CERTIFIED SOFTWARE FOR DESIGNING ASYNCHRONOUS CIRCUITS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Brittany Ro Nkounkou

August 2020

CERTIFIED SOFTWARE FOR DESIGNING ASYNCHRONOUS CIRCUITS

Brittany Ro Nkounkou, Ph.D.

Cornell University 2020

Correctness and trust in a piece of software is important, especially for complex and/or critical systems like medical devices, computerized weapons, or any management of private data. Certified software aims to evidence correctness and establish trust in software by pairing executable instructions with mechanically-verifiable proofs that certain properties of the instructions are upheld. In this dissertation, we build certified software that is used to design asynchronous circuits. These circuits differ from synchronous circuits by coordinating communication via local message-passing channels rather than a global clock signal. Asynchronous circuits improve upon the time- and power-efficiency of synchronous circuits, but also introduce the complexity challenges of decentralized circuit communication.

Communicating Hardware Processes (CHP) is a language used to write high-level asynchronous circuit designs. We use the Coq Proof Assistant to build a mechanical formalization of CHP language semantics and a certified CHP program simulator. The semantics formalization is novel in the combination of its mechanization, its comprehensive expression of the CHP language, and its precise abstraction of lower-level circuit design decisions. The simulator is novel in its certification, which mechanically verifies that it simulates a CHP program correctly, and it serves as a model for future Coq-built CHP software tools that can be certified against the mechanical language semantics.

EDUCATIONAL BACKGROUND

In 2008, Brittany Nkounkou received her high school diploma from Wethersfield High School in Wethersfield, CT, graduating fourth in her class.

In 2012, Brittany received her Bachelor of Science in Engineering (B.S.E.) in Computer Science and Engineering from the University of Connecticut in Storrs, CT, graduating summa cum laude, with honors, with minors in Mathematics and Music, and first in her class in the School of Engineering. Also in 2012, she was named a National Science Foundation Graduate Research Fellow and an Alfred P. Sloan Scholar.

In 2016 and 2020, Brittany respectively received her Master of Science and Doctor of Philosophy in Computer Science from Cornell University in Ithaca, CT. During her time as a Cornell graduate student, she spent three semesters at the Cornell Tech campus in New York, NY, and three semesters as an Exchange Scholar at Yale University in New Haven, CT.

Dedicated to Asline Joy, Chanel Hope, and Celestin Noble

ACKNOWLEDGEMENTS

Thank you, God. You mercy me with the blood of your son Jesus. You grace me with the fire of your Holy Spirit. All glory, honor, and praise to you.

Thank you, Mommy and Daddy. You pray for me. You wipe my tears. You lift me up. You support me faithfully.

Thank you to the National Science Foundation, the Alfred P. Sloan Foundation, and the Cornell Graduate School. You fully funded my graduate studies and empowered me to freely choose how I wanted to study.

Thank you to Cornell's Diversity Programs in Engineering, the Institute on Teaching and Mentoring, and the ACM Richard Tapia Celebration of Diversity in Computing Conference. You inspired me to finish.

Thank you, Ross Tate. You introduced me to certified software, led me through the trenches of Coq, persisted in meeting with me, and pushed me over the finish line.

Thank you, Rajit Manohar. You taught me asynchronous circuits and helped me escape Ithaca sooner than later.

Thank you, Hakim Weatherspoon. You remind me to remember and respect who I am and where I come from—no matter what the field of computer science looks like.

Thank you, Bob Constable. Your words and enthusiasm are the strongest antidote to *imposter syndrome* that I have ever had the pleasure of consuming.

Thank you, Alex Schwarzmann. Your "walk-on-water" letter of recommendation was pivotal in my acceptance into Cornell's Computer Science Ph.D. Program.

Thank you, Mr. (Joe) Kess. You introduced me to computer science through your exceptional teaching of AP Computer Science during my junior year at Wethersfield High School.

TABLE OF CONTENTS

1 INTRODUCTION

Software is a set of instructions for a computer to execute. And as computer systems continue to increase in complexity, so does software. Yet correctness and trust in a piece of software is important, especially for complex and/or critical systems like medical devices, computerized weapons, or any management of private data. Certified software aims to evidence correctness and establish trust in software by pairing executable instructions with mechanically-verifiable proofs that certain properties of the instructions are upheld.

In this dissertation, we build certified software that is used to design asynchronous circuits. These circuits differ from synchronous circuits by coordinating communication via local message-passing channels rather than a global clock signal. Asynchronous circuits improve upon the time- and power-efficiency of synchronous circuits, but also introduce the complexity challenges of decentralized circuit communication.

Communicating Hardware Processes (CHP) is a language used to write high-level asynchronous circuit designs. We use the Coq Proof Assistant to build a mechanical formalization of CHP language semantics and a certified CHP program simulator. The semantics formalization is novel in the combination of its mechanization, its comprehensive expression of the CHP language, and its precise abstraction of lower-level circuit design decisions. The simulator is novel in its certification, which mechanically verifies that it simulates a CHP program correctly, and it serves as a model for future Coq-built CHP software tools that can be certified against the mechanical language semantics.

1.1 CERTIFIED SOFTWARE

A piece of software is *certified* when its executable instructions are supplemented by a mechanically-verifiable guarantee that the instructions satisfy some properties. And these are typically properties that a computer could not figure out just by analyzing the instructions alone. This separation of instructions and proof is useful because a user will typically only need to confirm the proof once and then run the instructions as many times as needed.

For example, consider the following program:

```
float main(int i) {
    int x = i % 4;
    int y = 4 - x;
    float z = 1 / y;
    return z;
}
```

This program takes an input integer `i`; computes `i` modulo `4` and assigns the resulting integer to `x`; computes `4` minus `x` and assigns the resulting integer to `y`; computes `1` divided by `y` and assigns the resulting floating point number to `z`; then outputs the value of `z`. This program is certified to have the property that a divide-by-zero error is never encountered during its computation of `1` divided by `y`. This certification is guaranteed in part by the following proof:

$$\dfrac{\dfrac{x = i \,\%\, 4}{x \neq 4} \qquad y = 4 - x}{y \neq 0}$$

This proof infers from the assignment of i modulo 4 to x that x does not equal 4, and infers from that and the assignment of 4 minus x to y that y does not equal 0.

A property is *decidable* if there exists a computer algorithm that can determine whether or not the property holds. Depending on the programming language in which a piece of software is written, certain properties of the software are decidable, while others are not.

For example, *typed* languages have a *typing system* which assigns *types* to certain elements of a program (e.g. integer, floating point number, etc.) and makes rules about how certain typed elements can be used (e.g. only integer values should be assigned to integer variables). A program is *well-typed* when a type-assignment exists for each element of the program that should have a type, and when each of the typing rules is respected. Typed languages will ideally come with a *decidable type-checker*, which determines whether or not a program is *well-typed* with respect to the typing system. This helps software developers establish program structure and verify the validity of that structure.

On the other hand, it is well-known to be *undecidable*, for example, to verify that any arbitrary program in a Turing-complete language will always terminate when executed. For many proof systems, however, it *is* decidable to verify the validity of a proof that some *particular* program in a Turing-complete language will always terminate when executed. Such a proof serves as a certification for that particular program.

1.1.1 THE PROOFS-AS-PROGRAMS CORRESPONDENCE

Certified software often utilizes the *proofs-as-programs* correspondence [1]. This correspondence states that there is an isomorphism between mathematical proofs and computer programs. Proofs are programs and programs are proofs. One thinks of certified software as a program paired with a proof, but these two pieces are fundamentally the same type of computable/mathematical object. This correspondence is at the heart of how certified software works—how a proof can be mechanically verified because it is a program that can be analyzed and executed.

In our previous example, the program `main` is a proof that a floating point number can be computed from any integer; and the example proof is a program such that given any input

integer `i`, it outputs a proof that the execution of `main(i)` assigns a value to `y` that does not equal `0`.

Proofs and programs correspond further by the trust that one must put in the respective mechanical algorithms built around them. In the same way that a software developer puts her trust in a programming language's type-checking algorithm, for example, a *certified* software developer *also* puts her trust in a certified software system's proof-verification algorithm. Those certified software systems include the Nuprl Proof Development System, the Agda proof assistant, HOL theorem provers, the Isabelle proof assistant, the Coq Proof Assistant, and others. These systems are specifically designed to aid certified software developers in writing the certifying "proof parts" rather than the main-functionality "program parts" about which something is being proved. And those proof parts are often much larger and much more complex than the program parts of certified software.

1.1.2 THE COQ PROOF ASSISTANT

In this dissertation, we build certified software for designing asynchronous circuits using the Coq Proof Assistant [2]. To our knowledge, Coq has been used similarly only for synchronous circuits [3, 4, 5, 6]. A substantial number of other proofs and formalization works in both academia and industry have been built in Coq. It has a large and growing community of users, and the more people that use it, the more bugs that are found and fixed—the more it can be trusted. Coq has a rich typing system that corresponds to a wide range of things that can be proved about a program. Further, the problem of deciding whether or not a proof is valid is reduced to simply type checking it, which in the case of Coq is decidable. Another feature of Coq is that it allows for extracting programs to OCaml, which is a fast and practical executable language as compared to most proof assistants' internal computation.

CompCert [7] is one of the first large-scale practical pieces of certified software built in Coq. It is a certified compiler for the C programming language. CompCert translates high-level C programs into a handful of lower-level architecture-specific languages, and it provides Coq proofs that the observable behavior of any output program is equivalent to that of the input C program. This certification is made with respect to a common-ground interpretation of *observable behavior* among C and each of the lower-level languages. As such, a CompCert user must only trust that relatively straightforward interpretation (and Coq itself) rather than trust a C compiler's relatively complex translation algorithm.

A program called Csmith [8] was used to try to find errors in C compilers. Csmith generated small test-case C programs that might be compiled incorrectly. For two popular C compilers, GCC and LLVM, programs as small as 5 lines of code were found to be compiled incorrectly. For GCC, Csmith found 4 programs that were compiled incorrectly with all compiler optimizations disabled and 79 incorrectly compiled programs for GCC with optimizations. LLVM was found to miscompile 19 programs with optimizations disabled, and 202 programs with them enabled. On the other hand, there were only 2 bugs found in CompCert at the time the Csmith paper was being written. The authors let the CompCert developers know, and by the time the paper was published in 2011, no bugs were found in CompCert. "This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users" [8].

1.2 ASYNCHRONOUS CIRCUITS

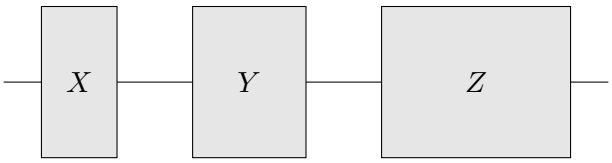
The certified software we build in this dissertation is used to provide trusted aid in designing asynchronous circuits. These electrical circuits differ from more commonly used syn-

chronous circuits by the way in which they coordinate communication between parts of the circuit. Asynchronous circuits use local message-passing channels rather than a global clock signal. A number of large-scale asynchronous circuits have been built, including floating-point units [9, 10], FPGAs [11], GPS processors [12], microprocessors [13], and others. However, we have yet to produce a fully comprehensive and effective set of automation tools for developing these and other asynchronous circuits at very-large-scale integration (VLSI) scale. These tools are especially challenging to build—in part because of the complexities involved with reasoning about the exponentially-many orderings of asynchronous circuit events.

1.2.1 VS. SYNCHRONOUS CIRCUITS

Synchronous circuits are very widely used. There has been a lot of infrastructure put in place for their design and development. Yet as computer systems continue to increase in complexity, the potential benefits of asynchronous circuits—including increased time-efficiency, power-efficiency, robustness, and cleaner abstraction—become more impactful.

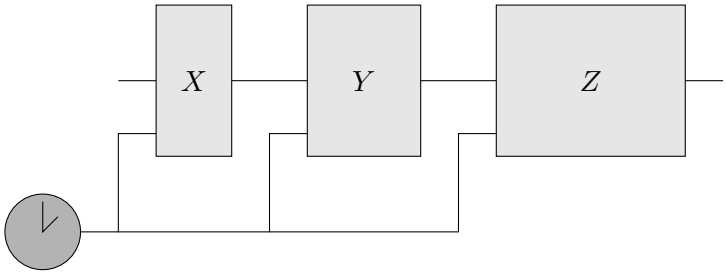
For example, consider the following circuit:



This circuit consists of a sequence of three components, X, Y and Z, each of which receives some incoming data signal, performs some operation on that data, then sends its output to the next part of the circuit. Suppose we want to pipeline data through the components such that each of them perform their operations on subsequent data signals in parallel. For example, X can process its next data signal while Y processes X's previous signal. This pipelining requires some coordination between the components in order to ensure that one data signal

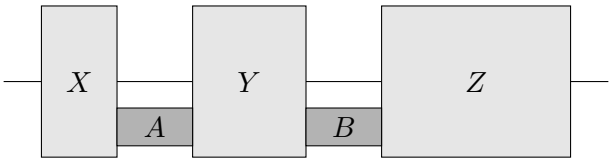
does not interfere with another. For example, component X should not communicate its signal to Y until Y is finished operating on its previous signal.

The synchronous-circuit solution to this needed coordination is to propagate a global clock signal to each of the three components as follows:



In this synchronous circuit, the clock periodically sends a global clock signal to components X, Y, and Z, which all communicate their respective data signals at the same time. Data signals do not interfere with one another because the clock period is set to be at least as long as the component that takes the *longest* time to perform its operation.

The asynchronous-circuit solution to the coordination needed for pipelining is to use a local message-passing channel between each pair of communicating components as follows:



In this asynchronous circuit, components X and Y communicate with each other over channel A, and components Y and Z communicate over channel B. Data signals do not interfere with one another because communication on each channel only takes place when both the sending component and the receiving component each indicate that they are ready.

TIME

A circuit component may take more or less time to perform its operation depending on the value of the data signal it is processing. Asynchronous circuits are usually as fast as the average case, while synchronous circuits are always as fast as the slowest case.

In the example above, suppose component X takes 1 second to perform its operation, Y takes 2 seconds, and Z takes 3 or 5 seconds, depending on the data. Further suppose that we know component Z to take 3 seconds 99% of the time and 5 seconds only 1% of the time. Finally, suppose the time overhead for each asynchronous channel is 0.5 seconds. Then, the average latency and throughput for each of the three example circuits is as follows:

pipelining	latency	throughput
none	6.12 s	0.163 Hz
synchronous	15 s	0.2 Hz
asynchronous	7.12 s	0.285 Hz

Without pipelining, we pass one data signal at a time through the original circuit with an average latency of $1+2+3(0.99)+5(0.01) = 6.12$ seconds and an average throughput of $1/6.12 = 0.163$ Hertz. In the synchronous circuit, the clock must be set to (at least) 5 seconds in order to provide enough time for component Z to perform its operation in the worst case. With this, we improve throughput to $1/5 = 0.2$ Hz, but we worsen latency to $5 + 5 + 5 = 15$ s since each component must wait the full 5-second clock period before communicating its data. In the asynchronous circuit, we improve the original average throughput to $0.99/3.5 + 0.01/5.5 = 0.285$ Hz, and we worsen the average latency to only $1 + 0.5 + 2 + 0.5 + 3(0.99) + 5(0.01) = 7.12$ s.

The above is an is idealistic example, but it demonstrates the real timing improvements

observed in asynchronous circuits. For example, an asynchronous floating-point multiplier (FPM) [9] was found to compare to a synchronous FPM as follows:

FPM	latency	throughput
synchronous	701 ps	666 MHz
asynchronous	705 ps	1.53 GHz

The table above shows the asynchronous FPM to have a latency less than 1% slower than that of the synchronous FPM, yet a throughput more than double that of the synchronous. The asynchronous FPM was further able to take advantage of floating-point operations where one or both of the operands is zero. In these cases, the synchronous FPM latency stayed the same, while that of the asynchronous was only 464 ps.

POWER

While using a global clock to coordinate circuit components eliminates the time overhead introduced by local message-passing channels, it introduces much more power overhead. Global clocks must generate a signal that is strong enough to propagate to all necessary circuit components. Further, clocks are typically *active* by default—they send their global signal to all components, regardless of which of them are utilizing the signal in a particular cycle. As such, in some cases the clock of a synchronous circuit might account for up to one third of the power usage. This has led to the adoption of *clock gating*, a technique in which the clock is dynamically turned on and off depending on whether or not it is needed.

On the other hand, asynchronous circuits' communication channels consume power comparable to that of the rest of the circuit. And unlike global clocks, the local channels are typically *inactive* be default—they only consume energy when they are being used. In the case of

the asynchronous FPM [9], it was found to compare to the synchronous FPM as follows:

FPM	energy/op
synchronous	280.8 pJ
asynchronous	92.1 pJ

The table above shows that the asynchronous FPM uses one third of the energy per operation than the synchronous FPM. And as was the case with latency, the asynchronous FPM consumed less power when performing floating-point operations where one or both of the operands is zero. In these cases, the synchronous FPM power consumption stayed the same, while that of the asynchronous was only 15.8 pJ.

ROBUSTNESS

In addition to time and power advantages, asynchronous circuits are also generally more robust than synchronous circuits. This is because clocks are only functional at limited temperatures and voltages. Asynchronous-circuit channels, on the other hand, can generally function within the same conditions as the rest of the circuit.

ABSTRACTION

Much of the effort that goes into designing synchronous circuits is actually spent on making sure each component completes its operation within some fixed clock period. Time is a prominent element of synchronous circuit design.

In asynchronous circuit design, time is instead abstracted away as a lower-level implementation detail. This abstraction enables one to disregard the precise timing of circuit elements when constructing a high-level design, which is especially useful for large-scale circuits. It

also allows for certain low-level timing improvements to be made without affecting the high-level behavior of a circuit. Asynchronous circuit designs instead concern the *ordering* of certain circuit events. And these event orderings, of which there are exponentially many, present a fundamental complexity challenge for asynchronous-circuit design.

1.2.2 MARTIN SYNTHESIS

A process known as *Martin synthesis* is commonly used to develop asynchronous circuits [14, 15]. The process begins with a program written in the Communicating Hardware Processes (CHP) language [16]. This CHP program specifies the high-level behavior of an asynchronous circuit. The program is then iteratively translated into lower- and lower-level representations until a circuit blueprint is reached. Automatic asynchronous circuit synthesis algorithms do in fact exist, for CHP [17] and for other languages [18], but the circuits they produce fall short compared to handcrafted blueprints on a large scale. The process of getting from a CHP program to an asynchronous-circuit blueprint is relatively manual, and Martin synthesis outlines the best practices for doing so.

For asynchronous circuits at VLSI scale, the greatest performance improvements are achieved by design decisions made at the CHP level, where the general structure of an asynchronous circuit is largely determined. Martin synthesis typically begins with a fully sequential CHP program, i.e. a program that describes things happening in a certain order with no two things happening at the same time. CHP-to-CHP transformations are then used to improve a circuit's performance by adding parallelism/concurrency. These transformations are currently applied *by hand*, despite requiring complex reasoning about synchronization that is critical to the performance and correctness of a circuit implementation. An evolving CHP program must be verified not to exhibit erroneous behavior (e.g. two things happening at the

same time that conflict with each other) or any behavior that is not some valid parallelization of the original sequential program.

1.3 OVERVIEW & RELATED WORK

In this dissertation, we develop and mechanically formalize the syntax and semantics of CHP and build a certified CHP simulator, all within the Coq Proof Assistant. The *syntax* of a language refers to its structure, and the *semantics* of a language refers to its meaning/behavior. We formalize a high-level semantics for CHP in which the behavior of a CHP program is defined as a set of event traces. Our semantics formalization is novel in the combination of its mechanical formalization, its comprehensive expression of CHP, and its precise abstraction of lower-level design decisions. Our CHP simulator functions according to our newly developed semantics by first translating a CHP program into a labeled transition system (LTS)—namely a delimited labeled safe Petri net. The Petri net is then simulated, and we prove that the traces it produces are exactly those defined by the semantics of the original CHP program.

Construction and Analysis of Distributed Processes (CADP) [19] is a collection of software tools used for designing certain systems. CHP has been translated into a format based on communicating state machines [20] and into the LOTOS standard process calculus [21], both for program property-checking using CADP. These translations formalize CHP semantics as an LTS that disallows shared variables across concurrent processes, which our formalization allows. Further, the formalization is not mechanically verified as ours is.

