like we did for exclusivity of channels.

Lastly, one subtlety is that, for deterministic choice, it is common for one branch to be guarded by x and the other by $\neg x$, which are clearly mutually exclusive but each get approximated as **true** and **true**, which are not mutually exclusive. Thus, we first construct the guard for mutual exclusion, and then abstract the variables, which in this case results in **false**, recognizing that mutual exclusivity is guaranteed.

5.3 Example: Fetch and Increment

An example of a correctly shared data value is the program counter from a self-timed microprocessor [22]. Below is an adapted fragment of its specification:

$$\begin{aligned} FETCH &\equiv *([\bar{F} \to (M! \,\|\, A!); I?instr; A!; F?]) \\ IMEM &\equiv *(M?; I!imem[pc]) \\ PCADD &\equiv *(A?; x := pc + 1; [\bar{A} \to pc := x; A?]) \end{aligned}$$

$IMEM \,\|\, FETCH \,\|\, PCADD$ with A, M, I, and x exclusive

The syntax $[G \to P]$ is short for $[G \to P \,[]\, \mathbf{false} \to \mathtt{skip}]$, which in effect here makes $FETCH$ wait for $\bar{F}$ to hold. Eventually, F is sent on by the environment to indicate the next instruction needs to be fetched and the program counter incremented. $FETCH$ then sends on M and A to inform $IMEM$ to start loading the instruction and $PCADD$ to start incrementing the program counter. The danger to avoid is modifying the program counter while it is being used to access instruction memory. This is why $PCADD$ is broken into two phases: compute and store the increment of the program counter locally, and then copy that to the program counter only when told to do so. Thus, $FETCH$ only sends on A a second time *after* receiving the instruction from $IMEM$ and storing it into the externally visible variable $instr$. Finally, $PCADD$ acknowledges the second send on A once it has finished updating pc, so

	Our effect-inference analysis					SPIN with partial-order reduction		
Experiment	Naïve NFA States	States Seen	Max Size	RAM (MB)	Time (s)	States Seen	RAM (MB)	Time (s)
GPS acquisition	129,600	1,182	305	10.0	0.25	70,287	191.7	2
Microprocessor	2.39×10^{12}	3,277	1,094	24.2	1.12	•	•	76,035
FIFO, 10 stage	177,147	127	39	6.8	0.07	2,050	181.7	0.6
FIFO, 50 stage	2.15×10^{24}	1,724	681	22.5	1.27	•	•	227,914
FIFO, 100 stage	1.54×10^{48}	6,040	2,519	71.8	1.27	•	•	236,440
Arbitration, 5	34,816	220	60	6.8	0.09	1,824	181.7	0.7
Arbitration, 10	67,108,864	741	120	8.2	0.22	114,688	196.5	3.6
Arbitration, 20	1.36×10^{14}	2,695	240	13.5	0.87	2.32×10^{8}	57,692	17,871
Token-ring, 5	2,048	234	175	7.2	0.05	2	181.6	0.7
Token-ring, 7	32,768	1,523	1,385	24.5	1.76	56	181.6	0.7
Token-ring, 10	2,097,152	44,996	44,287	738.0	442.63	80	181.6	0.7

Table 1. Experimental Evaluation Results. The • indicates SPIN failed to complete because it exceeded 500 GB of RAM

that only then does $FETCH$ acknowledge the external send on F to indicate its job has been completed.

Our effect-inference system derives the following effects:

$$*(\bar{F}\ \mathsf{holds}; (M! \times (\mathsf{st}_!^A; \mathsf{fn}_!^A)); (I? \parallel \mathsf{wr}\ instr); \mathsf{st}_!^A; \mathsf{fn}_!^A; \mathsf{st}_?^F; \mathsf{fn}_?^F)$$
$$*(M?; (I! \parallel \mathsf{rd}\ imem \parallel \mathsf{rd}\ pc))$$
$$*(\mathsf{st}_?^A; \mathsf{fn}_?^A; \mathsf{rd}\ pc; \bar{A}\ \mathsf{holds}; \mathsf{wr}\ pc; \mathsf{st}_?^A; \mathsf{fn}_?^A)$$

Note that the last effect makes no mention of x since x is exclusive to $PCADD$. Next, we combine the first two effects, now exploiting exclusivity of M and I, into the following:

$$*(\bar{F}\ \mathsf{holds}; \mathsf{st}_!^A; \mathsf{fn}_!^A; (\mathsf{wr}\ instr \parallel \mathsf{rd}\ imem \parallel \mathsf{rd}\ pc); \mathsf{st}_!^A; \mathsf{fn}_!^A; \mathsf{st}_?^F; \mathsf{fn}_?^F)$$

