The claim that extended regular expressions (EREs) and deterministic finite automata (DFAs) are equivalent in expressive power is accurate. Both formalisms can represent exactly the class of regular languages. EREs include additional operators like **`+`** (one or more occurrences) and **`?`** (zero or one occurrence), which provide syntactic convenience but do not increase the expressive power beyond that of classical regular expressions. Every ERE can be converted into an equivalent DFA, and vice versa, meaning they recognize the same set of languages. This equivalence is well-established in formal language theory.

EXERCISE 0F-2: SET THEORY

Demonstrate the correspondence between A and B by presenting an appropriate function and proving that it is a bijection. For example, you might construct a function f : B → A and prove that f is an injection and a surjection.

1. Understanding the Problem

We are tasked with demonstrating the 1-1 correspondence (bijection) between two sets:

A = X → P(Y): The set of functions from X → P(Y), where P(Y) is the powerset of Y,

B = P(X × Y): the powerset of X × Y, or the set of all subsets of X × Y.

To show this correspondence, we define a function: $f: B \to A$, and prove that it is both:

1. **Injective**: Every element in B maps to a unique element in A.
2. **Surjective**: Every element in A has a pre-image in B.

2. Solution

Step 1: Define the Function f : B → A

For any subset $S \in B$ i.e. , $S \subseteq X \times Y$, we define: $f(S): X \to P(Y)$.

Specifically, $f(S)(x) = \{y \in Y \mid (x, y) \in S\}$.

This means that $(f(S)$ maps each $x \in X$ to the subset of Y consisting of elements y such that $(x, y) \in S$.

- Intuitively, S represents a relation between elements of X and Y, and f(S) extracts the set of related y-values for each x.

Example:

Let:

$$X = \{a, b\}, Y = \{1,2\}, \text{and } S = \{(a, 1), (a, 2), (b, 1)\}.$$

Then:

$$f(S)(a) = \{1,2\}, f(S)(b) = \{1\}.$$

Thus $f(S)$ is the function $f(S): a \mapsto \{1,2\}, b \mapsto \{1\}$.

Step 2: Prove Injectivity of *f*

To prove injectivity, we must show:

$$f(S_1) = f(S_2) \Longrightarrow S_1 = S_2$$

Proof:

1. Assume $f(S_1) = f(S_2)$, where $S_1,\ S_2 \subseteq X \times Y$

2. For any pair $(x, y) \in S_1$

- By definition of $f, y \in f(S_1)(x)$.
- since $f(S_1) = f(S_2),\ y \in f(S_2)(x)$.
- Therefore $(x, y) \in S_2$.

3. Similarly, for any pair $(x, y) \in S_2$, We conclude $(x, y) \in S_1$.

4. Hence $S_1 = S_2$,

Thus f is injective.

Step 2: Prove Surjectivity of *f*

To prove surjectivity, we must show:

$$\forall g \in A, \exists S \in B \text{ such that } f(S) = g.$$

Proof:

1. Let: $g \in A$, where $g: X \to P(Y)$.

2. Define $S \subseteq X \times Y$ as

$$S = \{(x, y) \in X \times Y \mid y \in g(x)\}.$$

3. For each $x \in X$:

- By Construction $f(S)(x) = \{y \in Y \mid (x, y) \in S\}$.
- This simplifies to $f(S)(x) = g(x)$.

4. Therefore: $f(S) = g$.

Thus f is surjective.