In our previous work, we developed an LTS formalization of CHP semantics and used it to build a CHP error-checker [22]. We used Coq to prove the LTS semantics to be equivalent to a more commonly accepted (non-labeled) state-transition-system semantics [23]. Our semantics in that work excludes data-communicating synchronization, which our formaliza-

tion here allows. Our previous formalization also assumes active-sender/passive-receiver synchronization behavior in which a sender process always initiates channel communication. Our new formalization does not make such an assumption, thus precisely abstracting away the lower-level circuit design decision of how channel synchronization is implemented.

A comprehensive formalization of the semantics used at every stage of Martin synthesis has been developed [24]. The CHP-level trace semantics in that work are most similar to ours, but they and the proofs about them are not defined mechanically. Further, while that work translates a language-independent program into a possibly-infinite tree, we translate a CHP program into an actually-constructible finite Petri net which we can then mechanically simulate.

CHP programs have also been interpreted as action systems for the verification of transformations in that domain [25]. The proofs are formal, but not machine-checked. Further, the action-system interpretation itself is made informally. We provide a precise, formal relationship between CHP syntax and semantics. And our more direct reasoning about CHP is more conducive to integrating with other elements of asynchronous-circuit development using Martin synthesis.

Our mechanical formalization of event orderings within a trace is similar to the Logic of Events developed in NuPRL [26]. This logic targets distributed systems in which entities without shared state communicate asynchronously by causally-ordered send and receive events. This differs from our CHP semantics in that our communicating processes *do* have shared state. Further, our traces include simultaneous events.

CHPsim [27] is a CHP simulator that "cosimulates" a system in which different pieces of an asynchronous circuit are at different notation levels of Martin synthesis. However, that cosimulation is not verified in any way. While our certified simulator only simulates programs

at the CHP level, it is verified to do so correctly.

Our construction of Petri nets from the CHP language is similar to that which was done for Concurrent Kleene Algebra [28]. That work constructs Petri nets from series-rational (sr) expressions and shows that every sr-language is the pomset trace language of a safe labeled Petri net. An sr-language is similar to CHP, but does not express infinite traces or simultaneous events as CHP does.

In summary, the CHP semantics we develop differs from that of prior works due to its mechanization, comprehensiveness, and precision. Our semantics formalization is built mechanically within the Coq Proof Assistant framework, it embraces the challenging features of CHP including shared data variables across concurrent processes and data-communicating synchronization, and it precisely abstracts away design decisions made at lower levels of Martin synthesis, e.g. the implementation of channel synchronization. The CHP simulator we build differs from that of prior works in that it is certified to simulate a CHP program correctly with respect to the language semantics. Our simulator further serves as a model for future Coq-built certified CHP software tools.

2 COMMUNICATING HARDWARE PROCESSES (CHP)

Communicating Hardware Processes (CHP) [16] is a programming language in which high-level asynchronous circuit designs are written. CHP is a variant of Communicating Sequential Processes (CSP) [29] with modifications made in order to better express the actual physical behavior of asynchronous circuits. Those modifications include guarded commands [30], an additional *probe* construct, and erroneous behavior that corresponds to physical circuit errors (e.g. short circuits).

2.1 BASICS

We preface our presentation of CHP syntax with that of some basic building blocks of the language, including values, data variables, data expressions, channels, guards, and probes.

A *value* is a finite piece of information. A *digital* circuit ultimately concerns values in the form of finite sequences of Booleans. At the CHP level, however, we can concern ourselves with higher-level values like numbers, letters, or words. These pieces of information must ultimately be expressed in some Boolean representation, but that representation is abstracted away in CHP.

A *sequential* digital circuit includes state-holding elements. These are pieces of a circuit that can remember values, as opposed to just having values flow through them like a wire.

CHP has *data variables* which correspond to these elements. A data variable stores a piece of information; and just as CHP can be formalized with different high-level values, CHP data variables can store those high-level values. In this case, a data variable may correspond to a *collection* of state-holding circuit elements.

CHP also includes high-level *data expressions*. A data expression can be thought of as a value that can depend upon the values stored in some data variables at the time the expression is used. For example, the expression $x+1$ will be evaluated to whatever number is stored in data variable x incremented by 1. Expressions in CHP correspond to logic blocks on the circuit whose output can be computed as a pure function. The evaluation of an expression corresponds to signals flowing through those blocks. The data variables present in an expression correspond to the input to those blocks, and the resulting value of the expression corresponds to the output of the blocks. For example, the logic block for the expression $x+1$ would have an input that corresponds to the value stored in x, and its output corresponds to the result of adding 1 to that input.

The main concept of an asynchronous circuit is that concurrently executing pieces of the circuit synchronize and communicate with each other via local message passing. At the CHP level, this local message passing is modeled by *channels* and the actions performed on them. Channels correspond to wires connecting two pieces of a circuit, and the channel actions correspond to the communication protocol used across those wires. As with the Boolean representation of values, at the CHP level we need not specify the precise communication protocol used across a channel. Instead, we model the communication by two different types of high-level channel actions that a CHP process can perform: send and receive. When two processes synchronize together on a channel, one of them is the sender and the other is the receiver. As their names imply, these channel actions are not only used to coordinate

two concurrent processes, but also for those processes to exchange information. The sender process includes an expression that it evaluates and then sends along the channel; and the receiver process includes a variable into which it stores its received value.

A *guard* is a special expression that must evaluate to a Boolean. We will see that while data expressions are used to store a value into a data variable or to send a value on a channel, guards are used as decision factors for a process determining what to do next. For example, if the guard evaluates to `true`, the process does *this*, and if it evaluates to `false`, the process does *that*—hence the requirement that a guard must evaluate to a Boolean.

Probes, which can only appear in guards (not data expressions), contain values that indicate the intermediate steps of a channel exchange. Given a channel A, the *sender probe*, denoted $\overline{A}$, is a Boolean value indicating a sender is ready on channel A and waiting for a receiver; the *receiver probe*, denoted $\widehat{A}$, is a Boolean indicating a receiver is ready for a sender; and the *data probe*, denoted $A\#$, provides the value pushed on channel A by a sender waiting for a receiver. The data probe effectively allows a process to peek at the value being sent on a channel without actually receiving it.

Finally, we use the term *mutable* to refer to an element in the collective set of both data variables and probes.

2.2 CORE SYNTAX & INFORMAL SEMANTICS

Figure 2.1 displays the core syntax of CHP.

We restrict the type of values, data variables, data expressions, and channels to the natural numbers $\mathbb{N}$. This simplifies our formalization of CHP semantics without taking away from the main challenges of the language. We leave the formalization of a CHP type system to future work. Such a system would, for example, assign types to data variables and channels, and

$$
\begin{array}{rcl}
\text{value} & k & ::= 0 \mid 1 \mid 2 \mid \ldots \\
\text{data variable} & x, y, z, w & \\
\text{data expression} & e & ::= k \mid x \mid \ldots \\
\text{channel} & A, B & \\
 & & \\
\text{sender probe} & \overline{A}, \overline{B} & \\
\text{receiver probe} & \widehat{A}, \widehat{B} & \\
\text{data probe} & A\#, B\# & \\
\text{mutable} & X & ::= x \mid \overline{A} \mid \widehat{A} \mid A\# \\
 & & \\
\text{Boolean} & b & ::= \texttt{true} \mid \texttt{false} \\
\text{guard} & G & ::= b \mid \overline{A} \mid \widehat{A} \mid A\# = e \mid e_1 = e_2 \\
 & & \quad \mid \neg G \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \\
 & & \\
\text{communication} & C & ::= A!(e) \\
 & & \quad \mid A?(x) \\
 & & \quad \mid C_1 \bullet C_2 \\
 & & \\
\text{program} & P & ::= \texttt{skip} \\
 & & \quad \mid x := e \\
 & & \quad \mid C \\
 & & \quad \mid P_1 \,;\, P_2 \\
 & & \quad \mid P_1 \parallel P_2 \\
 & & \quad \mid [G_1 \to P_1 \mid G_2 \to P_2] \\
 & & \quad \mid [G_1 \to P_1 \,[\!]\, G_2 \to P_2] \\
 & & \quad \mid *[G \to P]
\end{array}
$$

Figure 2.1: CHP syntax

reject programs that try to assign to them an expression of a different type.

We restrict the syntactic structure of guards in order to more easily reason about probes separate from data expressions. For example, the grammar disallows the guard $f(A\#)$ for some arbitrary expression function f. This restriction is not significantly limiting, as most if not all guards in CHP programs we have encountered already adhere to this structure. Also, since all data expressions are restricted to natural numbers, we avoid the case where a program evaluates the guard $e_1 = e_2$ when e_1 and e_2 evaluate to values of two different types.

2.2.1 PROGRAMS

As shown in Figure 2.1, a *program* is a top-level CHP process, which either:

- does nothing (`skip`),
- performs some data-variable assignment (e.g. $x := x + 1$),
- performs some channel communication (e.g. $A!(4), B?(y), A!(4) \bullet B?(y)$),
- executes two programs in sequence ($P_1 \,;\, P_2$),
- executes two programs in parallel ($P_1 \parallel P_2$),
- selects one of two guarded programs ($[G_1 \to P_1 \mid G_2 \to P_2]$ or $[G_1 \to P_1 \,[]\, G_2 \to P_2]$), or
- repeats a guarded program ($*[G \to P]$).

A *data-variable assignment* consists of evaluating some data expression then storing the resulting value into a data variable. For example, the assignment $x := x + 1$ evaluates the expression $x + 1$ then stores its value into x.

A *communication* is a CHP process that either performs a channel send (e.g. $A!(4)$), a channel receive (e.g. $B?(y)$), or two communications that are forced to complete together (e.g. $A!(4) \bullet B?(y)$), which we exemplify below. Communications are blocking processes, meaning a send on some channel A must proceed together with a concurrent receive on channel A, and vice versa. For example, in the following program:

$$(x := 6 \,;\, A!(7) \,;\, y := x) \parallel A?(x)$$

data variable y is always assigned the value 7 (never 6). Similarly, in the following program:

$$(w := 4 \,;\, x := 6 \,;\, A!(7) \,;\, y := x \,;\, z := w) \parallel (A?(x) \bullet B!(5)) \parallel B?(w)$$

data variable y is always assigned the value 7 (never 6) and z is always assigned 5 (never 4).

A *guarded program* is a program that is preceded by a guard, which must evaluate to `true` before the program is executed. Guarded programs are used to construct selection and repetition programs.

A *selection program* blocks until one of its guards evaluates to `true`, then executes the respective subprogram. For example, in the following program:

$$A!(3) \parallel ([A\# = 3 \rightarrow x := 30 \mid A\# = 4 \rightarrow x := 40]\,;\, A?(y))$$

x is assigned 30 (never 40). A selection program is declared to be deterministic (denoted with []s) or nondeterministic (denoted with |s). A *nondeterministic-selection program* allows for multiple `true` guards, in which case one of the respective subprograms is arbitrarily selected for execution. For example, in the following program:

$$A?(x) \parallel B?(y) \parallel [\widehat{A} \rightarrow A!(3)\,;\, B!(3) \mid \widehat{B} \rightarrow A!(4)\,;\, B!(4)]$$

data variables x and y are either both assigned 3 or both assigned 4. On the other hand, a *deterministic-selection program* declares that when it is executed, it should never be the case that more than one of its guards is `true`. For example, the following program:

$$A?(x) \parallel B?(y) \parallel [\widehat{A} \rightarrow A!(3)\,;\, B!(3) \,[]\, \widehat{B} \rightarrow A!(4)\,;\, B!(4)]$$

exhibits erroneous behavior because it is possible for both $\widehat{A}$ and $\widehat{B}$ to be `true` upon execution of the selection. The circuit implementation of a deterministic selection statement will assume such a case is never possible, and might exhibit unwanted hazardous behavior if the case is encountered.

The reason why we distinguish between deterministic and nondeterministic selection is because their respective circuit implementations perform very differently. Nondeterministic selection is more expensive to implement on a circuit and impossible to implement without *metastability* issues [31]. Metastability refers to a circuit's inability to properly react to its inputs in bounded time. Such a circuit instead persists in an unstable equilibrium, or metastable state, where one or more of its components is unable to reach a stable Boolean logical value. This issue arises when the implementation of a nondeterministic selection must decide which subprogram to execute when either of their guards may become `true` at any time. Instead, if a circuit designer can guarantee that no two guards of a CHP selection statement are both `true` at the time of execution, metastability can be avoided and a cheaper circuit can be used to implement the program.

A *repetition program* is similar to a selection but has only one guarded program and is nonblocking. The repetition program first evaluates its guard. If the guard is `true`, the subprogram is executed then the repetition program repeats. If the guard is `false`, the repetition program terminates.

2.2.2 SHARED DATA VARIABLES

Channels of course must be shared across concurrent processes because that is the only way in which they can properly function. However, certain formalizations of CHP are often restricted by disallowing shared data variables across concurrent processes. Such shared data variables exhibit the inherent challenges of concurrent programming, including the usual checks for conflicting read/write and write/write data-variable errors, and the check for unstable guards (i.e. when an initially true guard becomes false during its evaluation). These situations can translate to hazardous circuit behavior, like short circuits, glitches, and faulty

metastability. We do not shy away from dealing with this erroneous behavior so as not to limit the possible circuits that can implement a CHP design. We allow and embrace shared data variables in our formalization of CHP, and we comprehensively express how they are used safely.

2.2.3 DATA-COMMUNICATING SYNCHRONIZATION

We also allow and embrace data-communicating synchronization, meaning information is sent and received during a channel exchange. Some formalizations of CHP are restricted to only allow dataless synchronization across channels. Again, we do not impose such a restriction so as not to limit the circuit implementations that can be made of a CHP design.

2.2.4 VS. FULL SYNTAX

The syntax presented above excludes a few language features that are often included in what we call "full" CHP. Those features include multi-assignment, dataless and multi-data communication, and n-ary selection and repetition. The exclusion of these features helps to simplify the implementation and certification of our CHP tools, but does not take away from the expressive power of the language. We detail these three features below, and show how each of them can be emulated by the core CHP syntax.

MULTI-ASSIGNMENT

A multi-assignment process performs multiple data-variable assignments together. For example, the process $x, y := x + 1, y + 1$ first evaluates the expressions $x + 1$ and $y + 1$ in parallel, then assigns their respective resulting values to x and y in parallel. Given any multi-assignment process $x_1, \ldots, x_n := e_1, \ldots, e_n$, we can emulate its behavior using only single-

assignment processes with the following program:

$$(x'_1 := e_1 \parallel \cdots \parallel x'_n := e_n)\,;\,(x_1 := x'_1 \parallel \cdots \parallel x_n := x'_n)$$

where we introduce variables $x'_1, \ldots, x'_n$ that are not used elsewhere in the program.

DATALESS AND MULTI-DATA COMMUNICATION

A dataless communication process synchronizes two concurrent processes without performing a data exchange, e.g. $A!() \parallel A?()$. We can emulate dataless send $A!()$ and receive $A?()$ using $A!(0)$ and $A?(x_0)$, respectively, where we introduce a variable x_0 that is not used elsewhere in the program.

A multi-data communication process exchanges multiple pieces of information when synchronizing two concurrent processes, e.g. $A!(3,4) \parallel A?(x,y)$. Given any multi-data communication processes $A!(e_1, \ldots, e_n)$ and $A?(x_1, \ldots, x_n)$, we can emulate their respective behavior using proesses $A_1!(e_1) \bullet \cdots \bullet A_n!(e_n)$ and $A_1?(x_1) \bullet \cdots \bullet A_n?(x_n)$.

N-ARY SELECTION AND REPETITION

In full CHP syntax, selection and repetition programs may consist of one or more guarded programs (and repetition programs are similarly deterministic or nondeterministic). In the case of one guarded selection program, e.g. $[G \to P]$, we can simply add a `false`-guarded `skip` as the second guarded program, i.e. $[G \to P \,[]\, \texttt{false} \to \texttt{skip}]$. In the case of many-guarded selection or repetition programs, e.g. $*[G_1 \to P_1 \mid \cdots \mid G_n \to P_n]$, we can fold the program into nested selection programs as follows:

$$
\begin{array}{l}
*[\, G_1 \vee \ldots \vee G_n \rightarrow [\, G_1 \rightarrow P_1 \\
\qquad\qquad\qquad\qquad\quad |\, G_2 \vee \ldots \vee G_n \rightarrow [\, G_2 \rightarrow P_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad |\, G_3 \vee \ldots \vee G_n \rightarrow \cdots [\, G_{n-1} \rightarrow P_{n-1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad |\, G_n \rightarrow P_n \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad] \cdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad] \\
\qquad\qquad\qquad\qquad\quad] \\
\quad]
\end{array}
$$

2.3 FORMAL SEMANTICS

A CHP program specifies the intended behavior of a circuit. A circuit's true behavior is inherently dependent upon physical properties, including the time it takes for a signal to propagate through various circuit elements. When modeling asynchronous circuits, however, these delays are mostly ignored; time is abstracted away as a low-level implementation detail that does not affect the *functional* correctness of a design. This abstraction is especially useful for reasoning about the behavior of a complex VLSI system at a high level in which its exact timing behavior is not yet known.

Instead of having to deal with real-time measurements, the semantics of CHP needs only capture the ordering of some circuit events that can be considered instantaneous. Our semantics models these events by the elements that make up an execution trace of the CHP program. We preface our presentation of events and traces with that of states, which we use to formalize the behavior of individual events.

2.3.1 STATES

Let ς be something distinct from all values, which we will use to indicate when a variable in the system is uninitialized. Then, a *state* is a function from data variables and data probes to

a value or ς, and from sender and receiver probes to a Boolean. States are used to model the values assigned to mutables at some point during program execution.

The $zero$ state, denoted σ_0, maps all data variables and data probes to ς, and maps all sender and receiver probes to `false`. This is the starting state used for CHP program execution. Given a state σ, a mutable X, and a {value, ς, or Boolean} K, $\sigma[X \mapsto K]$ denotes the state that results from updating σ to map X to K.

We use $\sigma(X)$ to denote the value or ς to which state σ maps mutable X. We extend this notation to data expressions and guards. $\sigma(e)$ denotes the value or ς to which state σ evaluates expression e, defined in the usual manner such that $\sigma(e) = \varsigma$ iff $\sigma(x) = \varsigma$ for some data variable x appearing in e. $\sigma(G)$ denotes the Boolean or ς to which state σ evaluates guard G as follows:

$$\sigma(A\# = e) \triangleq \begin{cases} \texttt{true}, & \text{if } \exists k, \sigma(A\#) = k = \sigma(e) \\ \texttt{false}, & \text{if } \exists k, \sigma(A\#) \neq k = \sigma(e) \\ \varsigma, & \text{otherwise} \end{cases}$$

$$\sigma(e_1 = e_2) \triangleq \begin{cases} \texttt{true}, & \text{if } \exists k, \sigma(e_1) = k = \sigma(e_2) \\ \texttt{false}, & \text{if } \exists k_1, k_2,\ \sigma(e_1) = k_1 \neq k_2 = \sigma(e_2) \\ \varsigma, & \text{otherwise} \end{cases}$$

$$\sigma(\neg G) \triangleq \begin{cases} \neg b, & \text{if } \sigma(G) = b \\ \varsigma, & \text{otherwise} \end{cases}$$

$$\sigma(G_1 \wedge G_2) \triangleq \begin{cases} b_1 \wedge b_2, & \text{if } \sigma(G_1) = b_1 \text{ and } \sigma(G_2) = b_2 \\ \varsigma, & \text{otherwise} \end{cases}$$

$$\sigma(G_1 \vee G_2) \triangleq \begin{cases} b_1 \vee b_2, & \text{if } \sigma(G_1) = b_1 \text{ and } \sigma(G_2) = b_2 \\ \varsigma, & \text{otherwise} \end{cases}$$

Note that $\sigma(A\# = e)$ and $\sigma(e_1 = e_2)$ evaluate to a Boolean iff each data expression evaluates

to a value. For example:

$$\sigma_0[A\# \mapsto 3](A\# = 3) = \texttt{true} \qquad \sigma_0[x \mapsto 3](A\# = x) = \texttt{false} \qquad \sigma_0[A\# \mapsto 3](A\# = x) = \varsigma$$

$$\sigma_0(0 = 0) = \texttt{true} \qquad \sigma_0[x \mapsto 0](x = 1) = \texttt{false} \qquad \sigma_0[x \mapsto 3](x = y) = \varsigma \qquad \sigma_0(x = x) = \varsigma$$

In the case where $\sigma(A\#) = \varsigma$ and $\sigma(e) = k$, the guard $A\# = e$ evaluates to `false`. This is because CHP program execution may validly encounter a channel with no value on it (yet), whereas it is erroneous to encounter a data expression with uninitialized data variables (the case where $\sigma(e) = \varsigma$).