Note that we infer that these two concurrent programs produce a deterministic sequential ordering of externally visible events and requirements. Finally, we combine this with the third effect, exploiting exclusivity of A, into the following:

$$*(\bar{F}\ \mathsf{holds}; ((\mathsf{rd}\ pc) \times (\mathsf{wr}\ instr \parallel \mathsf{rd}\ imem \parallel \mathsf{rd}\ pc)); \mathsf{wr}\ pc; \mathsf{st}_?^F; \mathsf{fn}_?^F)$$

Looking at this effect, we can see that two processes read from the program counter concurrently, suggesting the implementation actually exploits parallelism, and actually updates the current instruction and program counter without interferences or instabilities. Furthermore, the acknowledgement of the send on F happens only after these two updates. Thus, from the effect we can determine both that there are no errors (provided the environment executes correctly) and that the program follows appropriate protocol for interacting with the environment.

5.4 Incompleteness

In theory, variables can be used to guarantee safe synchronization behavior, an invariant our analysis cannot recognize. In practice, this is rarely the case. Nonetheless, we provide an example of a safe closed program for which we incorrectly infer the effect **error**:

$$\begin{array}{ll} & A!\mathbf{true} \parallel B!\mathbf{false} \parallel C? \\ \parallel & A?x_0; [x_0 \rightarrow C! \,[]\, \neg x_0 \rightarrow \mathtt{skip}] \\ \parallel & B?x_1; [x_1 \rightarrow C! \,[]\, \neg x_1 \rightarrow \mathtt{skip}] \end{array}$$

Our analysis cannot recognize that x_0 and x_1 are never both **true**, so here it mistakenly believes the two sends on C can happen simultaneously and cause erroneous behavior due to interference.

6. Experimental Evaluation

We implemented our effect-inference analysis in approximately 3,000 lines of Python. We then evaluated our implementation using benchmarks from the specifications of a self-timed microprocessor [22], a self-timed GPS [34], a FIFO buffer, and two systems for mutual exclusion: arbitration and a token-ring network.

For comparison, we implemented a translation from CHP into Promela, the specification language for the SPIN model checker [12]. This model checker was chosen because Promela is similar enough to CHP to make this translation with minimal overhead, and it was simple to express the properties we want to check (stability, non-interference, and guard mutual exclusion). In addition, SPIN supports partial-order reduction, another model-checking technique for reducing search spaces [9].

Evaluations were run on an 80-core 2.13GHz Xeon machine with 1 TB of RAM. To be fair to SPIN, all evaluations were run single-threaded. Our analysis required at most 1 GB of RAM, but SPIN often exceeded our cap of 500 GB.

The results are presented in Table 1. In addition to memory and time measurements, this table presents the size of the full naïve NFA describing the internal and external synchronization behavior, the total number of states that were generated during our analysis, the maximum size of the NFAs described by any effect at any stage, and the total number of states explored by SPIN.

In most cases, our analysis significantly outperforms the SPIN model checker. However, in the case of the token-ring mutual exclusion, SPIN is much better. The token-ring is challenging for our analysis because our analysis is done compositionally, whereas SPIN analyzes a full, closed system. In the case of the token-ring network, before the entire ring is closed, there is significant possibility for error which cannot be reduced. The FIFO was specifically designed antagonistically for SPIN, knowing SPIN's techniques, whereas the token-ring was specifically designed antagonistically for our analysis, knowing our techniques. Thus, our analysis serves as a good complement to partial-order reduction. We have made the our implementations and experimental framework available so that others may examine and reproduce our results [16].

7. Related Work

The semantics of CHP has been formalized before, both as a term-rewriting system [33], and also as generating a tree of traces [37]. In the first case, error semantics were given for interference only, and in the second case there was no consideration of errors at all.

Prior work on this problem in the model-checking community has looked at the connection between CHP and Petri nets [28], and used that connection to translate to existing model-checking techniques [2, 3]. In both cases, the focus on existing model-checking techniques limited the amount of domain-specific reductions that could be performed.

The other work we build upon for this system is the advances in the model-checking community. We complement the work on partial-order model checking, which like us aims to reason only about significant interleavings [9]. However, we reason about these

interleavings implicitly, instead of building up the non-interacting sets explicitly. More significantly, we build on the work of compositional model checking and hierarchical compression [4, 30]. This has been successfully applied to self-timed systems in the past, using a limited temporal logic to reason about the low-level implementations [26], but here we apply it at a higher level.