Conclusion

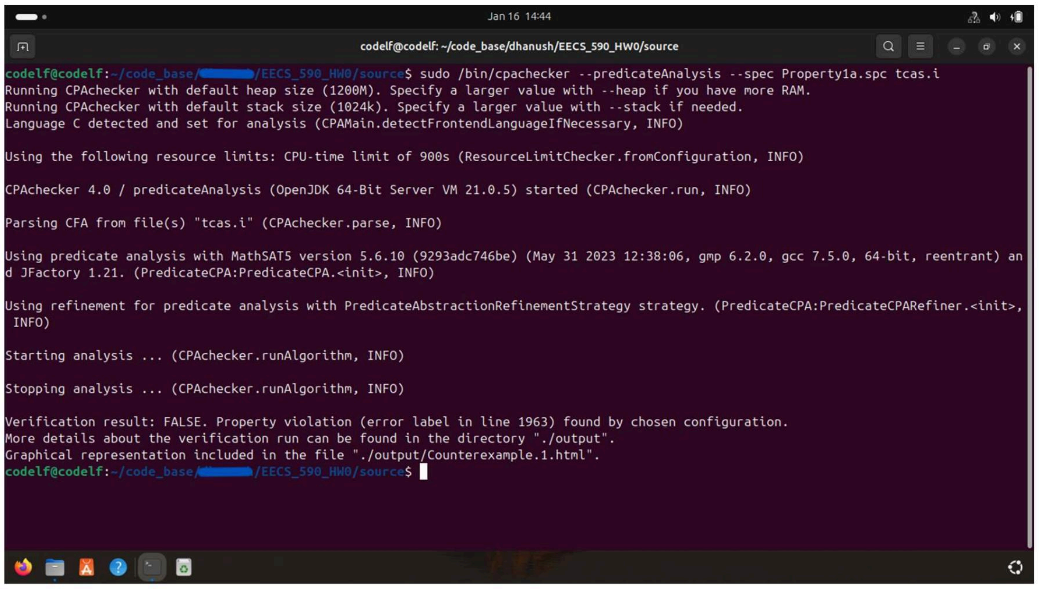
Since $f: B \to A$ is both injective and surjective, it is a bijection. This establishes the 1-1 correspondence between A and B.

EXERCISE 0F-3: MODEL CHECKING

Screenshots & Outputs

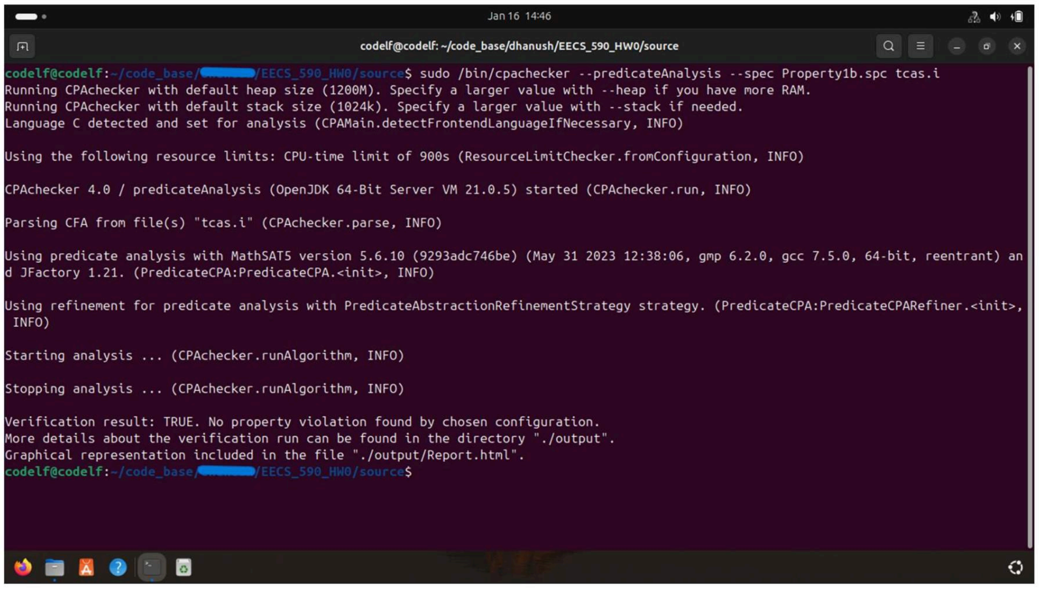
Below are the last ten non-empty lines of output for each property checked using CPAChecker:

Property1a Output



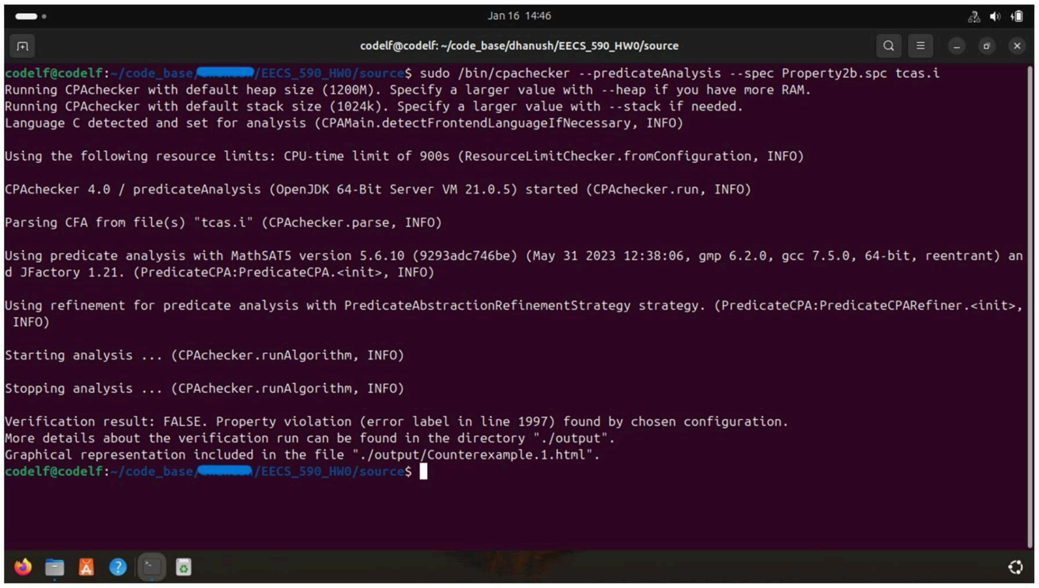
```
codelf@codelf:~/code_base/[redacted]/EECS_590_HW0/source$ sudo /bin/cpachecker --predicateAnalysis --spec Property1a.spc tcas.i
Running CPAchecker with default heap size (1200M). Specify a larger value with --heap if you have more RAM.
Running CPAchecker with default stack size (1024k). Specify a larger value with --stack if needed.
Language C detected and set for analysis (CPAMain.detectFrontendLanguageIfNecessary, INFO)

Using the following resource limits: CPU-time limit of 900s (ResourceLimitChecker.fromConfiguration, INFO)

CPAchecker 4.0 / predicateAnalysis (OpenJDK 64-Bit Server VM 21.0.5) started (CPAchecker.run, INFO)

Parsing CFA from file(s) "tcas.i" (CPAchecker.parse, INFO)

Using predicate analysis with MathSAT5 version 5.6.10 (9293adc746be) (May 31 2023 12:38:06, gmp 6.2.0, gcc 7.5.0, 64-bit, reentrant) an
d JFactory 1.21. (PredicateCPA:PredicateCPA.<init>, INFO)

Using refinement for predicate analysis with PredicateAbstractionRefinementStrategy strategy. (PredicateCPA:PredicateCPARefiner.<init>,
 INFO)

Starting analysis ... (CPAchecker.runAlgorithm, INFO)

Stopping analysis ... (CPAchecker.runAlgorithm, INFO)

Verification result: FALSE. Property violation (error label in line 1963) found by chosen configuration.
More details about the verification run can be found in the directory "./output".
Graphical representation included in the file "./output/Counterexample.1.html".
codelf@codelf:~/code_base/[redacted]/EECS_590_HW0/source$
```

Property1b Output



```
codelf@codelf:~/code_base/[redacted]/EECS_590_HW0/source$ sudo /bin/cpachecker --predicateAnalysis --spec Property1b.spc tcas.i
Running CPAchecker with default heap size (1200M). Specify a larger value with --heap if you have more RAM.
Running CPAchecker with default stack size (1024k). Specify a larger value with --stack if needed.
Language C detected and set for analysis (CPAMain.detectFrontendLanguageIfNecessary, INFO)

Using the following resource limits: CPU-time limit of 900s (ResourceLimitChecker.fromConfiguration, INFO)

CPAchecker 4.0 / predicateAnalysis (OpenJDK 64-Bit Server VM 21.0.5) started (CPAchecker.run, INFO)

Parsing CFA from file(s) "tcas.i" (CPAchecker.parse, INFO)

Using predicate analysis with MathSAT5 version 5.6.10 (9293adc746be) (May 31 2023 12:38:06, gmp 6.2.0, gcc 7.5.0, 64-bit, reentrant) an
d JFactory 1.21. (PredicateCPA:PredicateCPA.<init>, INFO)

Using refinement for predicate analysis with PredicateAbstractionRefinementStrategy strategy. (PredicateCPA:PredicateCPARefiner.<init>,
 INFO)

Starting analysis ... (CPAchecker.runAlgorithm, INFO)

Stopping analysis ... (CPAchecker.runAlgorithm, INFO)

Verification result: TRUE. No property violation found by chosen configuration.
More details about the verification run can be found in the directory "./output".
Graphical representation included in the file "./output/Report.html".
codelf@codelf:~/code_base/[redacted]/EECS_590_HW0/source$
```