Also note that $\sigma(\neg G)$, $\sigma(G_1 \wedge G_2)$ and $\sigma(G_1 \vee G_2)$ each evaluate to a Boolean iff each of their subguards evaluates to a Boolean. We take this non-short-circuiting approach to guard evaluation in order to conservatively identify errors, e.g. a CHP program execution is erroneous if it encounters a guard with any non-Boolean subguard (regardless of other subguards). For example:

$$\sigma_0(\neg\, x = 0) = \varsigma \qquad \sigma_0(\texttt{true} \vee x = 1) = \varsigma \qquad \sigma_0(\texttt{false} \wedge A\# = x) = \varsigma$$

2.3.2 EVENTS

Events are the indivisible behavioral elements that make up a CHP program-execution trace. An event is either an *assignment* $x{\leftarrow}e$, a *send-up* $A!{\uparrow}e$, a *send-down* $A!{\downarrow}$, a *receive-up* $A?{\uparrow}$, a *receive-down* $A?{\downarrow}x$, a *wait* $[G]$, or a *determinism-violation* `detv`:

$$
\begin{aligned}
\text{event } E ::= {} & x{\leftarrow}e \\
& \mid A!{\uparrow}e \mid A!{\downarrow} \\
& \mid A?{\uparrow} \mid A?{\downarrow}x \\
& \mid [G] \mid \texttt{detv}
\end{aligned}
$$

Given a state σ and event E, the *update* of σ using E, denoted $\sigma[E]$, is the state that results after applying E to σ:

$$
\begin{aligned}
\sigma[x{\leftarrow}e] &\triangleq \sigma[x \mapsto \sigma(e)] \\
\sigma[A!{\uparrow}e] &\triangleq \sigma[A\# \mapsto \sigma(e)][\overline{A} \mapsto \texttt{true}] \\
\sigma[A!{\downarrow}] &\triangleq \sigma[\overline{A} \mapsto \texttt{false}] \\
\sigma[A?{\uparrow}] &\triangleq \sigma[\widehat{A} \mapsto \texttt{true}] \\
\sigma[A?{\downarrow}x] &\triangleq \sigma[x \mapsto \sigma(A\#)][A\# \mapsto \varsigma][\widehat{A} \mapsto \texttt{false}] \\
\sigma[[G]] &\triangleq \sigma \\
\sigma[\texttt{detv}] &\triangleq \sigma
\end{aligned}
$$

The assignment event $x{\leftarrow}e$ models the behavior of a data-variable assignment process $x := e$.

The behavior of a send process $A!(e)$ is modeled by the send-up event $A!{\uparrow}e$ followed by the send-down event $A!{\downarrow}$. Similarly, the behavior of a receiver process $A?(x)$ is modeled by the receive-up event $A?{\uparrow}$ followed by the receive-down event $A?{\downarrow}x$. These channel events model the handshake that occurs between two communicating parts of an asynchronous circuit. Lower-level circuit design decisions will further implement the behavior of this handshake, i.e. a sender might actively initiate the handshake while a receiver passively waits, and vice versa. We will see that our semantics formalization constitutes a precise abstraction of such design decisions, where a channel handshake can be initiated by a send-up *or* receive-up event (or both), so long as it completes with the send-down and receive-down events syn-

chronized together.

The wait event $[G]$ models the evaluation of guard G to `true`, as is done at the start of the execution of guarded processes that make up selection and repetition programs.

The determinism-violation event `detv` is used to model the erroneous case when both guards of a deterministic-selection statement evaluate to `true` at the same time.

Finally, note that in the cases where a state evaluation equals ς, the state is validly updated to set the corresponding data variable or data probe to ς. We use this to maintain a simple definition for updating a state. However, we do not expect such updates to actually be executed. We will see below that they are considered erroneous.

EVENT PROPERTIES

Given a state σ, an event E is *uninitialized*, denoted $\mathit{uninit}_\sigma(E)$, when its action assigns ς to a data variable or data probe:

$$\frac{\sigma(e) = \varsigma}{\mathit{uninit}_\sigma(x{\leftarrow}e)} \qquad \frac{\sigma(e) = \varsigma}{\mathit{uninit}_\sigma(A!{\uparrow}e)} \qquad \frac{\sigma(A\#) = \varsigma}{\mathit{uninit}_\sigma(A?{\downarrow}x)}$$

Given a mutable X and event E, the property $\mathit{reads}_X(E)$ holds when E reads X, and the property $\mathit{writes}_X(E)$ holds when E writes to X:

$$\frac{x \in e}{\mathit{reads}_x(y{\leftarrow}e)} \qquad \frac{x \in e}{\mathit{reads}_x(A!{\uparrow}e)} \qquad \frac{}{\mathit{reads}_{A\#}(A?{\downarrow}x)}$$

$$\frac{}{\mathit{writes}_x(x{\leftarrow}e)} \qquad \frac{}{\mathit{writes}_{A\#}(A!{\uparrow}e)} \qquad \frac{}{\mathit{writes}_{\overline{A}}(A!{\uparrow}e)} \qquad \frac{}{\mathit{writes}_{\overline{A}}(A!{\downarrow})}$$

$$\frac{}{\mathit{writes}_{\widehat{A}}(A?{\uparrow})} \qquad \frac{}{\mathit{writes}_x(A?{\downarrow}x)} \qquad \frac{}{\mathit{writes}_{A\#}(A?{\downarrow}x)} \qquad \frac{}{\mathit{writes}_{\widehat{A}}(A?{\downarrow}x)}$$

Note that it is *not* the case that $\mathit{reads}_X([G])$ holds for wait events with guard G such that $X \in G$.

This is because the $\mathit{reads}_X(E)$ property is used to express a potential read/write error, and we will see that wait events do not contribute to read/write errors but to unstable-guard errors, where the true guard G of a wait event $[G]$ becomes false during its evaluation.

Two events E_1 and E_2 are *conflicting*, denoted $\mathit{conflict}(E_1, E_2)$, when the two exhibit a read/write error or a write/write error:

$$\frac{\mathit{reads}_X(E_1) \quad \mathit{writes}_X(E_2)}{\mathit{conflict}(E_1, E_2)} \qquad \frac{\mathit{writes}_X(E_1) \quad \mathit{reads}_X(E_2)}{\mathit{conflict}(E_1, E_2)} \qquad \frac{\mathit{writes}_X(E_1) \quad \mathit{writes}_X(E_2)}{\mathit{conflict}(E_1, E_2)}$$

FACT. *Given two non-conflicting events E_1 and E_2, either order in which they can be applied to state σ results in the same final state:*

$$\neg \mathit{conflict}(E_1, E_2) \implies \sigma[E_1][E_2] = \sigma[E_2][E_1]$$

BAGS OF EVENTS

We will see that CHP program-execution traces are composed of finite bags of events, each modeling the execution of those events in any (possibly-overlapping) order. We use $\langle$ and $\rangle$ to delimit event-bags, $\uplus$ to denote bag-addition and $\sqsubseteq$ to denote bag-inclusion.

An event-bag β is *synchronized*, denoted $\mathit{sync}(\beta)$, when it has both or neither send-down and receive-down events for each channel:

$$\mathit{sync}(\beta) \triangleq \forall A,\ A!{\downarrow} \in \beta \iff \exists x, A?{\downarrow}x \in \beta$$

For example:

$$\mathit{sync}(\langle\rangle) \qquad \mathit{sync}(\langle A!{\downarrow}, A?{\downarrow}x\rangle) \qquad \mathit{sync}(\langle A!{\downarrow}, A?{\downarrow}x, A?{\downarrow}y\rangle) \qquad \mathit{sync}(\langle A!{\downarrow}, A?{\downarrow}x, B!{\downarrow}, B?{\downarrow}y\rangle)$$

$$\neg \mathit{sync}(\langle A!\downarrow \rangle) \qquad \neg \mathit{sync}(\langle A?\downarrow x \rangle) \qquad \neg \mathit{sync}(\langle A!\downarrow, A?\downarrow x, B!\downarrow \rangle)$$

An event-bag β is *satisfied* by state σ, denoted $\mathit{sat}_\sigma(\beta)$, when $\sigma(G) = \texttt{true}$ for the guard G of each wait event in β:

$$\mathit{sat}_\sigma(\beta) \triangleq \forall [G] \in \beta, \sigma(G) = \texttt{true}$$

For example:

$$\mathit{sat}_{\sigma_0}(\langle\rangle) \qquad \mathit{sat}_{\sigma_0}(\langle [\texttt{true}] \rangle) \qquad \mathit{sat}_{\sigma_0[x \mapsto 2]}(\langle [x = 2] \rangle) \qquad \mathit{sat}_{\sigma_0[\overline{A} \mapsto \texttt{false}]}(\langle [\neg \overline{A}] \rangle)$$

$$\neg \mathit{sat}_{\sigma_0}(\langle [\texttt{false}] \rangle) \qquad \neg \mathit{sat}_{\sigma_0}(\langle [x = x] \rangle) \qquad \neg \mathit{sat}_{\sigma_0}(\langle [\overline{A}] \rangle)$$

Given a state σ, an event-bag β is *feasible*, denoted $\mathit{feas}_\sigma(\beta)$, when it is both synchronized and satisfied by σ:

$$\mathit{feas}_\sigma(\beta) \triangleq \mathit{sync}(\beta) \wedge \mathit{sat}_\sigma(\beta)$$

For example:

$$\mathit{feas}_{\sigma_0}(\langle\rangle) \qquad \mathit{feas}_{\sigma_0}(\langle A!\downarrow, A?\downarrow \rangle) \qquad \mathit{feas}_{\sigma_0}(\langle [\texttt{true}] \rangle) \qquad \mathit{feas}_{\sigma_0}(\langle A!\downarrow, A?\downarrow, [\texttt{true}] \rangle)$$

$$\neg \mathit{feas}_{\sigma_0}(\langle A!\downarrow, A?\downarrow, [\texttt{false}] \rangle) \qquad \neg \mathit{feas}_{\sigma_0}(\langle A!\downarrow, [\texttt{true}] \rangle) \qquad \neg \mathit{feas}_{\sigma_0}(\langle A?\downarrow, [\texttt{false}] \rangle)$$

An event-bag β is *invalid* in state σ, denoted $\mathit{inval}_\sigma(\beta)$, when $\sigma(G) = \varsigma$ for the guard G of some wait event in β:

$$\mathit{inval}_\sigma(\beta) \triangleq \exists [G] \in \beta, \sigma(G) = \varsigma$$

For example:

$$\mathit{inval}_{\sigma_0}(\langle[x = x]\rangle) \qquad \mathit{inval}_{\sigma_0}(\langle[\texttt{true}], [x = x]\rangle) \qquad \mathit{inval}_{\sigma_0}(\langle[\texttt{false}], [A\# = x]\rangle)$$

We extend the notion (and notation) of uninitialization to a bag of events. Given a state σ and event-bag β, β is uninitialized, denoted $\mathit{uninit}_\sigma(\beta)$, when it contains an uninitialized event:

$$\mathit{uninit}_\sigma(\beta) \triangleq \exists E \in \beta, \mathit{uninit}_\sigma(E)$$

For example:

$$\mathit{uninit}_{\sigma_0[x\mapsto 1]}(\langle x{\leftarrow}y\rangle) \qquad \mathit{uninit}_{\sigma_0}(\langle A!{\uparrow}x\rangle) \qquad \mathit{uninit}_{\sigma_0[x\mapsto 1]}(\langle A?{\downarrow}x\rangle)$$

An event-bag β is *interfering*, denoted $\mathit{int}(\beta)$, when it contains `detv` or two conflicting events (including two occurrences of the same self-conflicting event):

$$\mathit{int}(\beta) \triangleq \texttt{detv} \in \beta \ \vee \ \exists\langle E_1, E_2\rangle \sqsubseteq \beta, \ \mathit{conflict}(E_1, E_2)$$

For example:

$$\mathit{int}(\langle\texttt{detv}\rangle) \qquad \mathit{int}(\langle A!{\uparrow}0, A!{\downarrow}\rangle) \qquad \mathit{int}(\langle A?{\uparrow}, A?{\uparrow}\rangle) \qquad \mathit{int}(\langle A!{\uparrow}x, x{\leftarrow}3\rangle) \qquad \mathit{int}(\langle A!{\uparrow}0, A?{\downarrow}x\rangle)$$

We extend the notion (and notation) of a state-update to bags of events. Given a state σ and event-bag β, the *update* of σ and β, denoted $\sigma[\beta]$, is the state that results after applying

each event in β in some order to state σ. That is, given event-bag $\langle E_1, \ldots, E_n \rangle$:

$$\sigma[\langle E_1, \ldots, E_n \rangle] \triangleq \sigma[E_1] \ldots [E_n]$$

FACT. *Given a non-interfering event-bag, any order in which its events are applied to state σ results in the same final state:*

$$\neg \mathit{int}(\langle E_1, \ldots, E_n \rangle) \ \wedge \ \langle E_1, \ldots, E_n \rangle \stackrel{bag}{=} \langle E'_1, \ldots, E'_n \rangle \implies \sigma[E_1] \ldots [E_n] = \sigma[E'_1] \ldots [E'_n]$$

We are also interested in whether or not the update of a state σ using a bag of events β preserves the property $\mathit{sat}_\sigma(\beta)$. More precisely, we want to express whether or not this property holds *throughout* the update. Given a state σ and an event-bag β such that $\mathit{sat}_\sigma(\beta)$, β is unstable, denoted $\mathit{unstable}_\sigma(\beta)$, when the update of σ using any sub-bag of β does not preserve state-satisfaction of β:

$$\mathit{unstable}_\sigma(\beta) \triangleq \exists \beta' \sqsubseteq \beta, \neg \mathit{sat}_{\sigma[\beta']}(\beta)$$

For example:

$$\mathit{unstable}_{\sigma_0[x \mapsto 5]}(\langle [x = 5], x \leftarrow 6 \rangle) \qquad \mathit{unstable}_{\sigma_0[x \mapsto 5][y \mapsto 5]}(\langle [x = y], x \leftarrow 6, y \leftarrow 6 \rangle)$$

Note that $\mathit{unstable}_\sigma(\beta)$ is implied by the condition $\neg \mathit{sat}_{\sigma[\beta]}(\beta)$ (as in the left example above) or by $\exists E \in \beta, \neg \mathit{sat}_{\sigma[E]}(\beta)$ (as in the right example above), but may also hold if both of those cases do not. For example:

$$\mathit{unstable}_{\sigma_0[w \mapsto 5][x \mapsto 5][y \mapsto 5][z \mapsto 5]}(\langle [w = x \vee y = z], w \leftarrow 6, x \leftarrow 6, y \leftarrow 6, z \leftarrow 6 \rangle)$$

Finally, given a state σ, an event-bag β is *erroneous*, denoted $\mathit{error}_\sigma(\beta)$, when it is synchronized and invalid in σ, or synchronized, satisfied by σ, and either uninitialized, interfering, or unstable:

$$\mathit{error}_\sigma(\beta) \triangleq \mathit{sync}(\beta) \wedge \Big(\mathit{inval}_\sigma(\beta) \vee \Big(\mathit{sat}_\sigma(\beta) \wedge \big(\mathit{uninit}_\sigma(\beta) \vee \mathit{int}(\beta) \vee \mathit{unstable}_\sigma(\beta)\big)\Big)\Big)$$

For example:

$$\mathit{error}_{\sigma_0}(\langle [A\# = x], [\texttt{false}]\rangle) \qquad \mathit{error}_{\sigma_0}(\langle A!{\uparrow}x\rangle) \qquad \mathit{error}_{\sigma_0}(\langle \texttt{detv}\rangle)$$

$$\mathit{error}_{\sigma_0}(\langle x{\leftarrow}0, x{\leftarrow}1\rangle) \qquad \mathit{error}_{\sigma_0[x\mapsto 0]}(\langle [x = 0], x{\leftarrow}1\rangle)$$

Note that an event-bag may be erroneous if it is invalid (contains a wait event with a guard that evaluates to ς) even if it contains a wait event with a guard that evaluates to `false`.

The intuition behind our definition of an erroneous event-bag is that first and foremost, an event-bag is only executable if it is synchronized. Second, we conservatively declare the evaluation of any guard to ς to be an error, regardless of the evaluation of other wait events in the same bag. Third, given the guards of all wait events do not evaluate to ς, each must evaluate to `true` for the event-bag to be executable. And finally, given an event-bag that is both synchronized and satisfied, it is erroneous if it is also uninitialized, interfering, or unstable.

2.3.3 TRACES

A *trace* is a possibly-infinite sequence of finite bags of events, respectively modeling terminating and nonterminating executions of a program. Each bag models simultaneous execution and the order of bags in a trace models their ordered execution. The empty trace is denoted

by ϵ and trace concatenation is denoted by juxtaposition. A trace is formally defined coinductively as ϵ or an event-bag followed by a trace.

Similar to a trace, an *option-trace* is a possibly-infinite sequence with each of its elements being a finite bag of events or $\diamond$, which denotes a unit of suspended execution of a program. The empty option-trace is denoted by $\epsilon_\diamond$ and option-trace concatenation is denoted by juxtaposition. An option-trace is formally defined coinductively as $\epsilon_\diamond$, an event-bag followed by an option-trace, or $\diamond$ followed by an option-trace.

Our need for the use of option-traces stems from the fact that we do not assume fairness of the parallel operator $\|$, which is a common postulate of CHP semantics. This postulate states that given any action of the subprogram of a parallel CHP program, if that action stays enabled to execute then it will eventually execute. We instead allow either subprogram to suspend indefinitely while the other subprogram executes. We formalize this suspension behavior by a possibly-infinite sequence of the repeated $\diamond$ symbol.

It is important to note that while we do allow the subprogram of a parallel CHP program to suspend indefinitely, we do not allow any top-level program to suspend its execution in general. To enforce this restriction, rather than reason about an arbitrary option-trace, we reason about the *lift* of a trace t, denoted $\lfloor t \rfloor_\diamond$, which is the suspension-free option-trace that naturally results from coercing t to an option-trace. This function is defined coinductively as follows:

$$\lfloor \epsilon \rfloor_\diamond \triangleq \epsilon_\diamond$$
$$\lfloor \beta t \rfloor_\diamond \triangleq \beta \lfloor t \rfloor_\diamond$$

We define CHP program behavior in terms of sets of traces and sets of option-traces. As such, the proofs of correctness for the tools we are building will largely involve the need to

demonstrate that a given trace or option-trace is in some set. Moreover, we will construct those proofs by structuring them according to the structure of the given trace or option-trace. Thus, in order to suitably reason about these possibly-infinite coinductive structures, we define certain sets coinductively as well.

STRUCTURAL LANGUAGES

A *structural language* is a set of option-traces that corresponds to the structurally-valid execution of a CHP program.

The *epsilon* structural language, denoted $\boldsymbol{L}_\epsilon$, contains the possibly-suspended epsilon option-trace. It is defined coinductively as follows:

$$\boldsymbol{L}_\epsilon \triangleq \{\epsilon_\diamond\} \cup \{\diamond t_\diamond \mid t_\diamond \in \boldsymbol{L}_\epsilon\}$$

Given an event-bag β, the *singleton-bag* structural language, denoted $\boldsymbol{L}_\beta$, contains the possibly-suspended singleton-bag option-trace. It is defined coinductively as follows:

$$\boldsymbol{L}_\beta \triangleq \{\diamond t_\diamond \mid t_\diamond \in \boldsymbol{L}_\beta\} \cup \{\beta t_\diamond \mid t_\diamond \in \boldsymbol{L}_\epsilon\}$$

The structural language that results from *appending* an option-trace $t_\diamond$ with a structural language L, denoted $t_\diamond \fatsemi L$, is defined coinductively as follows:

$$\begin{aligned}
\epsilon_\diamond \fatsemi L &\triangleq L \\
\diamond t_\diamond \fatsemi L &\triangleq \{\diamond t'_\diamond \mid t'_\diamond \in t_\diamond \fatsemi L\} \\
\beta t_\diamond \fatsemi L &\triangleq \{\beta t'_\diamond \mid t'_\diamond \in t_\diamond \fatsemi L\}
\end{aligned}$$

Note that if $t_\diamond$ is infinite, $t_\diamond \fatsemi L$ is inhabited even if L is empty.