8. Conclusion

We have created an inferable effect system for identifying the glitches and short circuits possible in CHP programs due to the limitations of hardware realities. The effect system is based on a trace semantics that we developed for CHP and formally proved equivalent in Coq to a traditional state semantics. Each effect precisely describes the externally visible traces of the program. Our inference algorithm executes efficiently due to the compositional design of the effect system and the specialization of our constructions to exploit the channel-exclusivity common in CHP programs. It works by representing the synchronization behavior as an NFA effect denoting the reachable traces, defining composition operators on NFA effects, and searching the inferred effect for reachable errors. Our inference is sound and is only incomplete when synchronization behavior is dependent on data values, which in practice rarely affects the interference and stability errors that we aim to identify. Furthermore, our analysis is efficient, requiring little more than a second and only 24 MB of RAM to guarantee an entire self-timed microprocessor has no glitches or short circuits, as opposed to the SPIN model checker employing partial-order reduction, which failed to complete after 21 hours of processing due to requiring over 500 GB of RAM.

References

[1] J. V. Arthur, P. A. Merolla, F. Akopyan, R. Alvarez, A. Cassidy, S. Chandra, S. K. Esser, N. Imam, W. Risk, D. B. D. Rubin, et al. Building block of a programmable neuromorphic substrate: A digital neurosynaptic core. In *International Joint Conference on Neural Networks*, 2012.

[2] D. Borrione, M. Boubekeur, E. Dumitrescu, M. Renaudin, J.-B. Rigaud, and A. Sirianni. An approach to the introduction of formal validation in an asynchronous circuit design flow. In *Hawaii International Conference on System Sciences*, 2003.

[3] A. Cerone and G. J. Milne. A methodology for the formal analysis of asynchronous micropipelines. In *Formal Methods in Computer-Aided Design*, 2000.

[4] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS*, 1989.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[6] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[7] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. Centrum voor Wiskunde en Informatica, 1989.

[8] D. Fang, J. Teifel, and R. Manohar. A high-performance asynchronous FPGA: Test results. In *Symposium on Field-Programmable Custom Computing Machines*, 2005.

[9] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, 1996.

[10] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[12] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[13] ITRS. International Technology Roadmap for Semiconductors: 2012 update, 2012. URL `http://www.itrs.net/`.

[14] M. Kishinevsky, A. Kondratyev, A. Taubin, V. Varshavsky, A. Yakovlev, E. Napelbaum, and O. Reva. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. John Wiley & Sons, Inc., 1994.

[15] A. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1998.

[16] S. Longfield, B. Nkounkou, R. Manohar, and R. Tate. Preventing glitches and short circuits in high-level self-timed chip specifications artifact. Cornell University, 2015.

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

[18] A. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, 1997.

[19] A. J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125–130, 1985.

[20] A. J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5:265–276, 1985.

[21] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[22] A. J. Martin. The design of an asynchronous microprocessor. Technical report, California Institute of Technology, 1989.

[23] A. J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *MIT Conference on Advanced Research in VLSI*, 1990.

[24] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

[25] M. Mendler and T. Stroup. Newtonian arbiters cannot be proven correct. *Formal Methods in System Design*, 3(3):233–257, 1993.

[26] B. Mishra and E. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38:269–291, 1985.

[27] D. E. Muller and W. S. Bartky. *A Theory of Asynchronous Circuits*. University of Illinois, Graduate College, Digital Computer Laboratory, 1957.

[28] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[29] K. Papadantonakis. *Rigorous Analog Verification of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 2005.

[30] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP or how to check 1020 dining philosophers for deadlock. In *Tools and Algorithms for the Construction and Analysis of Systems*, 1995.

[31] B. R. Sheikh and R. Manohar. An operand-optimized asynchronous IEEE 754 double-precision floating-point adder. In *ASYNC*, 2010.

[32] S. F. Smith and A. E. Zwarico. Provably correct synthesis of asynchronous circuits. *Designing Correct Circuits*, 5:237–260, 1992.

[33] S. F. Smith and A. E. Zwarico. Correct compilation of specifications to deterministic asynchronous circuits. In *Formal Methods in System Design*, 1995.

[34] B. Tang, S. Longfield, S. Bhave, and R. Manohar. A low power asynchronous GPS baseband processor. In *ASYNC*, 2012.

[35] R. Tate. The sequential semantics of producer effect systems. In *POPL*, 2013.

[36] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In *Design Automation Conference*, 1998.

[37] M. van der Goot. *Semantics of VLSI Synthesis*. PhD thesis, California Institute of Technology, 1995.

[38] T. Verhoeff. Delay-insensitive codes – an overview. *Distributed Computing*, 3(1):1–8, 1988.