Property2b Output



```
codelf@codelf: ~/code_base/dhanush/EECS_590_HW0/source
codelf@codelf:~/code_base/[redacted]/EECS_590_HW0/source$ sudo /bin/cpachecker --predicateAnalysis --spec Property2b.spc tcas.i
Running CPAchecker with default heap size (1200M). Specify a larger value with --heap if you have more RAM.
Running CPAchecker with default stack size (1024k). Specify a larger value with --stack if needed.
Language C detected and set for analysis (CPAMain.detectFrontendLanguageIfNecessary, INFO)

Using the following resource limits: CPU-time limit of 900s (ResourceLimitChecker.fromConfiguration, INFO)

CPAchecker 4.0 / predicateAnalysis (OpenJDK 64-Bit Server VM 21.0.5) started (CPAchecker.run, INFO)

Parsing CFA from file(s) "tcas.i" (CPAchecker.parse, INFO)

Using predicate analysis with MathSAT5 version 5.6.10 (9293adc746be) (May 31 2023 12:38:06, gmp 6.2.0, gcc 7.5.0, 64-bit, reentrant) an
d JFactory 1.21. (PredicateCPA:PredicateCPA.<init>, INFO)

Using refinement for predicate analysis with PredicateAbstractionRefinementStrategy strategy. (PredicateCPA:PredicateCPARefiner.<init>,
 INFO)

Starting analysis ... (CPAchecker.runAlgorithm, INFO)

Stopping analysis ... (CPAchecker.runAlgorithm, INFO)

Verification result: FALSE. Property violation (error label in line 1997) found by chosen configuration.
More details about the verification run can be found in the directory "./output".
Graphical representation included in the file "./output/Counterexample.1.html".
codelf@codelf:~/code_base/[redacted]/EECS_590_HW0/source$
```

Summary and Discussion

a. Understanding the Tool and Results

When running CPAChecker, the tool analyzes the control flow of the provided C program (tcas.i) against the specified properties (Property1a.spc, Property1b.spc, Property2b.spc) using predicate analysis. Each property represents a specific behavior or safety requirement the program must meet.

- **Property1a**: The tool reported a violation (FALSE) with an error label at line 1963, indicating that the program fails to meet this property under some conditions.
- **Property1b**: The tool verified this property as TRUE, meaning no violations were detected, and the program adheres to this safety constraint.
- **Property2b**: Similar to Property1a, a violation (FALSE) was reported at line 1997, highlighting another area where the program fails to satisfy the specified property.

The tcas.i file serves as a simplified test suite simulating a traffic collision avoidance system. While it effectively demonstrates basic verification tasks, its structure and scale might limit its applicability in real-world scenarios. These results establish the tool's capability to identify safety violations and provide detailed counterexamples for debugging.

b. Usability of CPAChecker

CPAChecker is a powerful yet intricate tool. Setting it up required installing dependencies and configuring a proper environment. Its input format is straightforward but assumes familiarity with formal verification concepts and the C programming language. The commands rely on specifying properties and the program to analyze, making it essential to understand the structure of the .spc files and the code being verified.

The output includes both console logs and graphical representations (e.g., Counterexample.1.html), which visualize counterexamples or proof steps, aiding in understanding and debugging violations. However, the graphical output could benefit from improved navigation and labeling to enhance user-friendliness.

c. Reflections and Validity Concerns

While CPAChecker provides rigorous verification guarantees, its limitations must be considered:

- **Scalability**: The performance may degrade with larger or more complex programs.
- **Assumptions**: The tool assumes the correctness of its analysis engine and external libraries like MathSAT5. Bugs in these dependencies could compromise results.
- **Environment-Specific Behavior**: Variability in results might occur due to system configurations or library versions, raising concerns about reproducibility in experimental setups.

CPAChecker is a robust verification tool for academic and research purposes, albeit with a steep learning curve and potential threats to validity in large-scale applications. Its outputs, while informative, demand careful interpretation to avoid misrepresentation.