The *concatenation* of two structural languages L_1 and L_2, denoted $L_1 \fatsemi L_2$, is defined as follows:

$$L_1 \fatsemi L_2 \triangleq \{t_\diamond \mid \exists t_\diamond^1 \in L_1, t_\diamond \in t_\diamond^1 \fatsemi L_2\}$$

The structural language that results from *merging* two option-traces $t_\diamond^1$ and $t_\diamond^2$, denoted $t_\diamond^1 \mid\mid\mid t_\diamond^2$, is defined coinductively as follows:

$$\begin{aligned}
\epsilon_\diamond \mid\mid\mid \epsilon_\diamond &\triangleq \{\epsilon_\diamond\} \\
\diamond t_\diamond^1 \mid\mid\mid \diamond t_\diamond^2 &\triangleq \{\diamond t_\diamond \mid t_\diamond \in t_\diamond^1 \mid\mid\mid t_\diamond^2\} \\
\beta t_\diamond^1 \mid\mid\mid \diamond t_\diamond^2 &\triangleq \{\beta t_\diamond \mid t_\diamond \in t_\diamond^1 \mid\mid\mid t_\diamond^2\} \\
\diamond t_\diamond^1 \mid\mid\mid \beta t_\diamond^2 &\triangleq \{\beta t_\diamond \mid t_\diamond \in t_\diamond^1 \mid\mid\mid t_\diamond^2\} \\
\beta_1 t_\diamond^1 \mid\mid\mid \beta_2 t_\diamond^2 &\triangleq \{(\beta_1 \uplus \beta_2) t_\diamond \mid t_\diamond \in t_\diamond^1 \mid\mid\mid t_\diamond^2\}
\end{aligned}$$

Note that $t_\diamond^1$ and $t_\diamond^2$ must have the same length for $t_\diamond^1 \mid\mid\mid t_\diamond^2$ to be inhabited.

The *concur* of two structural languages L_1 and L_2, denoted $L_1 \mid\mid\mid L_2$, is defined as follows:

$$L_1 \mid\mid\mid L_2 \triangleq \{t_\diamond \mid \exists t_\diamond^1 \in L_1, \exists t_\diamond^2 \in L_2, t_\diamond \in t_\diamond^1 \mid\mid\mid t_\diamond^2\}$$

The *union* of two structural languages L_1 and L_2, denoted $L_1 \Cup L_2$, is defined coinductively as follows:

$$L_1 \Cup L_2 \triangleq \{\diamond t_\diamond \mid t_\diamond \in L_1 \Cup L_2\} \cup L_1 \cup L_2$$

Finally, the *star* of an option-trace $t_\diamond$ and structural language L, denoted $t_\diamond \star L$, which is

the structural language that results from appending $t_\diamond$ with a finite or infinite concatenation of non-empty option-traces in L, is defined coinductively as follows:

$$\epsilon_\diamond \star L \triangleq \{\epsilon_\diamond\} \cup \{\diamond t_\diamond \mid t_\diamond \in \epsilon_\diamond \star L\} \cup \{t_\diamond \mid \exists \diamond t'_\diamond \in L, t_\diamond \in \diamond t'_\diamond \star L\} \cup \{t_\diamond \mid \exists \beta t'_\diamond \in L, t_\diamond \in \beta t'_\diamond \star L\}$$
$$\diamond t_\diamond \star L \triangleq \{\diamond t'_\diamond \mid t'_\diamond \in t_\diamond \star L\}$$
$$\beta t_\diamond \star L \triangleq \{\beta t'_\diamond \mid t'_\diamond \in t_\diamond \star L\}$$

Note that we exclude the set $\{t_\diamond \mid \epsilon_\diamond \in L \wedge t_\diamond \in \epsilon_\diamond \star L\}$ from the definition of $\epsilon_\diamond \star L$ in order to exclude the concatenation of empty option-traces in L.

FEASIBLE TRACES

We extend the notion of feasibility from event-bags to traces. Given an initial state σ, a trace t is *feasible*, denoted $\sigma \vdash t$, when each of its event-bags is iteratively feasible. The property is defined coinductively as follows:

$$\frac{}{\sigma \vdash \epsilon} \qquad \frac{\mathit{feas}_\sigma(\beta) \quad \sigma[\beta] \vdash t}{\sigma \vdash \beta t}$$

For example:

$$\sigma_0 \vdash \epsilon \qquad \sigma_0 \vdash \langle[\texttt{true}]\rangle \qquad \sigma_0 \vdash \langle A!\downarrow, A?\downarrow x\rangle\langle B!\downarrow, B?\downarrow y\rangle\langle C!\downarrow, C?\downarrow z\rangle \qquad \sigma_0 \vdash \langle\rangle\langle\rangle\langle\rangle\cdots$$

$$\sigma_0 \vdash \langle x{\leftarrow}0\rangle\langle[x=0], x{\leftarrow}1\rangle\langle[x=1], x{\leftarrow}2\rangle\langle[x=2], x{\leftarrow}3\rangle\cdots$$

This property is defined coinductively because the proof that a possibly-infinite trace is feasible requires a possibly-infinite sequence of event-bag-feasibility proofs.

ERRONEOUS TRACES

We also extend the notion of erroneousness from event-bags to traces. Given an initial state σ, a trace t is *erroneous*, denoted $\sigma \prec\!\!-\, t$, when it contains a finite prefix composed of a feasible trace followed by an erroneous event-bag. The property is defined inductively as follows:

$$\frac{\mathit{error}_\sigma(\beta)}{\sigma \prec\!\!-\, \beta t} \qquad \frac{\mathit{feas}_\sigma(\beta) \quad \sigma[\beta] \prec\!\!-\, t}{\sigma \prec\!\!-\, \beta t}$$

For example:

$$\sigma_0 \prec\!\!-\, \langle [x = x] \rangle \qquad \sigma_0 \prec\!\!-\, \langle A!\!\downarrow, A?\!\downarrow x, A!\!\downarrow, A?\!\downarrow y \rangle \qquad \sigma_0 \prec\!\!-\, \langle x := x \rangle$$

$$\sigma_0 \prec\!\!-\, \langle x \leftarrow 0 \rangle \langle [x = 0], x \leftarrow 1 \rangle \langle [x = 1], x \leftarrow 2 \rangle \langle [x = 2], x \leftarrow 3 \rangle \cdots$$

This property is defined inductively because an erroneous trace need only be *partially* feasible and have a single erroneous event-bag. The proof that a trace is erroneous requires a finite sequence of event-bag-feasibility proofs followed by a single proof that an event-bag is erroneous.

2.3.4 PROGRAM BEHAVIOR

We formalize the *raw* behavior of a CHP program P, denoted $\{\!|P|\!\}$, by a structural language where each of its option-traces models a structurally-valid possibly-suspended execution of P. This formalization is shown in Figure 2.2. The raw behavior of a communication program C, is formed by the concatenation of its *initiation*-behavior structural language, denoted $\{\!\{C\}\!\}$, and its *completion*-behavior bag of events, denoted $\langle\!\langle C \rangle\!\rangle$.

The *feasible* behavior of a CHP program P, denoted $[\![P]\!]$, is the set of traces whose lift is

$$\{\!\{A!(e)\}\!\} \triangleq \boldsymbol{L}_{\langle A!\uparrow e\rangle} \qquad \langle\!\langle A!(e)\rangle\!\rangle \triangleq \langle A!\downarrow\rangle$$
$$\{\!\{A?(x)\}\!\} \triangleq \boldsymbol{L}_{\langle A?\uparrow\rangle} \qquad \langle\!\langle A?(x)\rangle\!\rangle \triangleq \langle A?\downarrow x\rangle$$
$$\{\!\{C_1 \bullet C_2\}\!\} \triangleq \{\!\{C_1\}\!\} \mid\mid\mid \{\!\{C_2\}\!\} \qquad \langle\!\langle C_1 \bullet C_2\rangle\!\rangle \triangleq \langle\!\langle C_1\rangle\!\rangle \uplus \langle\!\langle C_2\rangle\!\rangle$$

$$\begin{aligned}
\{\!\{\texttt{skip}\}\!\} &\triangleq \boldsymbol{L}_{\epsilon} \\
\{\!\{x := e\}\!\} &\triangleq \boldsymbol{L}_{\langle x\leftarrow e\rangle} \\
\{\!\{C\}\!\} &\triangleq \{\!\{C\}\!\} \fatsemi \boldsymbol{L}_{\langle\!\langle C\rangle\!\rangle} \\
\{\!\{P_1 \,;\, P_2\}\!\} &\triangleq \{\!\{P_1\}\!\} \fatsemi \{\!\{P_2\}\!\} \\
\{\!\{P_1 \parallel P_2\}\!\} &\triangleq \{\!\{P_1\}\!\} \mid\mid\mid \{\!\{P_2\}\!\} \\
\{\!\{[G_1 \to P_1 \mid G_2 \to P_2]\}\!\} &\triangleq \left(\boldsymbol{L}_{\langle [G_1]\rangle} \fatsemi \{\!\{P_1\}\!\}\right) \Cup \left(\boldsymbol{L}_{\langle [G_2]\rangle} \fatsemi \{\!\{P_2\}\!\}\right) \\
\{\!\{[G_1 \to P_1 \,[\!]\, G_2 \to P_2]\}\!\} &\triangleq \left(\boldsymbol{L}_{\langle [G_1]\rangle} \fatsemi \{\!\{P_1\}\!\}\right) \Cup \left(\boldsymbol{L}_{\langle [G_2]\rangle} \fatsemi \{\!\{P_2\}\!\}\right) \Cup \boldsymbol{L}_{\langle [G_1],[G_2],\texttt{detv}\rangle} \\
\{\!\{*[G \to P]\}\!\} &\triangleq \left(\epsilon_\diamond \star \left(\boldsymbol{L}_{\langle [G]\rangle} \fatsemi \{\!\{P\}\!\}\right)\right) \fatsemi \boldsymbol{L}_{\langle [\neg G]\rangle}
\end{aligned}$$

Figure 2.2: raw CHP behavior

in $\{\!\{P\}\!\}$ and which are feasible from the zero state:

$$[\![P]\!] \triangleq \{t \mid \lfloor t \rfloor_\diamond \in \{\!\{P\}\!\} \wedge \sigma_0 \vdash t\}$$

Similarly, the *erroneous* behavior of a CHP program P, denoted $\mathbb{E}[P]$, is the set of traces whose lift is in $\{\!\{P\}\!\}$ and which are erroneous from the zero state:

$$\mathbb{E}[P] \triangleq \{t \mid \lfloor t \rfloor_\diamond \in \{\!\{P\}\!\} \wedge \sigma_0 \prec t\}$$

3 DELIMITED LABELED SAFE PETRI NETS

We have used sets of traces and option-traces to provide a concise high-level formalization of CHP program behavior. We now use transition systems to employ basic low-level algorithms (e.g. CHP program simulation) that analyze and reason about that behavior. Given these two forms, we must ensure that our usage of each is correct with respect to the CHP programs they represent and correct with respect to one another. We do so with mechanical proofs.

The transition systems we use to employ CHP-semantic algorithms are called *Petri nets*. These are commonly used to model concurrent systems. In fact, Petri nets are already used in lower-level stages of Martin synthesis. Here, we use them at the CHP level in order to answer questions about the behavior of a circuit independent of lower-level design decisions.

3.1 STRUCTURE

We model the raw behavior of a CHP program using a *delimited labeled safe* Petri net, a machine from which we can generate a set of option-traces. A Petri net's structure consists of *places* and *transitions* between non-empty sets of places. We call each non-empty set of places a *configuration*. Thus, each transition consists of a starting configuration and an ending configuration. A *delimited* Petri net further has an *initial* configuration and a *final* configuration. In a *labeled* Petri net, each transition additionally has an optional label. In our case, labels are bags of events, and we use γ to indicate the lack of a label.

We use $S_1 \uplus S_2$ to denote the disjoint union of sets S_1 and S_2, i.e. the set $S_1 \cup S_2$, but which is well-defined when $S_1 \cap S_2 = \emptyset$. Let S^+ denote the set of non-empty subsets of a set S, and let $\mathcal{B}$ denote the set of bags of events. Then, a delimited labeled Petri net is a tuple $N = (\mathcal{P}, c_{init}, c_{fin}, \mathcal{T})$ where:

- $\mathcal{P}$ is a finite set of places,
- $c_{init} \in \mathcal{P}^+$ is the initial configuration,
- $c_{fin} \in \mathcal{P}^+$ is the final configuration,
- and $\mathcal{T} \subseteq \mathcal{P}^+ \times (\mathcal{B} \uplus \{\gamma\}) \times \mathcal{P}^+$ is a finite set of transitions.

A delimited labeled Petri net is *safe* when certain properties hold of its operation, defined below.

3.2 OPERATION

We do not particularly care about the Petri net itself, but rather about the set of all option-traces that it can generate. When we turn a CHP program into a Petri net, we want the Petri net's set of generateable option-traces to be exactly those that define the raw behavior of the CHP program. Then we can perform some analysis on the Petri net and make inferences about the behavior of the original CHP program.

A Petri net operates via a possibly-infinite sequence of firings which generates an option-trace. Given a Petri net N and its set of transitions $\mathcal{T}$, there is a *firing* between configurations c and c' with an optional bag of events $\omega \in \mathcal{B} \uplus \{\gamma\}$, denoted $c \xrightarrow{\omega}_N c'$, when the following holds:

$$\frac{(c_i, \omega, c_o) \in \mathcal{T} \quad c_i \subseteq c}{c \xrightarrow{\omega}_N (c \setminus c_i) \cup c_o}$$

A configuration c is *reachable* in Petri net N, denoted $\mathit{reach}_N(c)$, when there exists a finite sequence of firings from N's initial configuration c_{init} to c. Reachability is defined inductively as follows:

$$\frac{}{\mathit{reach}_N(c_{\mathit{init}})} \qquad \frac{\mathit{reach}_N(c) \quad c \xrightarrow{\omega}_N c'}{\mathit{reach}_N(c')}$$

A Petri net N is *safe*, denoted $\mathit{safe}(N)$, when any reachable configuration containing its final configuration c_{fin}, is *exactly* its final configuration:

$$\mathit{safe}(N) \triangleq \forall c, \mathit{reach}_N(c) \wedge c_{\mathit{fin}} \subseteq c \implies c \subseteq c_{\mathit{fin}}$$

A *list* is a finite sequence of elements. Let ϵ_l denote the empty list and let juxtaposition denote list concatenation.

Given Petri net N, two of its configurations c and c', and a list of its configurations l, there is an *empty firing chain*, denoted $c \xrightarrow{l}\!\!\!\!\rightarrow_N c'$, when there exists a finite sequence of firings with no event-bags from c to c' in N whose configurations combine to form l. An empty firing chain is defined inductively as follows:

$$\frac{}{c \xrightarrow{\epsilon_l}\!\!\!\!\rightarrow_N c} \qquad \frac{c \xrightarrow{\gamma}_N c' \quad c' \xrightarrow{l}\!\!\!\!\rightarrow_N c''}{c \xrightarrow{c'l}\!\!\!\!\rightarrow_N c''}$$

A *path* is a possibly-infinite alternating sequence of configuration lists and configurations where the first element and last element (if any) is a configuration list. A path is formally defined coinductively as a (final) configuration list or a configuration list followed by a single configuration followed by a path (where each "followed by" is denoted by juxtaposition).

A Petri net N *generates* an option-trace $t_\diamond$ by starting from configuration c and following path π, denoted $c \vDash^{\pi}_{N} t_\diamond$, when there exists a possibly-infinite sequence of firings from c

whose configurations combine to form π, whose event-bags combine to form $t_\diamond$, and whose last configuration (if any) is N's final configuration. This property is defined coinductively as follows:

$$\frac{c \xrightarrow{l}\!\!\!\!\rightarrow_N c_{fin}}{c \vDash^{l}_{N} \epsilon_\diamond} \qquad \frac{c \xrightarrow{l}\!\!\!\!\rightarrow_N c' \quad c' \vDash^{\pi}_{N} t_\diamond}{c \vDash^{lc'\pi}_{N} \diamond t_\diamond} \qquad \frac{c \xrightarrow{l}\!\!\!\!\rightarrow_N c' \quad c' \xrightarrow{\beta}_N c'' \quad c'' \vDash^{\pi}_{N} t_\diamond}{c \vDash^{lc''\pi}_{N} \beta t_\diamond}$$

This formalization ensures that an infinite sequence of γ-labeled firings in general has no direct meaning. Instead, only a finite sequence of γ-labeled firings can be collapsed at each unit of Petri-net operation. This is favorable because a γ-labeled transition is simply a formal device that does not actually correspond to a physical circuit event. In particular, we will see in the next Section that γ-labeled transitions are intended to help adapt the behavior of one or two existing Petri nets, not to introduce new standalone behavior.

Finally, the language of a Petri net N, denoted $\mathcal{L}(N)$, is the set of option-traces N generates from its initial configuration c_{init}:

$$\mathcal{L}(N) \triangleq \{t_\diamond \mid \exists \pi, c_{init} \vDash^{\pi}_{N} t_\diamond\}$$

3.3 BASIC CONSTRUCTIONS

Like our CHP trace semantics, we will construct Petri nets according to the syntactic structure of a CHP program. We further prove that each of those constructions is safe and that their languages appropriately correspond to the structural languages presented in Section 2.3.3. Some of those language proofs require the axiom of *Dependent Choice*, denoted DC, which is defined as follows:

$$DC \triangleq \forall X, R \subseteq X \times X, \big(\forall x, \exists y, (x, y) \in R\big) \implies \forall x, \exists f, f(0) = x \wedge \forall n, \big(f(n), f(n+1)\big) \in R$$

This axiom is commonly used in both constructive and classical settings, and is the only assumption we make across all of our mechanical Coq proofs. We indicate precisely which proofs use the axiom.

Our Petri-net constructions consist of the epsilon Petri net; the singleton-bag Petri net; the sequential, parallel, and selection composition of two Petri nets; and the repetition of a Petri net. Along with their mechanically-proven safety and language properties, these constructions are defined formally and graphically below, where $\circ$ and $\bullet$ are any two distinct abstract elements.

