Compiling Reactive State Machines via Kleene Algebra with Tests for UAV Precision Landing

Muhammad Akmal — 13524099
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha No. 10 Bandung
muhammad.akmal.3806@gmail.com, 13524099@std.stei.itb.ac.id

***Abstract*—Finite state machines govern mission-critical behavioral logic in embedded and reactive control systems, from UAV flight controllers to network protocol handlers. Despite their conceptual simplicity, translating state machine specifications into correct and verifiable implementations remains an ongoing source of defects: engineers write switch-case dispatchers by hand, without a mechanical correctness guarantee. We propose a formal compilation framework using Kleene Algebra with Tests (KAT), a two-sorted algebraic structure that consist of regular expressions with Boolean Algebra and is equipped with a complete and decidable equational theory. A state machine is encoded as a KAT expression where compilation reduces that expression to a target normal form whose correctness is certified by the KAT decision procedure. We develop two algorithmic compilation strategies rooted in classical paradigms: Divide and Conquer via state elimination (the McNaughton-Yamada algorithm), which decomposes the machine one state at a time using a Kleene-star label update recurrence; and Dynamic Programming via matrix Kleene closure (Warshall's algorithm generalized to the KAT semiring), which computes the complete reachability expression via a bottom-up recurrence over intermediate states. Both strategies are formally sound under the KAT equational theory and produce semantically equivalent compilations from a shared specification. We formalize an encoding scheme for reactive state machines with mutable Boolean flags and abstracted threshold predicates, and implement both strategies in a prototype compiler written in Rust, including a Brzozowski-derivative-based equivalence checker. The framework is validated on a six-state precision landing machine from a UAV competition system, comparing strategy outputs by normal form structure, expression complexity, generated code shape, and compilation overhead.**

***Index Terms*—Kleene algebra, Kleene algebra with test, KAT, finite state machines, FSM, control systems, UAV mission control, divide and conquer, dynamic programming, algorithm strategy**

I. Introduction

Finite state machines are among the most widely deployed abstractions in embedded and reactive control systems. UAV mission controllers, traffic signal managers, network protocol stacks, and industrial automation firmware all encode behavioral logic as states, guards, and transitions. In these domains, correctness is not merely desirable, considering an UAV that fails to exit a search state, or descends without a confirmed landing target, does not just produce a wrong output (it crashes).

Despite the maturity of state machine theory, the path from specification to implementation remains largely informal. Engineers transcribe state diagrams into switch-case dispatchers or nested conditional chains by hand, with no mechanical bridge between specification and code. The implementation is correct if the engineer is careful, and incorrect otherwise. Model-based tools such as Simulink/Stateflow [1] generate code from graphical specifications but carry no formal algebraic correctness guarante and only validated empirically. Formal verification tools exist but operate post-hoc, only checking a finished implementation rather than deriving one from a certified transformation.

We close this gap using Kleene Algebra with Tests (KAT) [2]. KAT generalizes regular expressions to programs with Boolean guards, equipped with a complete equational theory and a decidable equivalence problem: two KAT expressions denote the same computation if and only if they are equal under the KAT axioms, reducible to language equivalence of finite automata over guarded strings [2]. This provides a formal substrate for compilation: a state machine is encoded as a KAT expression, a compilation strategy rewrites that expression to a normal form, and the KAT decision procedure certifies semantic equivalence between input and output at each step.

The central thesis is that two classical algorithmic paradigms each correspond to a distinct, formally grounded compilation algorithm in the KAT setting. Divide and Conquer: the McNaughton-Yamada state elimination algorithm [3] decomposes the compilation problem by eliminating states one at a time, with each step a self-contained subproblem and the Kleene-star label update formula the conquering recurrence. Dynamic Programming: matrix Kleene closure [4] computes the full reachability expression by a bottom-up recurrence over intermediate states; the exact generalization of Floyd-Warshall to an arbitrary Kleene algebra semiring. Beyond the algorithmic duality, both strategies are instances of a common algebraic elimination scheme applied to the KAT transition matrix, distinguished only by the order in which elimination steps are applied. This algebraic unity provides a clean theoretical basis for comparing their properties.

This paper proposes a formal encoding of reactive state machines, including mutable Boolean flags and abstracted threshold predicates, into KAT expressions, with a soundness theorem relating the encoding to the concrete machine semantics. We also provide the formal development of three compilation strategies as KAT rewriting algorithms, with char-

acterization of their respective normal forms, applicable state machine classes, output code shapes, and complexity.