The epsilon Petri net, denoted $\boldsymbol{N}_\epsilon$, is defined as follows:

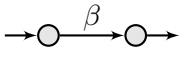
$$\boldsymbol{N}_\epsilon \triangleq \Big(\{\bullet\},\ \{\bullet\},\ \{\bullet\},\ \emptyset\Big)$$

LEMMA (Epsilon Safety)**.** $\mathit{safe}(\boldsymbol{N}_\epsilon)$

LEMMA (Epsilon Correctness)**.** $\mathcal{L}(\boldsymbol{N}_\epsilon) = \boldsymbol{L}_\epsilon$

Given a bag of events β, the singleton-bag Petri net, denoted $\boldsymbol{N}_\beta$, is defined as follows:

$$\boldsymbol{N}_\beta \triangleq \Big(\{\circ, \bullet\},\ \{\circ\},\ \{\bullet\},\ \big\{\big(\{\circ\}, \beta, \{\bullet\}\big)\big\}\Big)$$

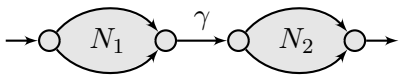


LEMMA (Singleton-Bag Safety)**.** $\forall \beta,\ \mathit{safe}(\boldsymbol{N}_\beta)$

LEMMA (Singleton-Bag Correctness)**.** $\forall \beta,\ \mathcal{L}(\boldsymbol{N}_\beta) = \boldsymbol{L}_\beta$

Given two Petri nets $N_1 = (\mathcal{P}_1, c_{init1}, c_{fin1}, \mathcal{T}_1)$ and $N_2 = (\mathcal{P}_2, c_{init2}, c_{fin2}, \mathcal{T}_2)$, their sequential composition, denoted $N_1 \cdot N_2$, is defined as follows:

$$N_1 \cdot N_2 \triangleq \Big(\mathcal{P}_1 \uplus \mathcal{P}_2,\ c_{init1},\ c_{fin2},\ \mathcal{T}_1 \uplus \big\{(c_{fin1}, \gamma, c_{init2})\big\} \uplus \mathcal{T}_2\Big)$$



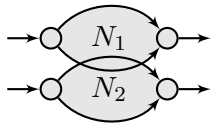
LEMMA (Sequential Safety). $\forall N_1, N_2,\ \mathit{safe}(N_1) \wedge \mathit{safe}(N_2) \implies \mathit{safe}(N_1 \cdot N_2)$

LEMMA (Sequential Soundness). $\forall N_1, N_2,\ \mathit{safe}(N_1) \implies \mathcal{L}(N_1 \cdot N_2) \subseteq \mathcal{L}(N_1) \fatsemi \mathcal{L}(N_2)$

LEMMA (Sequential Completeness). $DC \implies \forall N_1, N_2,\ \mathcal{L}(N_1) \fatsemi \mathcal{L}(N_2) \subseteq \mathcal{L}(N_1 \cdot N_2)$

Given two Petri nets $N_1 = (\mathcal{P}_1, c_{init1}, c_{fin1}, \mathcal{T}_1)$ and $N_2 = (\mathcal{P}_2, c_{init2}, c_{fin2}, \mathcal{T}_2)$, their parallel composition, denoted $N_1 \times N_2$, is defined as follows:

$$N_1 \times N_2 \triangleq \Big(\mathcal{P}_1 \uplus \mathcal{P}_2,\ c_{init1} \uplus c_{init2},\ c_{fin1} \uplus c_{fin2},\ \mathcal{T}_1 \uplus \mathcal{T}_2 \uplus \big\{(c_1 \uplus c_2, \beta_1 \uplus \beta_2, c_1' \uplus c_2') \mid (c_1, \beta_1, c_1') \in \mathcal{T}_1 \ \wedge\ (c_2, \beta_2, c_2') \in \mathcal{T}_2\big\}\Big)$$

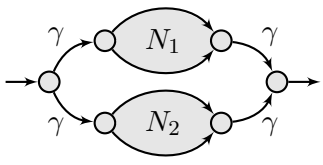


LEMMA (Parallel Safety). $\forall N_1, N_2,\ \mathit{safe}(N_1) \wedge \mathit{safe}(N_2) \implies \mathit{safe}(N_1 \times N_2)$

LEMMA (Parallel Correctness). $\forall N_1, N_2,\ \mathcal{L}(N_1 \times N_2) = \mathcal{L}(N_1) \mid\mid\mid \mathcal{L}(N_2)$

Given two Petri nets $N_1 = (\mathcal{P}_1, c_{init1}, c_{fin1}, \mathcal{T}_1)$ and $N_2 = (\mathcal{P}_2, c_{init2}, c_{fin2}, \mathcal{T}_2)$, their selection composition, denoted $N_1 + N_2$, is defined as follows:

$$N_1 + N_2 \triangleq \Big(\{\circ\} \uplus \mathcal{P}_1 \uplus \mathcal{P}_2 \uplus \{\bullet\},\ \{\circ\},\ \{\bullet\},\ \big\{(\{\circ\}, \gamma, c_{init1}), (\{\circ\}, \gamma, c_{init2})\big\} \uplus \mathcal{T}_1 \uplus \mathcal{T}_2 \uplus \big\{(c_{fin1}, \gamma, \{\bullet\}), (c_{fin2}, \gamma, \{\bullet\})\big\}\Big)$$



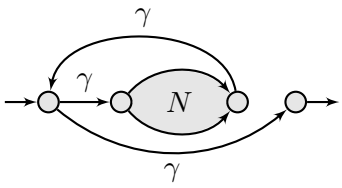
LEMMA (Selection Safety)**.** $\forall N_1, N_2,\ \mathit{safe}(N_1) \wedge \mathit{safe}(N_2) \implies \mathit{safe}(N_1 + N_2)$

LEMMA (Selection Soundness)**.** $\forall N_1, N_2,\ \mathit{safe}(N_1) \wedge \mathit{safe}(N_2) \implies \mathcal{L}(N_1 + N_2) \subseteq \mathcal{L}(N_1) \uplus \mathcal{L}(N_2)$

LEMMA (Selection Completeness)**.** $DC \implies \forall N_1, N_2,\ \mathcal{L}(N_1) \uplus \mathcal{L}(N_2) \subseteq \mathcal{L}(N_1 + N_2)$

Given a Petri net $N = (\mathcal{P}, c_{\mathit{init}}, c_{\mathit{fin}}, \mathcal{T})$, its repetition construction, denoted $N\circlearrowleft$, is defined as follows:

$$
\begin{aligned}
N\circlearrowleft \triangleq \Big(& \{\circ\} \uplus \mathcal{P} \uplus \{\bullet\},\ \{\circ\},\ \{\bullet\}, \\
& \big\{ \big(\{\circ\}, \gamma, c_{\mathit{init}}\big), \big(\{\circ\}, \gamma, \{\bullet\}\big) \big\} \\
& \uplus \mathcal{T} \uplus \big\{ \big(c_{\mathit{fin}}, \gamma, \{\circ\}\big) \big\} \Big)
\end{aligned}
$$



LEMMA (Repetition Safety)**.** $\forall N,\ \mathit{safe}(N) \implies \mathit{safe}(N\circlearrowleft)$

LEMMA (Repetition Soundness)**.** $\forall N,\ \mathit{safe}(N) \implies \mathcal{L}(N\circlearrowleft) \subseteq \epsilon_\diamond \star \mathcal{L}(N)$

LEMMA (Repetition Completeness)**.** $DC \implies \forall N,\ \epsilon_\diamond \star \mathcal{L}(N) \subseteq \mathcal{L}(N\circlearrowleft)$

4 A CERTIFIED CHP SIMULATOR

We now present our CHP program simulator, which is certified to operate correctly with respect to our CHP trace semantics. The two main steps of the simulator are to first construct a Petri net N from a CHP program P such that $\mathcal{L}(N) = \{\!|P|\!\}$, and to then generate a trace t such that $\lfloor t \rfloor_\diamond \in \mathcal{L}(N)$ and $\sigma_0 \vdash t$ (and therefore $t \in [\![P]\!]$). We use the Coq Proof Assistant to build our simulator, mechanically verify each of its steps, and ultimately certify that it correctly simulates a CHP program.

4.1 PETRI-NET CONSTRUCTION

The first main step of our simulator is to construct a Petri net from a CHP program. We define functions $\mathbf{constr'}(C)$ and $\mathbf{constr}(P)$ to inductively construct a Petri net from communication program C and from program P, respectively. Their definitions are shown in Figure 4.1. Our mechanical proofs of correctness for these constructed Petri nets are shown below, and follow easily from the definitions in Figure 2.2 and the lemmas in Section 3.3.

LEMMA (Communication-Initiation Safety)**.** $\forall C,\ \mathit{safe}(\mathbf{constr'}(C))$

LEMMA (Communication-Initiation Correctness)**.** $\forall C,\ \mathcal{L}(\mathbf{constr'}(C)) = \{\!\{C\}\!\}$

LEMMA (Construction Safety)**.** $\forall P,\ \mathit{safe}(\mathbf{constr}(P))$

THEOREM (Construction Soundness)**.** $\forall P,\ \mathcal{L}(\mathbf{constr}(P)) \subseteq \{\!|P|\!\}$

$$\begin{aligned}
\mathbf{constr'}(A!(e)) &\triangleq \boldsymbol{N}_{\langle A!\uparrow e\rangle} \\
\mathbf{constr'}(A?(x)) &\triangleq \boldsymbol{N}_{\langle A?\uparrow\rangle} \\
\mathbf{constr'}(C_1 \bullet C_2) &\triangleq \mathbf{constr'}(C_1) \times \mathbf{constr'}(C_2)
\end{aligned}$$

$$\begin{aligned}
\mathbf{constr}(\texttt{skip}) &\triangleq \boldsymbol{N}_{\epsilon} \\
\mathbf{constr}(x := e) &\triangleq \boldsymbol{N}_{\langle x\leftarrow e\rangle} \\
\mathbf{constr}(C) &\triangleq \mathbf{constr'}(C) \cdot \boldsymbol{N}_{\langle\!\langle C\rangle\!\rangle} \\
\mathbf{constr}(P_1 \,;\, P_2) &\triangleq \mathbf{constr}(P_1) \cdot \mathbf{constr}(P_2) \\
\mathbf{constr}(P_1 \parallel P_2) &\triangleq \mathbf{constr}(P_1) \times \mathbf{constr}(P_2) \\
\mathbf{constr}([G_1 \to P_1 \mid G_2 \to P_2]) &\triangleq \big(\boldsymbol{N}_{\langle[G_1]\rangle} \cdot \mathbf{constr}(P_1)\big) + \big(\boldsymbol{N}_{\langle[G_2]\rangle} \cdot \mathbf{constr}(P_2)\big) \\
\mathbf{constr}([G_1 \to P_1 \,[\!]\, G_2 \to P_2]) &\triangleq \big(\boldsymbol{N}_{\langle[G_1]\rangle} \cdot \mathbf{constr}(P_1)\big) + \big(\boldsymbol{N}_{\langle[G_2]\rangle} \cdot \mathbf{constr}(P_2)\big) + \boldsymbol{N}_{\langle[G_1],[G_2],\texttt{detv}\rangle} \\
\mathbf{constr}(*[G \to P]) &\triangleq \big(\boldsymbol{N}_{\langle[G]\rangle} \cdot \mathbf{constr}(P)\big)^{\circlearrowleft} \cdot \boldsymbol{N}_{\langle[\neg G]\rangle}
\end{aligned}$$

Figure 4.1: CHP program Petri net construction

THEOREM (Construction Completeness)**.** $DC \implies \forall P,\ \{\!|P|\!\} \subseteq \mathcal{L}(\mathbf{constr}(P))$

4.2 TRACE GENERATION

The second main step of our simulator is to generate a trace from the above Petri-net construction of a CHP program. The constructed Petri net generates the *raw* behavior of a CHP program, but our simulator must find the suspension-free feasible traces within that Petri net's language in order to produce the *feasible* behavior of a program, as defined in Section 2.3.4.

Given a CHP program P, our top-level simulation function $\mathbf{sim}_P$ steps through successive configurations of $\mathbf{constr}(P)$ that combine together to form a firing-sequence that generates a suspension-free feasible trace. The $\mathbf{sim}_P$ function additionally takes an infinite sequence of natural numbers i as input and corecursively constructs its possibly-infinite result piece-by-piece with each natural number it gets from i.

The **sim**$_P$ function constructs its output by making successive calls to the operation-search function **nth-op**. The **nth-op** function takes as input a configuration c that is reachable in Petri net **constr**(P), a state σ, and a natural number n (which **sim**$_P$ passes from its input stream i), and it outputs the nth suspension-free σ-feasible operation from c (if any). Such an operation is either to follow a finite empty firing chain from c to the final configuration of **constr**(P) and terminate, or to emit an event-bag that is feasible with respect to σ and transition from c to a new configuration in **constr**(P) according to Petri-net trace generation defined in Section 3.2.

The **nth-op** function operates on the *lazy* Petri net constructed from a CHP program P, denoted **lazy**(P), rather than operating directly on the (non-lazy) Petri net **constr**(P). This is because we found that performing the Petri-net construction defined in Figure 4.1 is too computationally expensive for large CHP programs. In doing so, we are effectively enumerating every possible structurally-valid option-trace of a CHP program, and that set of option-traces becomes exponentially large for multiple subprograms in parallel with one another. Instead, the **sim**$_P$ function builds the *lazy* Petri net **lazy**(P) and utilizes the **nth-op** function to lazily generate a single suspension-free feasible trace. We then prove that the lift of that trace is an option-trace that could have been generated by the non-lazy Petri net **constr**(P).

4.2.1 LAZY PETRI NETS

We define the same basic constructions for lazy Petri nets as that of non-lazy Petri nets presented in Section 3.3, and we prove each lazy construction correct according to the corresponding non-lazy construction, e.g. $\boldsymbol{LN}_\epsilon$ correctly corresponds to $\boldsymbol{N}_\epsilon$, $\boldsymbol{LN}_\beta$ correctly corresponds to $\boldsymbol{N}_\beta$, etc. We then define the function **lazy**(P) to inductively construct a lazy Petri net from CHP program P in the same manner as that of **constr**(P) shown in Figure 4.1, and

we prove that **lazy**(P) correctly corresponds to **constr**(P).

A *lazy* Petri net is a tuple $LN = (\mathcal{P}, c_{init}, \textbf{efc-fin}, \textbf{nth-bag})$ where:

- $\mathcal{P}$ is a finite set of places equal to that of LN's corresponding non-lazy Petri net,
- $c_{init} \in \mathcal{P}^+$ is the initial configuration equal to that of LN's corresponding non-lazy Petri net,
- **efc-fin**, which stands for *empty firing chain to final* configuration, is a function such that given a configuration c that is reachable in LN's corresponding non-lazy Petri net, it indicates whether not there exists an empty firing chain from c to the final configuration,
- and **nth-bag**, which stands for n*th* next feasible *event-bag* operation, is a function such that given a configuration c that is reachable in LN's corresponding non-lazy Petri net, a state σ, and a natural number n, it indicates the nth possible pair (β, c') (if any) such that event-bag β is feasible with respect to σ and the Petri net can emit β by transitioning to configuration c'.

While we specify the behavior of functions **efc-fin** and **nth-bag** more formally below, it is important to note that they do not operate on a corresponding non-lazy Petri net itself, but instead make decisions solely based on the structure of their input configuration. As such, the construction of a non-lazy Petri net is not used to *construct* a lazy Petri net or its parts, but rather to *certify* that a lazy Petri net has been constructed correctly.

The correctness of a lazy Petri net depends upon the correctness of each of its parts. We begin with a lazy Petri net's set of places $\mathcal{P}_{LN}$ and its initial configuration $c_{init\,LN}$. Given non-lazy Petri net $N = (\mathcal{P}_N, c_{init\,N}, c_{fin}, \mathcal{T})$, $\mathcal{P}_{LN}$ and $c_{init\,LN}$ are respectively correct with respect to N as follows:

SPECIFICATION (Place-Set Correctness)**.** $\mathcal{P}_{LN} = \mathcal{P}_N$

SPECIFICATION (Initial-Configuration Correctness)**.** $c_{init\,LN} = c_{init\,N}$

The **efc-fin**$_{LN}$ function of a lazy Petri net LN indicates whether not there exists an empty firing chain from a given reachable configuration in LN's corresponding non-lazy Petri net to the final configuration. More formally, given non-lazy Petri net $N = (\mathcal{P}_N, c_{init\,N}, c_{fin}, \mathcal{T})$, **efc-fin**$_{LN}$ is correct with respect to N as follows:

SPECIFICATION (Efc-Fin Correctness)**.**

$$\forall c,\ reach_N(c) \implies \big(\textbf{efc-fin}_{LN}(c) = \texttt{true} \iff \exists l, c \xrightarrow{l}\!\!\!\!\!\rightarrow_N c_{fin}\big)$$

The definition of correctness for the **nth-bag** function of a lazy Petri net LN is particularly complex. Informally, given a reachable configuration c in LN's corresponding non-lazy Petri net, a state σ, and a natural number n, the **nth-bag** function outputs the nth non-terminating operation from c with an event-bag that is feasible with respect to σ.

Recall that an event-bag is feasible with respect to state σ if it is both synchronized and satisfied by σ. Thus, we construct two functions, **sync** and **sat**, which are utilized by the **nth-bag** function and which we mechanically prove to be correct as follows:

LEMMA (Sync Correctness)**.** $\forall \beta,\ \textbf{sync}(\beta) = \texttt{true} \iff sync(\beta)$

LEMMA (Sat Correctness)**.** $\forall \sigma, \beta,\ \textbf{sat}_\sigma(\beta) = \texttt{true} \iff sat_\sigma(\beta)$

Since we build lazy Petri nets and their respective **nth-bag** functions modularly, the task of finding a feasible operation becomes complicated by the fact that although we can *locally* check for satisfiability of an event-bag (i.e. $\forall \sigma, \beta_1, \beta_2, sat_\sigma(\beta_1) \wedge sat_\sigma(\beta_2) \iff sat_\sigma(\beta_1 \uplus \beta_2)$), we must *globally* check for synchronization. We explored building our simulator with an **nth-bag** function that instead outputs a list of all *satisfied* operations from its input configuration, in which case the aforementioned complication is mitigated, but we found that constructing such a list for large Petri nets was too computationally expensive to be practical.

Conclusively, we specify **nth-bag** as a continuation-passing style function that takes as input a configuration c that is reachable in its corresponding non-lazy Petri net, a state σ, a Boolean b, a function F of type $(\mathcal{B} \times \mathcal{P}^+ \times \mathbb{N}) \to \big(\mathcal{B} \times \mathcal{P}^+ \times (\mathbb{N} \uplus \{\mu\})\big) \uplus \{\delta\}$, and a natural number n, and that outputs a value in $\big(\mathcal{B} \times \mathcal{P}^+ \times (\mathbb{N} \uplus \{\mu\})\big) \uplus \{\delta\}$. Intuitively, F is a continuation that takes as input a local operation from configuration c and uses it to build the nth globally-feasible operation from some configuration that is a superset of c. For example, given lazy Petri nets $LN_1 = (\mathcal{P}_1, c_{init1}, \textbf{efc-fin}_1, \textbf{nth-bag}_1)$ and $LN_2 = (\mathcal{P}_2, c_{init2}, \textbf{efc-fin}_2, \textbf{nth-bag}_2)$, the **nth-bag** function for $LN_1 \times LN_2$ calls $\textbf{nth-bag}_1$ with a continuation that takes as input an operation local to LN_1 and tries to merge it with an operation local to LN_2 (which the continuation finds by calling $\textbf{nth-bag}_2$) in order to build a globally feasible operation of $LN_1 \times LN_2$.

More specifically, $\textbf{nth-bag}(c, \sigma, b, F, n)$ iterates through those operations from c with an event-bag that is satisfied by σ and is synchronized if $b = \texttt{true}$. If such an operation is found, then its event-bag β and ending configuration c' are passed to function F along with natural number n. If F returns δ, then this indicates that F cannot use the given local operation to build a globally-feasible operation, and thus **nth-bag** tries calling F again with another local operation. If F returns a triple (β', c'', m), then (β', c'') is a globally-feasible operation successfully built from the local operation (β, c'). Further, if $m = \mu$, then exactly the nth such global operation has been found, and **nth-bag** returns the triple. Otherwise, m is the natural number such that m more globally-feasible operations need to be iterated through in order to find the nth; in this case **nth-bag** tries calling F again with another local operation and with m instead of n.

We formally define the $\textbf{nth-bag}_{LN}$ function of a lazy Petri net LN to be *sound* with respect to non-lazy Petri net N as follows:

SPECIFICATION (Nth-Bag Soundness)**.**

$$\forall c, \sigma, b, F, n,\ \mathit{reach}_N(c)\ \wedge\ \mathbf{nth\text{-}bag}_{LN}(c, \sigma, b, F, n) \neq \delta\ \Longrightarrow$$
$$\exists l, c', \beta, c'', n',\ c \xrightarrow{l}\!\!\!\!\rightarrow_N c'\ \wedge\ c' \xrightarrow{\beta}_N c''\ \wedge\ \mathit{sat}_\sigma(\beta)\ \wedge\ \big(b = \texttt{true} \Rightarrow \mathit{sync}(\beta)\big)\ \wedge$$
$$\mathbf{nth\text{-}bag}_{LN}(c, \sigma, b, F, n) = F(\beta, c'', n')$$

Intuitively, if $\mathbf{nth\text{-}bag}_{LN}(c, \sigma, b, F, n)$ is not δ, then it must be the result of an appropriate call to F.

Next, we define two properties, *δ-consistency* and *n-surjectivity*, each of which specifies the behavior of a function of type $\mathbb{N} \to \big(\mathcal{B} \times \mathcal{P}^+ \times (\mathbb{N} \uplus \{\mu\})\big) \uplus \{\delta\}$. Such a function f is *δ-consistent*, denoted $\delta\text{-}\mathit{cons}(f)$, when f outputs δ for any input if it does so for some particular input:

$$\delta\text{-}\mathit{cons}(f) \triangleq \Big(\big(\exists n, f(n) = \delta\big)\ \Longrightarrow\ \forall n, f(n) = \delta\Big)$$

and f is *n-surjective*, denoted $n\text{-}\mathit{surj}(f)$, when for any target natural number n', there exists an input such that f returns a triple with n' as its third element if f outputs a triple for some particular input:

$$n\text{-}\mathit{surj}(f) \triangleq \Big(\big(\exists n, f(n) \neq \delta\big)\ \Longrightarrow\ \forall n',\ \exists n, \beta, c, f(n) = (\beta, c, n')\Big)$$

Given these properties, we formally define the $\mathbf{nth\text{-}bag}_{LN}$ function of a lazy Petri net LN to be *complete* with respect to non-lazy Petri net N as follows:

SPECIFICATION (Nth-Bag Completeness)**.**

$$\forall c, l, c', \beta, c'', \sigma, b, F,\ \mathit{reach}_N(c)\ \wedge\ c \xrightarrow{l}\!\!\!\!\rightarrow_N c'\ \wedge\ c' \xrightarrow{\beta}_N c''\ \wedge\ \mathit{sat}_\sigma(\beta)\ \wedge\ \big(b = \texttt{true} \Rightarrow \mathit{sync}(\beta)\big)\ \wedge$$
$$\Big(\forall \beta, c,\ \delta\text{-}\mathit{cons}\big(\lambda n.\ F(\beta, c, n)\big)\ \wedge\ n\text{-}\mathit{surj}\big(\lambda n.\ F(\beta, c, n)\big)\Big)\ \Longrightarrow$$
$$\exists c''', l',\ \mathit{reach}_N(c''')\ \wedge\ c''' \xrightarrow{l'}\!\!\!\!\rightarrow_N c''\ \wedge$$
$$\Big(\forall n',\ \exists n,\ \big(\exists \beta', c',\ F(\beta, c''', n') = (\beta', c', \mu)\big) \Longrightarrow \mathbf{nth\text{-}bag}_{LN}(c, \sigma, b, F, n) = F(\beta, c''', n')\Big)$$

Intuitively, given an appropriate local operation (β, c'') from reachable configuration c and given a continuation F that is both δ-consistent and n-surjective, there exists a configuration c''' with an empty firing chain to c'' such that for any natural number n', there exists a natural number n such that if $F(\beta, c''', n')$ returns a triple with μ as its third element, then **nth-bag**$_{LN}(c, \sigma, b, F, n)$ equals that triple. Note that it is not necessarily the case that **nth-bag**$_{LN}(c, \sigma, b, F, n) = F(\beta, c'', n')$. This is precisely because of our intended simulation of the cross product of two lazy Petri nets LN_1 and LN_2. In this case, the (β, c'') operation's empty-firing-chain component $c \stackrel{l}{\twoheadrightarrow}_N c'$ might consist of empty firing chains from both LN_1 and LN_2, but we only want to step through both if the operation's bag-firing component $c' \stackrel{\beta}{\rightarrow}_N c''$ is a merge of firings from both LN_1 and LN_2. If $c' \stackrel{\beta}{\rightarrow}_N c''$ is a non-merged firing from LN_1, then we only step through the γ-firings of $c \stackrel{l}{\twoheadrightarrow}_N c'$ that are from LN_1 (and the same for LN_2). In this case, the operation passed to continuation F is (β, c''') for some configuration c''' where $c''' \stackrel{l'}{\twoheadrightarrow}_N c''$ is the remaining portion of $c \stackrel{l}{\twoheadrightarrow}_N c'$ that was not stepped through.

In general, we only want our simulator to step through γ-labeled firings if they are necessary to reach a particular event-bag-labeled firing (which the simulator will then step through). This conservative approach prevents our simulator from unnecessarily stepping through γ-labeled firings to dead end configurations from which there are no feasible operations. However, it remains possible that our simulator might reach a dead end via an event-bag-labeled firing.

Given that an **nth-bag** function's completeness requires its input continuation F to be both δ-consistent and n-surjective, and since an **nth-bag** function might be used within a continuation itself (as in the case for the parallel composition of two lazy Petri nets), we further require that the **nth-bag**$_{LN}$ function of a lazy Petri net LN is itself δ-consistent and n-surjective as

follows:

SPECIFICATION (Nth-Bag δ-Consistency)**.**

$$\forall c, \sigma, b, F, n,\ \Big(\forall \beta, c,\ \delta\text{-}\mathit{cons}\big(\lambda n.\ F(\beta, c, n)\big)\Big) \implies \delta\text{-}\mathit{cons}\big(\lambda n.\ \textbf{nth-bag}_{LN}(c, \sigma, b, F, n)\big)$$

SPECIFICATION (Nth-Bag N-Surjectivity)**.**

$$\forall c, \sigma, b, F, n,\ \Big(\forall \beta, c,\ \delta\text{-}\mathit{cons}\big(\lambda n.\ F(\beta, c, n)\big)\ \wedge\ n\text{-}\mathit{surj}\big(\lambda n.\ F(\beta, c, n)\big)\Big) \implies n\text{-}\mathit{surj}\big(\lambda n.\ \textbf{nth-bag}_{LN}(c, \sigma, b, F, n)\big)$$

Intuitively, $\textbf{nth-bag}_{LN}(c, \sigma, b, F, n)$ must preserve δ-consistency and n-surjectivity.

4.2.2 OPERATION-SEARCH FUNCTION

Given lazy Petri net $LN = (\mathcal{P}, c_{\mathit{init}}, \textbf{efc-fin}, \textbf{nth-bag})$, the operation-search function $\textbf{nth-op}_{LN}$ uses **efc-fin** and **nth-bag** to find a suspension-free feasible operation from some configuration. More specifically, given a configuration c that is reachable in LN's corresponding non-lazy Petri net, a state σ, and a natural number n, $\textbf{nth-op}_{LN}(c, \sigma, n)$ outputs either `term` (which indicates that LN's corresponding non-lazy Petri net can terminate by transitioning from c to its final configuration), $\texttt{emit}(\beta, c')$ (where the Petri net can generate feasible event-bag β by transitioning from c to configuration c'), or `none` (which indicates that there is no suspension-free feasible operation that can be taken from c). The formal definition of $\textbf{nth-op}_{LN}$ is shown below.

$$
\begin{aligned}
\textbf{nth-op}_{(\mathcal{P},c_{init},\textbf{efc-fin},\textbf{nth-bag})}(c,\sigma,n) \triangleq\ & \texttt{let } F := \big(\lambda(\beta,c,n).\ (\beta,c,\texttt{if } n=0 \texttt{ then } \mu \texttt{ else } n-1\big) \texttt{ in} \\
& \texttt{if } \textbf{efc-fin}(c) \\
& \texttt{then if } n=0 \\
& \qquad \texttt{then term} \\
& \qquad \texttt{else match } \textbf{nth-bag}(c,\sigma,\texttt{true},F,n-1) \texttt{ with} \\
& \qquad\quad (\beta,c',m) \mapsto \texttt{emit}(\beta,c') \\
& \qquad\quad \delta \mapsto \texttt{term} \\
& \texttt{else match } \textbf{nth-bag}(c,\sigma,\texttt{true},F,n) \texttt{ with} \\
& \qquad (\beta,c',m) \mapsto \texttt{emit}(\beta,c') \\
& \qquad \delta \mapsto \texttt{none}
\end{aligned}
$$

Given the definition above and a lazy Petri net LN that correctly corresponds to non-lazy Petri net $N = (\mathcal{P}, c_{init}, c_{fin}, \mathcal{T})$, the mechanically-proven correctness properties of the $\textbf{nth-op}_{LN}$ function are shown below.

LEMMA (Nth-Op Termination Soundness)**.**

$$\forall c,\sigma,n,\ \mathit{reach}_N(c)\ \wedge\ \textbf{nth-op}_{LN}(c,\sigma,n) = \texttt{term}\implies \exists l,\ c \xrightarrow{l}\!\!\!\!\rightarrow_N c_{fin}$$

LEMMA (Nth-Op Termination Completeness)**.**

$$\forall c,\sigma,l,\ \mathit{reach}_N(c)\ \wedge\ c \xrightarrow{l}\!\!\!\!\rightarrow_N c_{fin}\implies \textbf{nth-op}_{LN}(c,\sigma,0) = \texttt{term}$$

LEMMA (Nth-Op Generation Soundness)**.**

$$\begin{aligned}\forall c,\sigma,n,\beta,c',\ \mathit{reach}_N(c)\ \wedge\ \textbf{nth-op}_{LN}(c,\sigma,n) = \texttt{emit}(\beta,c')\implies\\ \exists l,c'',\ \mathit{feas}_\sigma(\beta)\ \wedge\ c \xrightarrow{l}\!\!\!\!\rightarrow_N c''\ \wedge\ c'' \xrightarrow{\beta}_N c'\end{aligned}$$

LEMMA (Nth-Op Generation Completeness)**.**

$$\begin{aligned}\forall c,\sigma,l,c',\beta,c'',\ \mathit{reach}_N(c)\ \wedge \mathit{feas}_\sigma(\beta)\ \wedge\ c \xrightarrow{l}\!\!\!\!\rightarrow_N c'\ \wedge\ c' \xrightarrow{\beta}_N c''\implies\\ \exists c',l',n,\ \mathit{reach}_N(c')\ \wedge\ c' \xrightarrow{l'}\!\!\!\!\rightarrow_N c''\ \wedge\ \textbf{nth-op}_{LN}(c,\sigma,n) = \texttt{emit}(\beta,c')\end{aligned}$$

4.2.3 TOP-LEVEL SIMULATION FUNCTION

Given a CHP program P, our top-level simulation function $\mathbf{sim}_P$ builds P's corresponding lazy Petri net $\mathbf{lazy}(P)$ (which is proven to correctly correspond to the non-lazy Petri net $\mathbf{constr}(P)$) and simulates P by using the $\mathbf{nth\text{-}op}_{\mathbf{lazy}(P)}$ function to find successive feasible operations of $\mathbf{constr}(P)$. The $\mathbf{sim}_P$ function uses each of the natural numbers in its infinite input stream i to determine which feasible operation $\mathbf{nth\text{-}op}_{\mathbf{lazy}(P)}$ should pick at each step of simulation.

Since the $\mathbf{nth\text{-}op}_{\mathbf{lazy}(P)}$ function has the possibility of returning `none`, the $\mathbf{sim}_P$ function has the possibility of reaching a dead end in $\mathbf{constr}(P)$, i.e. a configuration from which there are no suspension-free feasible operations (including no erroneous operations). It is undecidable to answer whether or not an arbitrary infinite sequence i will lead to a dead end in $\mathbf{constr}(P)$ for an arbitrary program P. This prevents us from declaring $\mathbf{sim}_P$'s return type to be something simple like $\mathit{Trace} \uplus \mathit{DeadEnd}$. Instead, $\mathbf{sim}_P$ returns a new coinductive type, a *simtrace*, which effectively models a (possibly-infinite) trace or a finite trace prefix followed by the indication of a dead end. A simtrace is formally defined coinductively as ϵ_s (the empty simtrace), δ_s (the indication of a dead end), or an event-bag followed by a simtrace.

In order to reason about the simtrace result of $\mathbf{sim}_P$ in terms of traces, we define a coinductive relation $\equiv$ between simtraces and traces as follows:

$$\frac{}{\epsilon_s \equiv \epsilon} \qquad \frac{t_s \equiv t}{\beta t_s \equiv \beta t}$$

Note that any simtrace containing δ_s has no trace equivalent.

Finally, the formal definition of $\mathbf{sim}_P$ and its mechanically-proven properties are shown below. The $\mathbf{sim}_P$ function operates according to its corecursive subfunction $\mathbf{sim}'$. Given a lazy Petri net LN, the $\mathbf{sim}'_{LN}$ function takes as input a configuration that is reachable in LN's

corresponding non-lazy Petri net, a state, and an infinite sequence of natural numbers, and it outputs a simtrace.

$$\mathbf{sim}_P(i) \triangleq \mathbf{sim}'_{\mathbf{lazy}(P)}(c_{init}, \sigma_0, i), \qquad \text{where } \mathbf{lazy}(P) = (\mathcal{P}, c_{init}, \mathbf{efc\text{-}fin}, \mathbf{nth\text{-}bag})$$

$$\begin{aligned}\mathbf{sim}'_{LN}(c, \sigma, i) \triangleq\ & \texttt{match } \mathbf{nth\text{-}op}_{LN}(c, \sigma, \texttt{head}(i)) \texttt{ with} \\ & \texttt{term} \mapsto \epsilon_s \\ & \texttt{emit}(\beta, c') \mapsto \beta\big(\mathbf{sim}'_{LN}(c', \sigma[\beta], \texttt{tail}(i))\big) \\ & \texttt{none} \mapsto \delta_s\end{aligned}$$

THEOREM (Simulation Soundness)**.** $\forall P, i, t,\ \mathbf{sim}_P(i) \equiv t \implies t \in [\![P]\!]$

THEOREM (Simulation Completeness)**.** $DC \implies \forall t, P,\ t \in [\![P]\!] \implies \exists i, \mathbf{sim}_P(i) \equiv t$

Note that we provide no formal guarantees about a CHP simulation that encounters a dead end. Our simulator might be modified to backtrack in such cases. It might try to find an alternative path of firings through the Petri net such that they produce the same simtrace prefix that has been constructed thus far but do not reach a dead end. We leave this and other features to future work.

5 SIMULATION EXAMPLES

We evaluate our certified CHP simulator on a FIFO buffer and the first asynchronous microprocessor [13]. We preface the program syntax and simulation runs of each with what we call a CHP *environment*. An environment consists of a closed set of data variables, data expressions, channels, and any other elements used in the syntax. Our tool and its proofs are parameterized across environments which differ from program to program, yet recall that in all cases we restrict the type of values, data variables, data expressions, and channels to natural numbers.

Since our simulator outputs a possibly-infinite simtrace, we build a simtrace observer function, denoted **observe**, that takes as input a natural number n and a simtrace t_s, and outputs an n-length finite trace prefix of t_s followed by `Eps` (which indicates that t_s has ended with ϵ_s), `More` (which indicates that there are more event-bags of t_s to observe), or `Dead` (which indicates that t_s has ended with δ). Further, the function **observe** modifies each event in the finite trace prefix to additionally display the result of each state evaluation made during its execution. For example, the trace prefix $\langle x{\leftarrow}(1+2)\rangle\langle A!{\uparrow}x, A?{\uparrow}\rangle\langle A!{\downarrow}, A?{\downarrow}y\rangle$ becomes $\langle x{\leftarrow}(1+2)(3)\rangle\langle A!{\uparrow}x(3), A?{\uparrow}\rangle\langle A!{\downarrow}, A?{\downarrow}y(3)\rangle$.

Recall that our simulator takes an infinite sequence of natural numbers as an input. Let **const**(n) denote the infinite sequence of repeating the natural number n.

5.1 A FIFO BUFFER

A FIFO (first in first out) buffer simply forwards data through a sequence of channels. In order to exemplify a terminating CHP program, we define our FIFO buffer to forward only three pieces of data (natural numbers 1, 2, and 3) through three channels (A, B, and C). The CHP environment and program syntax for our FIFO buffer are shown in Figure 5.1.

$$\begin{aligned} \text{value } k &::= 0 \mid 1 \mid 2 \mid \ldots \\ \text{data variable } v &::= x \mid y \mid z \\ \text{data expression } e &::= k \mid v \\ \text{channel } X &::= A \mid B \mid C \end{aligned}$$

$$\begin{aligned} \text{BFR} \triangleq\ & (A!(1)\,;\ A!(2)\,;\ A!(3)) \\ & \parallel (A?(x)\,;\ B!(x)\,;\ A?(x)\,;\ B!(x)\,;\ A?(x)\,;\ B!(x)) \\ & \parallel (B?(y)\,;\ C!(y)\,;\ B?(y)\,;\ C!(y)\,;\ B?(y)\,;\ C!(y)) \\ & \parallel (C?(z)\,;\ C?(z)\,;\ C?(z)) \end{aligned}$$

Figure 5.1: CHP environment and program syntax for our FIFO buffer

We simulate our FIFO buffer with two different inputs: $\mathbf{const}(0)$ and $\mathbf{const}(100)$, where 100 is just an arbitrary large-enough number to ensure that all feasible events are searched through at each simulation step. The results of observing the first 100 event-bags of these simulations are shown below, where 100 is just an arbitrary large-enough number to ensure that the entire output simtrace is observed.

$\mathbf{observe}(100, \mathbf{sim}_{\text{BFR}}(\mathbf{const}(0))) =$
$\langle A!\uparrow 1(1)\rangle\ \langle A?\uparrow\rangle\ \langle A!\downarrow, A?\downarrow x(1)\rangle$
$\langle A!\uparrow 2(2)\rangle\ \langle B!\uparrow x(1)\rangle\ \langle B?\uparrow\rangle\ \langle B!\downarrow, B?\downarrow y(1)\rangle\ \langle A?\uparrow\rangle\ \langle A!\downarrow, A?\downarrow x(2)\rangle$
$\langle A!\uparrow 3(3)\rangle\ \langle B!\uparrow x(2)\rangle\ \langle C!\uparrow y(1)\rangle\ \langle C?\uparrow\rangle\ \langle C!\downarrow, C?\downarrow z(1)\rangle\ \langle B?\uparrow\rangle\ \langle B!\downarrow, B?\downarrow y(2)\rangle\ \langle A?\uparrow\rangle\ \langle A!\downarrow, A?\downarrow x(3)\rangle$
$\langle B!\uparrow x(3)\rangle\ \langle C!\uparrow y(2)\rangle\ \langle C?\uparrow\rangle\ \langle C!\downarrow, C?\downarrow z(2)\rangle\ \langle B?\uparrow\rangle\ \langle B!\downarrow, B?\downarrow y(3)\rangle$
$\langle C!\uparrow y(3)\rangle\ \langle C?\uparrow\rangle\ \langle C!\downarrow, C?\downarrow z(3)\rangle$ `Eps`

$\mathbf{observe}(100, \mathbf{sim}_{\mathrm{BFR}}(\mathbf{const}(100))) =$
$\langle A!\uparrow 1(1), A?\uparrow, B?\uparrow, C?\uparrow\rangle\ \langle A!\downarrow, A?\downarrow x(1)\rangle$
$\langle A!\uparrow 2(2), B!\uparrow x(1)\rangle\ \langle B!\downarrow, B?\downarrow y(1)\rangle\ \langle A?\uparrow, C!\uparrow y(1)\rangle\ \langle A!\downarrow, A?\downarrow x(2), C!\downarrow, C?\downarrow z((1))\rangle$
$\langle A!\uparrow 3(3), B!\uparrow x(2), B?\uparrow, C?\uparrow\rangle\ \langle B!\downarrow, B?\downarrow y(2)\rangle\ \langle A?\uparrow, C!\uparrow y(2)\rangle\ \langle A!\downarrow, A?\downarrow x(3), C!\downarrow, C?\downarrow z((2))\rangle$
$\langle B!\uparrow x(3), B?\uparrow, C?\uparrow\rangle\ \langle B!\downarrow, B?\downarrow y(3)\rangle$
$\langle C!\uparrow y(3)\rangle\ \langle C!\downarrow, C?\downarrow z((3))\rangle$ `Eps`

We see that the former effectively simulates the buffer such that the "left-most" feasible event is executed at each step, while the latter does so such that as many feasible events as possible are executed at each step.

5.2 THE FIRST ASYNCHRONOUS MICROPROCESSOR

The first asynchronous microprocessor [13] was developed with CHP and is a well-studied exemplar of the language's semantics. Our formalization of the microprocessor differs from the original by some simplifications we make that do not take away significant complexity from the design, by our explicit implementation of abstract elements in the design, and by our core vs. full CHP equivalence discussed in Section 2.2.4. We list each of these differences in the presentation of our formalization below.

The microprocessor operates by iteratively reading and executing instructions stored in its instruction-memory array $imem$. The current location in the $imem$ is stored in its program-counter variable *pc*. Depending on the instruction at each iteration, the microprocessor performs the appropriate execution on its set of registers $r_0, r_1, \ldots r_n$, its ALU-flags variable *f*, its data-memory array $dmem$, and its *offset* variable.