To check the soundness of our approach in real case, we also implement a prototype compiler in Rust implementing all three strategies, including a Brzozowski-derivative-based KAT equivalence checker. The compiler will be tested for a case study evaluation on the precision landing state machine of a real UAV competition system, comparing strategy outputs by expression complexity, code structure, and compilation overhead.

The remainder of this paper is organized as follows. Section II provides background on KAT and guarded strings along with existing case studies. Section III presents the state machine encoding. Section IV develops the three algorithmic strategies.

II. Theoretical Basis

A. Kleene Algebra and Regular Expressions

Kleene Algebra (KA) is an algebraic structure $(K, +, \cdot, ^*, 0, 1)$ that formalizes the standard equational axioms of regular expressions [5]. In the context of control flow and reactive systems, the elements of K represent transition paths, programs, or sets of behaviors. The fundamental operators define how these behaviors combine: the choice operator $+$ (representing nondeterministic branching or alternative state transitions) is commutative, associative, and idempotent with an identity element 0 (representing the empty behavior, deadlock, or abort). The sequential composition operator $\cdot$ (representing the sequencing of states or actions) is associative with an identity element 1 (representing the no-op or skip action) and annihilator 0, and it distributes over $+$.

The Kleene star operator * models unbounded iteration, making it the algebraic equivalent of looping in control graphs. It is uniquely characterized by the fixpoint and induction axioms:

$$1 + p \cdot p^* \leq p^*, \quad q + p \cdot r \leq r \;\Rightarrow\; p^* \cdot q \leq r$$

along with their mirror-image right-handed variants. These axioms formally capture the logic of while-loop unfolding and invariant induction. The relation $\leq$ establishes a natural partial order on K, defined by $p \leq q \iff p+q = q$. Operationally, $p \leq q$ means that the behavior p is subsumed by q, or that p represents a specialized execution path within the broader set of system behaviors q.

The canonical model for KA is the language model over a finite alphabet Σ of primitive actions. In this interpretation, $K = 2^{\Sigma^*}$ is the powerset of all finite strings over Σ, with $+$ interpreted as set union, $\cdot$ as string concatenation, and * as the standard Kleene closure. Under this model, a KA expression completely describes the set of all possible execution traces of a system.

A foundational result by Kozen [5] establishes the completeness of these axioms: two KA expressions denote the same regular language (i.e., they represent identical sets of execution traces) if and only if they are provably equal using only the KA algebraic axioms. This completeness theorem reduces the abstract problem of algebraic verification to the standard problem of deterministic finite automaton (DFA) language equivalence. Crucially for our compilation framework, this equivalence is decidable in PSPACE, providing a mechanically computable pathway to certify that two different representations of a state machine encode the exact same sequence of behaviors.

B. Kleene Algebra with Tests

Kleene Algebra with Tests (KAT) is a two-sorted algebraic structure $(K, B, +, \cdot, ^*, \bar{\cdot}, 0, 1)$ where $(K, +, \cdot, ^*, 0, 1)$ is a standard Kleene Algebra and $(B, +, \cdot, \bar{\cdot}, 0, 1)$ is a Boolean algebra embedded within K as a distinguished subalgebra. The sets satisfy $B \subseteq K$, and every element $b \in B$ possesses a Boolean complement $\bar{b}$ such that $b \cdot \bar{b} = 0$ (contradiction) and $b + \bar{b} = 1$ (excluded middle) [2].

The two sorts cleanly separate the checking of state from the mutation of state. Elements of B are *tests* (e.g., guard conditions, sensor thresholds), representing side-effect-free Boolean assertions about the current environment. Elements of $K \setminus B$ are *actions* or *programs*, representing operations that mutate the system state or perform I/O (e.g., executing a state transition, updating a mutable flag).

Crucially, because $B \subseteq K$, tests double as specialized programs that act as execution filters. When evaluated in a sequence, a test b does not change the state. Instead, if the current state satisfies b, the test acts as the multiplicative identity 1 (the *skip* or *continue* action), allowing subsequent computations to proceed. Conversely, if the state fails to satisfy b, the test acts as the multiplicative annihilator 0 (the *abort* or *halt* action), immediately terminating that specific execution path. This filtering mechanism allows KAT to elegantly model guarded transitions natively: the composite term $b \cdot p$ strictly dictates "assert b holds, and only then execute p."

From the KAT axioms, the standard structured programming constructs are derivable identities:

$$\texttt{if } b \texttt{ then } p \texttt{ else } q \quad \equiv \quad b \cdot p \;+\; \bar{b} \cdot q \tag{1}$$

$$\texttt{while } b \texttt{ do } p \quad \equiv \quad (b \cdot p)^* \cdot \bar{b} \tag{2}$$

These are equalities within the KAT theory, usable in equational proofs and rewriting. KAT subsumes propositional Hoare logic: any valid Hoare triple $\{b\}\, p\, \{c\}$ is expressible as the KAT inequality $b \cdot p \leq p \cdot c$, and the valid triples are exactly the KAT-provable ones [2].

C. Guarded Strings and the Decision Procedure

The canonical semantic model for KAT is the algebra of *guarded strings* [2]. Given a finite set of primitive tests Π_T and primitive programs Π_P, an *atom* α is a maximal consistent Boolean assignment to all tests in Π_T: for $|\Pi_T| = n$ there are 2^n atoms. A *guarded string* is an alternating sequence

$$\alpha_0\, p_1\, \alpha_1\, p_2\, \alpha_2\, \cdots\, p_k\, \alpha_k$$

where each α_i is an atom and each $p_i \in \Pi_P$. A guarded string describes a complete trace of a reactive system: observe the

world state (atom), take an action, observe the resulting world state, repeat.

A KAT expression e denotes a set of guarded strings $\mathcal{L}(e)$ under the extension: a test b denotes $\{\alpha : \alpha \models b\}$; a primitive p denotes $\{\alpha \cdot p \cdot \beta : \alpha, \beta \in \mathrm{Atoms}\}$; composition, choice, and star extend homomorphically, with composition requiring the terminal atom of the left operand to match the initial atom of the right.

The Completeness Theorem [2] states that $e =_{\mathrm{KAT}} f$ if and only if $\mathcal{L}(e) = \mathcal{L}(f)$. The decision procedure builds a finite automaton over guarded strings using *Brzozowski derivatives*. The derivative $\partial_{\alpha,p}(e)$ of e with respect to a step (α, p) gives what remains of e after consuming one $\alpha \cdot p$ prefix. The derivative rules are defined by induction on e:

$$\partial_{\alpha,p}(0) = 0, \quad \partial_{\alpha,p}(1) = 0, \quad \partial_{\alpha,p}(b) = 0$$

$$\partial_{\alpha,p}(q) = \begin{cases} 1 & \text{if } q = p \\ 0 & \text{otherwise} \end{cases}$$

$$\partial_{\alpha,p}(e + f) = \partial_{\alpha,p}(e) + \partial_{\alpha,p}(f)$$

$$\partial_{\alpha,p}(e \cdot f) = \partial_{\alpha,p}(e) \cdot f \; + \; \varepsilon_\alpha(e) \cdot \partial_{\alpha,p}(f)$$

$$\partial_{\alpha,p}(e^*) = \partial_{\alpha,p}(e) \cdot e^*,$$

$$\partial_{\alpha,p}(b \cdot e) = \begin{cases} \partial_{\alpha,p}(e) & \text{if } \alpha \models b \\ 0 & \text{otherwise} \end{cases}$$

where $\varepsilon_\alpha(e) \in \{0, 1\}$ checks whether e accepts an empty guarded string starting at atom α. A worklist-based automaton construction using these rules terminates because the number of inequivalent derivatives is finite (bounded by the completeness theorem). Two KAT expressions are equivalent iff their automata are bisimilar.

D. Related Work

1) NetKAT: Foster et al. [6] apply KAT to network packet forwarding, defining domain-specific semantics and a coalgebraic decision procedure for network policies. Our work targets reactive state machines in embedded control systems, a different domain with sequential control flow, mutable state, and code generation as the primary goal rather than policy verification.

2) KAT and Compiler Optimizations: Kozen and Patron [7] use KAT to certify the correctness of compiler optimizations such as copy propagation and dead code elimination, expressing each optimization as a KAT equation and verifying it via the equational theory. Our contribution is complementary: rather than verifying pre-existing transformations, we use KAT rewriting as the compilation algorithm itself, generating implementations from specifications with correctness guaranteed by construction.

3) Statecharts and Model-Based Code Generation: Harel's statecharts [1] and their commercial descendants (Simulink/Stateflow, SCADE) provide graphical state machine specification with automated code generation. These tools lack formal algebraic foundations for their code generators; correctness arguments are empirical or rely on separately conducted testing. Our approach provides algebraic correctness certificates.