The six types of instructions are: ALU, memory, memory-offset, branch, jump, and store-pc. These instruction types and the execution of each are shown in Figure 5.2.

type	structure	inst	op	execution
ALU	$op\ x\ y\ z$	add	0	$r_z := r_x + r_y \parallel f := 0$
		sub	1	$r_z := r_x - r_y \parallel f :=$ `if` $r_x < r_y$ `then` 1 `else` 0
memory	$op\ x\ y\ z$	ld	2	$r_z := dmem[r_x + r_y]$
		st	3	$dmem[r_x + r_y] := r_z$
memory-offset	$op - y\ z$ *offset*	ldx	4	$r_z := dmem[r_y + offset]$
		stx	5	$dmem[r_y + offset] := r_z$
		lda	6	$r_z := r_y + offset$
branch	$op - - cc$ *offset*	$brch$	7	`if` $cond(f, cc)$ `then` $pc := pc + offset$ `else skip`
jump	$op - y -$	jmp	8	$pc := r_y$
store-pc	$op - - z$	$stpc$	9	$r_z := pc$

Figure 5.2: Our instruction set for the first asynchronous microprocessor

This instruction set differs from the original as follows:

- Each of an instruction's four fields is a single 0-9 digit rather than a sequence of four Booleans.
- The original multi-assignment of r_z and f during each ALU instruction has been transformed into two parallel single-assignments.
- ALU instructions are limited to natural-number addition and subtraction.
- ALU addition always sets the ALU-flags variable f to 0.
- ALU subtraction sets the ALU-flags variable f to 1 if the subtraction results in underflow. Otherwise, f is set to 0.
- The guard $cond(f, cc)$ returns `true` iff f or cc is non-zero.

Given the microprocessor's instruction set, we present its CHP environment in Figure 5.3. This environment takes as input a function $imem$ of type $\mathbb{N} \to \mathbb{N}$, which models the microprocessor's (fixed) instruction-memory array.

value	k	$::= 0 \mid 1 \mid 2 \mid \ldots$
functions	$imem[k]$	$\triangleq \ldots$
	$k\%4$	$\triangleq k \,\%\, 4$
	$k.op$	$\triangleq k/1000 \,\%\, 10$
	$k.x$	$\triangleq k/100 \,\%\, 10$
	$k.y$	$\triangleq k/10 \,\%\, 10$
	$k.z$	$\triangleq k \,\%\, 10$
	$k.cc$	$\triangleq k \,\%\, 10$
	$aluz(k_1, k_2, k_3)$	$\triangleq$ `if` $k_3 = add$ `then` $k_1 + k_2$ `else if` $k_3 = sub$ `then` $k_1 - k_2$ `else` 0
	$aluf(k_1, k_2, k_3)$	$\triangleq$ `if` $k_3 = sub$ `then if` $k_1 < k_2$ `then` 1 `else` 0 `else` 0
data variable	v	$::=$ *pc* | *i* | *offset* | *pc′* | *i′* | *f* | *f′* | *aop* | *ax* | *ay* | *az* | *mx* | *my* | *ma* | *mw* | *dmem*[0] | *dmem*[1] | *dmem*[2] | *dmem*[3] | b_0 | b_1 | b_2 | b_3 | r_0 | r_1 | r_2 | r_3 | *ignE2* | *ignPc* | *ignXof* | *ignMC* | *ignXs* | *ignYs* | *ignZWs* | *ignZAs* | *ignZRs*
data expression	e	$::= k \mid v \mid e_1 + e_2 \mid imem[v] \mid e\%4 \mid v.op \mid v.x \mid v.y \mid v.z \mid v.cc \mid aluz(v_1, v_2, v_3) \mid aluf(v_1, v_2, v_3)$
channel	A	$::=$ *ID* | *PCI1* | *PCI2* | *PCA1* | *PCA2* | *Xpc* | *Ypc* | *Xof* | *E1* | *E2* | *X* | *Y* | *Xs* | *Ys* | *AC* | *MC1* | *MC2* | *MC3* | *ZAs* | *ZRs* | *ZWs* | *F* | *ZA* | *ZM* | *MDl* | *MDs*

Figure 5.3: CHP environment for the first asynchronous microprocessor

Given the microprocessor's environment, its top-level program syntax is shown below, where the syntax for each of its subprograms is found in Figures 5.4 and 5.5.

$$\begin{aligned} FAM \triangleq\ & IMEM \parallel FETCH \parallel PCADD \parallel EXEC \parallel ALU \parallel MU \parallel DMEM \\ & \parallel REG_0 \parallel REG_1 \parallel REG_2 \parallel REG_3 \end{aligned}$$

This syntax differs from the original as follows:

- We limit the data memory $dmem$ to four addresses and explicitly treat each address as an individual data variable. Further, we preface access to $dmem$ by case analysis of $i\%4$ for any given natural-number index i.

- We reduce the number of registers from sixteen to four, and modify access to each register by case analysis of $i\%4$ for any given natural-number identifier i.

- We explicitly initialize each of the following data variables to 0: pc, f, $dmem[0]$, $dmem[1]$, $dmem[2]$, $dmem[3]$, b_0, b_1, b_2, b_3, r_0, r_1, r_2, and r_3.

- We modify each of the following originally-dataless channels to send value 0 and receive into a new (ignored) data variable: $E2$, $PCI1$, $PCI2$, $PCA1$, $PCA2$, Xpc, Ypc, Xof, $MC1$, $MC2$, $MC3$, Xs, Ys, ZWs, ZAs, and ZRs.

- We transform all single- and many-guarded selection programs into those with two guarded subprograms each.

We first simulate the microprocessor with $imem[k] \triangleq 0$ for all k, which indicates that the microprocessor should repeatedly perform the ALU add execution $r_0 := r_0 + r_0 \parallel f := 0$. The results of observing the first 100 event-bags of simulations with this instruction memory and with respective inputs **const**(0) and **const**(2) are shown below.

$\mathbf{observe}(100, \mathbf{sim}_{\mathrm{FAM}}(\mathbf{const}(0))) =$

$\langle[\mathtt{true}]\rangle$

$\langle[\mathtt{true}]\rangle\ \langle \mathit{PCI1}!{\uparrow}0(0)\rangle$

$\langle \mathit{pc}{\leftarrow}0(0)\rangle\ \langle[\mathtt{true}]\rangle\ \langle[\overline{\mathit{PCI1}}(\mathtt{true}) \vee \overline{\mathit{PCA1}}(\mathtt{false})]\rangle\ \langle[\overline{\mathit{PCI1}}(\mathtt{true})]\rangle\ \langle \mathit{PCI1}?{\uparrow}\rangle\ \langle[\mathtt{true}]\rangle$

$\langle \mathit{PCI1}!{\downarrow}, \mathit{PCI1}?{\downarrow}\mathit{ignPc}(0)\rangle$

$\langle \mathit{ID}?{\uparrow}\rangle\ \langle[\widehat{\overline{\mathit{ID}}}(\mathtt{true})]\rangle\ \langle \mathit{ID}!{\uparrow}\mathit{imem}[\mathit{pc}](0)\rangle\ \langle \mathit{ID}!{\downarrow}, \mathit{ID}?{\downarrow}\mathit{i}(0)\rangle\ \langle[\mathtt{true}]\rangle$

$\langle \mathit{PCI2}!{\uparrow}0(0)\rangle\ \langle \mathit{pc}'{\leftarrow}(\mathit{pc}+1)(1)\rangle\ \langle \mathit{PCI2}?{\uparrow}\rangle\ \langle \mathit{PCI2}!{\downarrow}, \mathit{PCI2}?{\downarrow}\mathit{ignPc}(0)\rangle$

$\langle[\neg(\mathit{i}.\mathit{op}(0) = 4(4) \vee \mathit{i}.\mathit{op}(0) = 5(5) \vee \mathit{i}.\mathit{op}(0) = 6(6) \vee \mathit{i}.\mathit{op}(0) = 9(9))]\rangle\ \langle \mathit{E1}!{\uparrow}\mathit{i}(0)\rangle$

$\langle \mathit{pc}{\leftarrow}\mathit{pc}'(1)\rangle\ \langle[\mathtt{true}]\rangle$

$\langle[\mathtt{true}]\rangle\ \langle \mathit{E1}?{\uparrow}\rangle\ \langle \mathit{E1}!{\downarrow}, \mathit{E1}?{\downarrow}\mathit{i}'(0)\rangle\ \langle \mathit{E2}?{\uparrow}\rangle$

$\langle[(\mathit{i}'.\mathit{op}(0) = 0(0) \vee \mathit{i}'.\mathit{op}(0) = 1(1)) \vee \mathit{i}'.\mathit{op}(0) = 2(2) \vee \mathit{i}'.\mathit{op}(0) = 3(3)]\rangle\ \langle \mathit{E2}!{\uparrow}0(0)\rangle$

$\langle \mathit{E2}?{\downarrow}\mathit{ignE2}(0), \mathit{E2}!{\downarrow}\rangle$

$\langle[\mathtt{true}]\rangle\ \langle \mathit{PCI1}!{\uparrow}0(0)\rangle\ \langle[\overline{\mathit{PCI1}}(\mathtt{true}) \vee \overline{\mathit{PCA1}}(\mathtt{false})]\rangle\ \langle[\overline{\mathit{PCI1}}(\mathtt{true})]\rangle\ \langle \mathit{PCI1}?{\uparrow}\rangle$

$\langle \mathit{PCI1}!{\downarrow}, \mathit{PCI1}?{\downarrow}\mathit{ignPc}(0)\rangle$

$\langle \mathit{ID}?{\uparrow}\rangle\ \langle[\widehat{\overline{\mathit{ID}}}(\mathtt{true})]\rangle\ \langle \mathit{ID}!{\uparrow}\mathit{imem}[\mathit{pc}](0)\rangle\ \langle \mathit{ID}!{\downarrow}, \mathit{ID}?{\downarrow}\mathit{i}(0)\rangle\ \langle[\mathtt{true}]\rangle$

$\langle \mathit{PCI2}!{\uparrow}0(0)\rangle\ \langle \mathit{pc}'{\leftarrow}(\mathit{pc}+1)(2)\rangle\ \langle \mathit{PCI2}?{\uparrow}\rangle\ \langle \mathit{PCI2}!{\downarrow}, \mathit{PCI2}?{\downarrow}\mathit{ignPc}(0)\rangle$

$\langle[\neg(\mathit{i}.\mathit{op}(0) = 4(4) \vee \mathit{i}.\mathit{op}(0) = 5(5) \vee \mathit{i}.\mathit{op}(0) = 6(6) \vee \mathit{i}.\mathit{op}(0) = 9(9))]\rangle\ \langle \mathit{E1}!{\uparrow}\mathit{i}(0)\rangle$

$\langle \mathit{pc}{\leftarrow}\mathit{pc}'(2)\rangle\ \langle[\mathtt{true}]\rangle$

$\langle[\mathit{i}'.\mathit{op}(0) = 0(0) \vee \mathit{i}'.\mathit{op}(0) = 1(1)]\rangle\ \langle \mathrm{Xs}!{\uparrow}0(0)\rangle\ \langle \mathrm{Ys}!{\uparrow}0(0)\rangle\ \langle \mathrm{AC}!{\uparrow}\mathit{i}'.\mathit{op}(0)\rangle\ \langle \mathrm{ZAs}!{\uparrow}0(0)\rangle$

$\langle \mathit{f}{\leftarrow}0(0)\rangle\ \langle[\mathtt{true}]\rangle\ \langle[\overline{\mathrm{AC}}(\mathtt{true})]\rangle\ \langle \mathrm{AC}?{\uparrow}\rangle\ \langle \mathrm{X}?{\uparrow}\rangle\ \langle \mathrm{Y}?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle$

$\langle \mathit{dmem}[0]{\leftarrow}0(0)\rangle\ \langle \mathit{dmem}[1]{\leftarrow}0(0)\rangle\ \langle \mathit{dmem}[2]{\leftarrow}0(0)\rangle\ \langle \mathit{dmem}[3]{\leftarrow}0(0)\rangle\ \langle[\mathtt{true}]\rangle$

$\langle \mathit{b}_0{\leftarrow}0(0)\rangle\ \langle \mathit{r}_0{\leftarrow}0(0)\rangle$

$\langle[\mathtt{true}]\rangle\ \langle[\mathit{b}_0(0) = 0(0) \wedge 0(0) = \mathit{i}'.x\%4(0) \wedge \overline{\mathrm{Xs}}(\mathtt{true})]\rangle\ \langle \mathrm{X}!{\uparrow}\mathit{r}_0(0)\rangle\ \langle \mathrm{Xs}?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle\ \langle[\mathit{b}_0(0) = 0(0) \wedge 0(0) = \mathit{i}'.y\%4(0) \wedge \overline{\mathrm{Ys}}(\mathtt{true})]\rangle\ \langle \mathrm{Y}!{\uparrow}\mathit{r}_0(0)\rangle\ \langle \mathrm{Ys}?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle$

$\langle[\mathtt{true}]\rangle\ \langle[\mathit{b}_0(0) = 0(0) \wedge 0(0) = \mathit{i}'.z\%4(0) \wedge \overline{\mathrm{ZAs}}(\mathtt{true})]\rangle\ \langle \mathit{b}_0{\leftarrow}1(1)\rangle\ \langle \mathrm{ZAs}?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle$

$\langle \mathrm{Xs}!{\downarrow}, \mathrm{Ys}!{\downarrow}, \mathrm{AC}!{\downarrow}, \mathrm{ZAs}!{\downarrow}, \mathrm{AC}?{\downarrow}\mathit{aop}(0), \mathrm{X}?{\downarrow}\mathit{ax}(0), \mathrm{Y}?{\downarrow}\mathit{ay}(0), \mathrm{X}!{\downarrow}, \mathrm{Xs}?{\downarrow}\mathit{ign}\mathrm{Xs}(0), \mathrm{Y}!{\downarrow},$
$\mathrm{Ys}?{\downarrow}\mathit{ign}\mathrm{Ys}(0), \mathrm{ZAs}?{\downarrow}\mathit{ign}\mathrm{ZAs}(0)\rangle$

$\langle[\mathtt{true}]\rangle\ \langle \mathit{E1}?{\uparrow}\rangle\ \langle \mathit{E1}!{\downarrow}, \mathit{E1}?{\downarrow}\mathit{i}'(0)\rangle\ \langle \mathit{E2}?{\uparrow}\rangle$

$\langle[(\mathit{i}'.\mathit{op}(0) = 0(0) \vee \mathit{i}'.\mathit{op}(0) = 1(1)) \vee \mathit{i}'.\mathit{op}(0) = 2(2) \vee \mathit{i}'.\mathit{op}(0) = 3(3)]\rangle\ \langle \mathit{E2}!{\uparrow}0(0)\rangle$

$\langle \mathit{E2}?{\downarrow}\mathit{ignE2}(0), \mathit{E2}!{\downarrow}\rangle$

$\langle[\mathtt{true}]\rangle\ \langle \mathit{PCI1}!{\uparrow}0(0)\rangle\ \langle[\overline{\mathit{PCI1}}(\mathtt{true}) \vee \overline{\mathit{PCA1}}(\mathtt{false})]\rangle\ \langle[\overline{\mathit{PCI1}}(\mathtt{true})]\rangle\ \langle \mathit{PCI1}?{\uparrow}\rangle$

$\langle \mathit{PCI1}!{\downarrow}, \mathit{PCI1}?{\downarrow}\mathit{ignPc}(0)\rangle$

$\langle \mathit{ID}?{\uparrow}\rangle\ \langle[\widehat{\overline{\mathit{ID}}}(\mathtt{true})]\rangle\ \langle \mathit{ID}!{\uparrow}\mathit{imem}[\mathit{pc}](0)\rangle\ \langle \mathit{ID}!{\downarrow}, \mathit{ID}?{\downarrow}\mathit{i}(0)\rangle\ \mathtt{More}$

(with $\mathit{imem}[k] \triangleq 0$ for all k)

$\mathbf{observe}(100, \mathbf{sim}_{\mathrm{FAM}}(\mathbf{const}(2))) =$

$\langle[\mathtt{true}], [\mathtt{true}]\rangle$

$\langle PCI1!{\uparrow}0(0), pc{\leftarrow}0(0)\rangle\ \langle[\overline{PCI1}(\mathtt{true}) \vee \overline{PCA1}(\mathtt{false})], [\mathtt{true}]\rangle$

$\langle[\overline{PCI1}(\mathtt{true})], E1?{\uparrow}\rangle\ \langle PCI1?{\uparrow}, f{\leftarrow}0(0)\rangle\ \langle PCI1!{\downarrow}, PCI1?{\downarrow}ignPc(0), [\mathtt{true}]\rangle$

$\langle ID?{\uparrow}, pc'{\leftarrow}(pc+1)(1)\rangle\ \langle[\widehat{ID}(\mathtt{true})], PCI2?{\uparrow}\rangle$

$\langle ID!{\uparrow}imem[pc](0), [\mathtt{true}]\rangle\ \langle ID!{\downarrow}, ID?{\downarrow}i(0), dmem[0]{\leftarrow}0(0)\rangle$

$\langle[\mathtt{true}], PCI2!{\uparrow}0(0)\rangle\ \langle PCI2!{\downarrow}, PCI2?{\downarrow}ignPc(0), dmem[1]{\leftarrow}0(0)\rangle$

$\langle[\neg(i.op(0) = 4(4) \vee i.op(0) = 5(5) \vee i.op(0) = 6(6) \vee i.op(0) = 9(9))], pc{\leftarrow}pc'(1)\rangle$

$\langle E1!{\uparrow}i(0), [\mathtt{true}]\rangle\ \langle E1!{\downarrow}, E1?{\downarrow}i'(0), dmem[2]{\leftarrow}0(0)\rangle$

$\langle E2?{\uparrow}, [(i'.op(0) = 0(0) \vee i'.op(0) = 1(1)) \vee i'.op(0) = 2(2) \vee i'.op(0) = 3(3)]\rangle$

$\langle E2!{\uparrow}0(0), dmem[3]{\leftarrow}0(0)\rangle\ \langle E2?{\downarrow}ignE2(0), E2!{\downarrow}, [\mathtt{true}]\rangle$

$\langle[\mathtt{true}], [i'.op(0) = 0(0) \vee i'.op(0) = 1(1)]\rangle\ \langle Ys!{\uparrow}0(0)\rangle\ \langle AC!{\uparrow}i'.op(0)\rangle\ \langle ZAs!{\uparrow}0(0)\rangle$

$\langle PCI1!{\uparrow}0(0), Xs!{\uparrow}0(0)\rangle\ \langle[\overline{PCI1}(\mathtt{true}) \vee \overline{PCA1}(\mathtt{false})], [\overline{AC}(\mathtt{true})]\rangle\ \langle X?{\uparrow}\rangle\ \langle Y?{\uparrow}\rangle$

$\langle[\overline{PCI1}(\mathtt{true})], AC?{\uparrow}\rangle\ \langle PCI1?{\uparrow}, b_0{\leftarrow}0(0)\rangle\ \langle PCI1!{\downarrow}, PCI1?{\downarrow}ignPc(0), r_0{\leftarrow}0(0)\rangle$

$\langle ID?{\uparrow}, pc'{\leftarrow}(pc+1)(2)\rangle\ \langle[\widehat{ID}(\mathtt{true})], PCI2?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle\ \langle[b_0(0) = 0(0) \wedge 0(0) = i'.y\%4(0) \wedge \overline{Ys}(\mathtt{true})]\rangle\ \langle Y!{\uparrow}r_0(0)\rangle\ \langle Ys?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle$

$\langle[\mathtt{true}]\rangle\ \langle[b_0(0) = 0(0) \wedge 0(0) = i'.z\%4(0) \wedge \overline{ZAs}(\mathtt{true})]\rangle\ \langle b_0{\leftarrow}1(1)\rangle\ \langle ZAs?{\uparrow}\rangle$

$\langle[\mathtt{true}]\rangle$

$\langle ID!{\uparrow}imem[pc](0), [\mathtt{true}]\rangle\ \langle ID!{\downarrow}, ID?{\downarrow}i(0), b_1{\leftarrow}0(0)\rangle$

$\langle[\mathtt{true}], PCI2!{\uparrow}0(0)\rangle\ \langle PCI2!{\downarrow}, PCI2?{\downarrow}ignPc(0), r_1{\leftarrow}0(0)\rangle$

$\langle[\neg(i.op(0) = 4(4) \vee i.op(0) = 5(5) \vee i.op(0) = 6(6) \vee i.op(0) = 9(9))], pc{\leftarrow}pc'(2)\rangle$

$\langle E1!{\uparrow}i(0), [\mathtt{true}]\rangle\ \langle[\mathtt{true}], [\mathtt{true}]\rangle\ \langle[\mathtt{true}], [\mathtt{true}]\rangle\ \langle[\mathtt{true}], b_2{\leftarrow}0(0)\rangle\ \langle r_2{\leftarrow}0(0), b_3{\leftarrow}0(0)\rangle$

$\langle[\mathtt{true}], [\mathtt{true}]\rangle\ \langle[\mathtt{true}], [\mathtt{true}]\rangle\ \langle[\mathtt{true}], r_3{\leftarrow}0(0)\rangle\ \langle[\mathtt{true}], [\mathtt{true}]\rangle\ \langle[\mathtt{true}], [\mathtt{true}]\rangle\ \langle[\mathtt{true}]\rangle\ \mathtt{Dead}$

(with $imem[k] \triangleq 0$ for all k)

We see that the latter simulation encounters a dead end. This is because of a read/write error on r_0 at the microprocessor-instruction level. In this case, the write-lock variable b_0 is set to 1 while the program waits for a value to be written to r_0. At the same time, the program is waiting for b_0 to become 0 so that it can read the value in r_0 in order to write a value back to r_0. This deadlock is not encountered in the former simulation because the microprocessor reads from r_0 *before* setting the write lock. The microprocessor paper [13] discusses that the refinement made to prevent such deadlock was omitted from the presented design. And

although our CHP simulator provides no formal guarantees about its simulations that reach a dead end, we show that it is able to simulate this case.

We now simulate the microprocessor with the instruction memory shown below, which implements the computation of the Fibonacci sequence.

$$
\begin{array}{lll}
imem[k] \triangleq & \texttt{if } k = 0 \texttt{ then } 6002 & (r_2 := r_0 + 1) \\
& \texttt{else if } k = 1 \texttt{ then } 0001 & \\
& \texttt{else if } k = 2 \texttt{ then } 0123 & (r_3 := r_1 + r_2) \\
& \texttt{else if } k = 3 \texttt{ then } 3003 & (dmem[r_0 + r_0] := r_3) \\
& \texttt{else if } k = 4 \texttt{ then } 0021 & (r_1 := r_0 + r_2) \\
& \texttt{else if } k = 5 \texttt{ then } 0032 & (r_2 := r_0 + r_3) \\
& \texttt{else if } k = 6 \texttt{ then } 6003 & (r_3 := r_0 + 2) \\
& \texttt{else if } k = 7 \texttt{ then } 0002 & \\
& \texttt{else if } k = 8 \texttt{ then } 8030 & (pc := r_3) \\
& \texttt{else } 0 &
\end{array}
$$

The instructions encoded above use r_0, r_1, r_2, r_3, *offset*, and *pc* to write the successive Fibonacci numbers to $dmem[0]$. Note that r_0 remains zero throughout the program execution. The microprocessor paper [13] actually mentions that r_0 is special in that it always contains the value zero, but this specification was not made explicit in their presented design.

The result of observing the first 5000 event-bags of the simulation with the instruction memory above and with input **const**(0) is shown below, where we only display the contents of those event-bags that assign to or receive into r_1, r_2, or $dmem[0]$.

$$\mathbf{observe}(5000, \mathbf{sim}_{\mathrm{FAM}}(\mathbf{const}(0))) =$$
$$\ldots \langle dmem[0] \leftarrow 0(0) \rangle \ldots \langle r_1 \leftarrow 0(0) \rangle \ldots \langle r_2 \leftarrow 0(0) \rangle \ldots \langle \mathrm{ZM!}\downarrow, \mathrm{ZM?}\downarrow r_2(1) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](1) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(1) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(1) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](2) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(1) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(2) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](3) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(2) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(3) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](5) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(3) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(5) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](8) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(5) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(8) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](13) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(8) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(13) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](21) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(13) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(21) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](34) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(21) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(34) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](55) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(34) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(55) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](89) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(55) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(89) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](144) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(89) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(144) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](233) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(144) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(233) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](377) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(233) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(377) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](610) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_1(377) \rangle \ldots \langle \mathrm{ZA!}\downarrow, \mathrm{ZA?}\downarrow r_2(610) \rangle \ldots$$
$$\langle \mathrm{MDs!}\downarrow, \mathrm{MDs?}\downarrow dmem[0](987) \rangle \ldots \texttt{More}$$
(with $imem$ defined above)

We see that the microprocessor simulation correctly writes the successive Fibonacci numbers to $dmem[0]$.

$$
\begin{aligned}
IMEM \triangleq\ & *[\texttt{true} \rightarrow [\widehat{ID} \rightarrow ID!(imem[pc]) \\
& \quad \mid \texttt{false} \rightarrow \texttt{skip}]]
\end{aligned}
$$

$$
\begin{aligned}
offset'(e) \triangleq\ & e = ldx \vee e = stx \vee e = lda \vee e = brch \\
FETCH \triangleq\ & *[\texttt{true} \rightarrow PCI1!(0)\,;\ ID?(i)\,;\ PCI2!(0)\,; \\
& \quad [offset'(i.op) \rightarrow PCI1!(0)\,;\ ID?(offset)\,;\ PCI2!(0) \\
& \quad [] \neg offset'(i.op) \rightarrow \texttt{skip}]\,; \\
& \quad E1!(i)\,;\ E2?(ignE2)]
\end{aligned}
$$

$$
\begin{aligned}
PCADD \triangleq\ & pc := 0\,; \\
& (*[\texttt{true} \rightarrow [\overline{PCI1} \vee \overline{PCA1} \rightarrow \\
& \quad [\overline{PCI1} \rightarrow PCI1?(ignPc)\,;\ pc' := pc + 1\,; \\
& \quad\quad PCI2?(ignPc)\,;\ pc := pc' \\
& \quad [] \overline{PCA1} \rightarrow PCA1?(ignPc)\,;\ pc' := pc + offset\,; \\
& \quad\quad PCA2?(ignPc)\,;\ pc := pc' \\
& \quad [] \overline{Xpc} \vee \overline{Ypc} \rightarrow [\overline{Xpc} \rightarrow X!(pc) \bullet Xpc?(ignPc) \\
& \quad\quad [] \overline{Ypc} \rightarrow Y?(pc) \bullet Ypc?(ignPc)]]] \\
& \quad \| *[\texttt{true} \rightarrow [\overline{Xof} \rightarrow X!(offset) \bullet Xof?(ignXof) \\
& \quad\quad \mid \texttt{false} \rightarrow \texttt{skip}]])
\end{aligned}
$$

$$
\begin{aligned}
alu(e) \triangleq\ & e = add \vee e = sub \\
EXECa \triangleq\ & [alu(i'.op) \rightarrow Xs!(0) \bullet Ys!(0) \bullet AC!(i'.op) \bullet ZAs!(0) \\
& \quad [] i'.op = ld \vee i'.op = st \rightarrow \\
& \quad\quad [i'.op = ld \rightarrow Xs!(0) \bullet Ys!(0) \bullet MC1!(0) \bullet ZRs!(0) \\
& \quad\quad [] i'.op = st \rightarrow Xs!(0) \bullet Ys!(0) \bullet MC2!(0) \bullet ZWs!(0)]] \\
EXECb1 \triangleq\ & [i'.op = ldx \vee i'.op = stx \rightarrow \\
& \quad\quad [i'.op = ldx \rightarrow Xof!(0) \bullet Ys!(0) \bullet MC1!(0) \bullet ZRs!(0) \\
& \quad\quad [] i'.op = stx \rightarrow Xof!(0) \bullet Ys!(0) \bullet MC2!(0) \bullet ZWs!(0)]] \\
& \quad [] i'.op = lda \rightarrow Xof!(0) \bullet Ys!(0) \bullet MC3!(0) \bullet ZRs!(0)] \\
cond(e_1, e_2) \triangleq\ & \neg(e_1 = 0) \vee \neg(e_2 = 0) \\
EXECb2 \triangleq\ & [i'.op = stpc \vee i'.op = jmp \rightarrow \\
& \quad\quad [i'.op = stpc \rightarrow Xpc!(0) \bullet Ys!(0) \bullet AC!(add) \bullet ZAs!(0) \\
& \quad\quad [] i'.op = jmp \rightarrow Ypc!(0) \bullet Ys!(0)]] \\
& \quad [] i'.op = brch \rightarrow F?(f')\,; \\
& \quad\quad\quad [cond(f', i'.cc) \rightarrow PCA1!(0)\,;\ PCA2!(0) \\
& \quad\quad\quad [] \neg cond(f', i'.cc) \rightarrow \texttt{skip}]] \\
EXECb \triangleq\ & [i'.op = ldx \vee i'.op = stx \vee i'.op = lda \rightarrow EXECb1 \\
& \quad [] i'.op = stpc \vee i'.op = jmp \vee i'.op = brch \rightarrow EXECb2] \\
EXEC \triangleq\ & *[\texttt{true} \rightarrow E1?(i')\,; \\
& \quad [alu(i'.op) \vee i'.op = ld \vee i'.op = st \rightarrow E2!(0)\,;\ EXECa \\
& \quad [] i'.op = ldx \vee i'.op = stx \vee i'.op = lda \vee \\
& \quad\quad i'.op = stpc \vee i'.op = jmp \vee i'.op = brch \rightarrow \\
& \quad\quad\quad EXECb\,;\ E2!(0)]]
\end{aligned}
$$

Figure 5.4: CHP syntax for the first asynchronous microprocessor, first part

$$
\begin{array}{l}
\mathrm{ALU} \triangleq f := 0\,; \\
\quad *[\mathtt{true} \to [\overline{\mathrm{AC}} \to \mathrm{AC}?(aop) \bullet \mathrm{X}?(ax) \bullet \mathrm{Y}?(ay)\,; \\
\qquad\qquad (az := aluz(ax, ay, aop) \parallel f := aluf(ax, ay, aop))\,; \mathrm{ZA}!(az) \\
\qquad [\!] \widehat{\mathrm{F}} \to \mathrm{F}!(f)]] \\
\\
\mathrm{MU} \triangleq *[\mathtt{true} \to [\overline{\mathrm{MC1}} \vee \overline{\mathrm{MC2}} \to \\
\qquad [\overline{\mathrm{MC1}} \to \mathrm{X}?(mx) \bullet \mathrm{Y}?(my) \bullet \mathrm{MC1}?(ign\mathrm{MC})\,; \\
\qquad\quad ma := mx + my\,; \mathrm{MDl}?(mw)\,; \mathrm{ZM}!(mw) \\
\qquad [\!] \overline{\mathrm{MC2}} \to \mathrm{X}?(mx) \bullet \mathrm{Y}?(my) \bullet \mathrm{MC2}?(ign\mathrm{MC}) \bullet \mathrm{ZM}?(mw)\,; \\
\qquad\quad ma := mx + my\,; \mathrm{MDs}!(mw)] \\
\qquad [\!] \overline{\mathrm{MC3}} \to \mathrm{X}?(mx) \bullet \mathrm{Y}?(my) \bullet \mathrm{MC3}?(ign\mathrm{MC})\,; \\
\qquad\quad ma := mx + my\,; \mathrm{ZM}!(ma)]] \\
\\
\mathrm{DMEM} \triangleq dmem[0] := 0\,;\ dmem[1] := 0\,;\ dmem[2] := 0\,;\ dmem[3] := 0\,; \\
\quad *[\mathtt{true} \to [\widehat{\mathrm{MDl}} \to [ma\%4 = 0 \vee ma\%4 = 1 \to \\
\qquad\qquad [ma\%4 = 0 \to \mathrm{MDl}!(dmem[0]) \\
\qquad\qquad [\!] ma\%4 = 1 \to \mathrm{MDl}!(dmem[1])] \\
\qquad\quad [\!] ma\%4 = 2 \vee ma\%4 = 3 \to \\
\qquad\qquad [ma\%4 = 2 \to \mathrm{MDl}!(dmem[2]) \\
\qquad\qquad [\!] ma\%4 = 3 \to \mathrm{MDl}!(dmem[3])] \\
\qquad [\!] \overline{\mathrm{MDs}} \to [ma\%4 = 0 \vee ma\%4 = 1 \to \\
\qquad\qquad [ma\%4 = 0 \to \mathrm{MDs}?(dmem[0]) \\
\qquad\qquad [\!] ma\%4 = 1 \to \mathrm{MDs}?(dmem[1])] \\
\qquad\quad [\!] ma\%4 = 2 \vee ma\%4 = 3 \to \\
\qquad\qquad [ma\%4 = 2 \to \mathrm{MDs}?(dmem[2]) \\
\qquad\qquad [\!] ma\%4 = 3 \to \mathrm{MDs}?(dmem[3])] \\
\\
\mathrm{REG}_k \triangleq b_k := 0\,;\ r_k = 0\,; \\
\quad (*[\mathtt{true} \to [b_k = 0 \wedge k = (i'.x)\%4 \wedge \overline{\mathrm{Xs}} \to \\
\qquad\qquad \mathrm{X}!(r_k) \bullet \mathrm{Xs}?(ign\mathrm{Xs}) \\
\qquad\quad | \mathtt{false} \to \mathtt{skip}]] \\
\qquad \parallel *[\mathtt{true} \to [b_k = 0 \wedge k = (i'.y)\%4 \wedge \overline{\mathrm{Ys}} \to \\
\qquad\qquad \mathrm{Y}!(r_k) \bullet \mathrm{Ys}?(ign\mathrm{Ys}) \\
\qquad\quad | \mathtt{false} \to \mathtt{skip}]] \\
\qquad \parallel *[\mathtt{true} \to [b_k = 0 \wedge k = (i'.z)\%4 \wedge \overline{\mathrm{ZWs}} \to \\
\qquad\qquad \mathrm{ZM}!(r_k) \bullet \mathrm{ZWs}?(ign\mathrm{ZWs}) \\
\qquad\quad | \mathtt{false} \to \mathtt{skip}]] \\
\qquad \parallel *[\mathtt{true} \to [b_k = 0 \wedge k = (i'.z)\%4 \wedge \overline{\mathrm{ZAs}} \to \\
\qquad\qquad b_k := 1\,; \mathrm{ZAs}?(ign\mathrm{ZAs})\,; \mathrm{ZA}?(r_k)\,; b_k := 0 \\
\qquad\quad | \mathtt{false} \to \mathtt{skip}]] \\
\qquad \parallel *[\mathtt{true} \to [b_k = 0 \wedge k = (i'.z)\%4 \wedge \overline{\mathrm{ZRs}} \to \\
\qquad\qquad b_k := 1\,; \mathrm{ZRs}?(ign\mathrm{ZRs})\,; \mathrm{ZM}?(r_k)\,; b_k := 0 \\
\qquad\quad | \mathtt{false} \to \mathtt{skip}]])
\end{array}
$$

Figure 5.5: CHP syntax for the first asynchronous microprocessor, second part

6 FUTURE WORK & CONCLUSION

We have formalized a novel mechanical semantics of CHP using possibly-infinite execution traces and we have built a CHP program simulator that is certified to operate correctly according to those semantics. Our simulator serves as a model for future CHP tools to be built and certified against the semantics. Such tools might include a CHP error-checker and a CHP program-comparison tool.

Future work for our CHP semantics includes formalizing the full CHP syntax, developing a CHP type system, introducing localized data variables and channels, and formalizing the notions of fairness and deadlock of a CHP program. That of our simulator includes checking for errors during simulation, providing some guarantee about simulations that encounter a dead end, implementing backtracking for when a dead end is encountered, and improving its performance and usability.

There is not yet a fully comprehensive and effective set of automation tools for developing asynchronous circuits at VLSI scale, and we are even farther from producing a *certified* set of such tools. In the meantime, it may be useful to *integrate* certified software into preexisting (non-certified) asynchronous-circuit development tools. For example, we might build a certified CHP-trace validator that is mechanically-proven to correctly answer whether or not a given trace is within the feasible behavior of a given CHP program. This certified tool might then be used alongside a non-certified CHP simulator in order to validate that the simulator has produced a correct trace. All in all, building certified software is challenging, but the

required effort may very well be worth the trust and correctness it establishes for complex systems like asynchronous-circuit development tools.

REFERENCES

1. Joseph L. Bates and Robert L. Constable. Proofs as programs. In ACM *Transactions on Programming Languages and Systems* (TOPLAS), vol. 7(1), pp. 53–71, 1985.

2. The Coq Development Team. The Coq proof assistant reference manual, version 8.10.1. Oct. 2019: https://coq.inria.fr/.

3. Thomas Braibant. Coquet: A Coq library for verifying hardware. In *Proceedings of the International Conference on Certified Programs and Proofs*, pp. 330–345, 2011.

4. Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In *Proceedings of the International Conference on Computer Aided Verification* (CAV), pp. 213–228, 2013.

5. Muralidaran Vijayaraghavan, Adam Chlipala, and Nirav Dave. Modular deductive verification of multiprocessor hardware designs. In *Proceedings of the International Conference on Computer Aided Verification* (CAV), pp. 109–127, 2015.

6. Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, and Adam Chlipala. Kami: A platform for high-level parametric hardware specification and its modular verification. In *Proceedings of the* ACM SIGPLAN *International Conference on Functional Programming* (ICFP), 2017.

7. Xavier Leroy. Formal verification of a realistic compiler. In *Communications of the* ACM, vol. 52(7), pp. 107–115, 2009.

8. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In ACM *SIGPLAN Notices*, vol. 46(6), pp. 283–294, 2011.

9. Basit Riaz Sheikh and Rajit Manohar. An asynchronous floating-point multiplier. In *Proceedings of the IEEE Symposium on Asynchronous Circuits and Systems* (ASYNC), 2012.

10. Basit Riaz Sheikh and Rajit Manohar. An operand-optimized asynchronous IEEE 754 double-precision floating-point adder. In *Proceedings of the IEEE Symposium on Asynchronous Circuits and Systems* (ASYNC), 2010.

11. David Fang, John Teifel, and Rajit Manohar. A high-performance asynchronous FPGA: Test results. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM), 2005.

12. Benjamin Tang, Stephen Longfield, Sunil Bhave, and Rajit Manohar. A low power asynchronous GPS baseband processor. In *Proceedings of the IEEE Symposium on Asynchronous Circuits and Systems* (ASYNC), 2012.

13. Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri V. Cummings, and Tak-Kwan Lee. The design of an asynchronous MIPS R3000 microprocessor. In *Proceedings of the Conference on Advanced Research in VLSI* (ARVLSI), pp. 164–181, 1997.

14. Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. In *Distributed Computing*, vol. 1(4), pp. 226–234, 1986.

15. Alain J. Martin. A synthesis method for self-timed VLSI circuits. In *Proceedings of the IEEE International Conference on Computer Design* (ICCD), pp. 224–229, 1987.

17. Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In *Proceedings of the MIT Conference on Advanced Research in VLSI*, pp. 35–40, 1988.

16. Alain J. Martin. The probe: An addition to communication primitives. In *Information Processing Letters*, vol. 20(3), pp. 125–130, 1985.

18. Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the Conference on European Design Automation*, pp. 384–389, 1991.

19. Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pp. 437–440, 1996.

20. Dominique Borrione, Menouer Boubekeur, Laurent Mounier, Marc Renaudin, and Antoine Siriani. Validation of asynchronous circuit specifications using IF/CADP. In *VLSI-SOC: From Systems to Chips*, pp. 85–100, 2006.

21. Hubert Garavel, Gwen Salaün, and Wendelin Serwe. On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP. In *Science of Computer Programming*, vol. 74(3), pp. 100–127, 2009.

22. Stephen Longfield, Brittany Nkounkou, Rajit Manohar, and Ross Tate. Preventing glitches and short circuits in high-level self-timed chip specifications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

23. Scott F. Smith and Amy E. Zwarico. Provably correct synthesis of asynchronous circuits. In *Designing Correct Circuits*, vol. 5, pp. 237–260, 1992.

24. Marcel Rene Van der Goot. Semantics of VLSI synthesis. PhD Thesis, *California Institute of Technology*, 1995.

25. Ralph-Johan R. Back, Alain J. Martin, and Kaisa Sere. An action system specification of the Caltech asynchronous microprocessor. In *Proceedings of the International Conference on Mathematics of Program Construction*, pp. 159–179, 1995.

26. Mark Bickford and Robert L. Constable. A causal logic of events in formalized computational type theory. Technical Report, *Cornell University Department of Computer Science*, 2005.

27. Alain J. Martin and Christopher D. Moore. CHP and CHPsim: A language and simulator for fine-grain distributed computation. Technical Report, *California Institute of Technology Department of Computer Science*, 2011.

28. Paul Brunet, Damien Pous, and Georg Struth. On decidability of concurrent Kleene algebra. In *Proceedings of the International Conference on Concurrency Theory (*CONCUR), 2017.

29. Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pp. 413–443, 1978.

30. Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pp. 166–175, 1978.

31. Michael Mendler and Terry Stroup. Newtonian arbiters cannot be proven correct. In *Formal Methods in System Design*, vol. 3(3), pp. 233–257, 1993.