4) Synchronous Languages and Certified Compilers: The synchronous programming family, notably languages like Esterel and Lustre [8], [9], provides formal semantics for reactive systems and serves as the theoretical foundation for industrial tools like SCADE. Recent advancements in this domain have yielded formally verified compilers (e.g., the Vélus compiler bridging Lustre to CompCert [10]), which guarantee semantic preservation from specification to executable code. However, these approaches rely on mechanized semantics and extensive manual proof engineering within interactive theorem provers like Coq. In contrast, our framework leverages the inherent decidability of the KAT equational theory, offering a lightweight, fully automated algebraic certification of the compilation steps without requiring a heavy-duty mechanized proof toolchain.

III. State Machine Model

A. Formal Model

We define a **reactive state machine (RSM)** as a tuple

$$\mathcal{M} = (Q,\, q_0,\, F,\, \Pi_P,\, \Pi_T^{\mathrm{ext}},\, \delta) \tag{3}$$

where

- Q is a finite set of control states with $|Q| = m$,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of terminal (accepting) states,
- Π_P is a finite set of primitive actions,
- Π_T^{ext} is a finite set of externally observable Boolean predicates, and
- $\delta \subseteq Q \times G(\Pi_T^{\mathrm{ext}}) \times \Pi_P \times Q$ is the transition relation, where $G(\Pi_T^{\mathrm{ext}})$ denotes Boolean combinations of predicates in Π_T^{ext}.

The running example throughout this paper is the **PrecisionLand** machine $\mathcal{M}_{\mathrm{PL}}$. States are indexed from s_0 to s_5 for *Idle*, *Search*, *Center*, *Descend*, *Finished*, and *Failure*; $q_0 = s_0$ and $F = \{s_4,\, s_5\}$.

The external predicate set is $\Pi_T^{\mathrm{ext}} = \{vs,\, tv,\, cx,\, blh,\, ld,\, to,\, wx\}$, where:

- vs: vision service reachable and mode-enable future resolved successfully
- tv: target position valid (landing pad in camera view)
- cx: UAV horizontally centered over target (position error $< 7\,\mathrm{cm}$)
- blh: LIDAR reading below landing threshold (`down_lidar_height` $<$ `param_land_height`; predicate abstraction of a real-valued comparison)
- ld: land-confirmed Boolean flag (mutable; handled via KAT+B!)
- to: target-lost timeout expired ($\Delta t_{\mathrm{target}} >$ `param_target_timeout_sec`)
- wx: search waypoints exhausted (`search_waypoint_index` $\geq |$`search_waypoints`$|$)

B. KAT Encoding

1) Atom Construction: Each control state $q_i \in Q$ induces a primitive test $s_i \in \Pi_T$ with the mutual exclusion invariant $s_i \cdot s_j = 0$ for $i \neq j$ and $\sum_i s_i = 1$. The full primitive

test set is $\Pi_T = \{s_1, \ldots, s_m\} \cup \Pi_T^{\text{ext}}$. Valid atoms are those satisfying the mutual exclusion constraint; there are $m \cdot 2^{|\Pi_T^{\text{ext}}|}$ valid atoms.

2) Transition Encoding: A transition $(q_i, g, p, q_j) \in \delta$ is encoded as the KAT expression $s_i \cdot g \cdot \tau_{ij}$, where $\tau_{ij} \in \Pi_P$ is a primitive action whose relational semantics maps the atom component from q_i to q_j (updating the state flag) while executing the associated side effect p. The full machine encoding defined as

$$E(\mathcal{M}) \;=\; \left(\sum_{(q_i,\, g,\, p,\, q_j) \,\in\, \delta} s_i \cdot g \cdot \tau_{ij} \right)^{*} \cdot \left(\sum_{q_f \in F} s_f \right) \quad (4)$$

The outer star iterates the machine until a terminal state is reached while the trailing sum asserts termination at an accepting state.

3) Soundness: The guarded string language $\mathcal{L}(E(\mathcal{M}))$ is in bijection with the accepting runs of $\mathcal{M}$: each guarded string $\alpha_0 \pi_{i_0 j_0} \alpha_1 \pi_{i_1 j_1} \cdots \alpha_k$ corresponds to a run $q_{i_0} \xrightarrow{g_0, a_0} q_{j_0} \xrightarrow{g_1, a_1} \cdots$ where each atom α_t records the control state q_{i_t} and the values of Π_T^{ext} at step t.

C. Mutable State Handling

Pure KAT treats tests as read-only, but in our model, RSMs in practice carry mutable state. We address two categories as follows.

1) Mutable Boolean Flags: Attributes such as `land_detected` are mutable Boolean variables that change as transitions execute. We handle these using the KAT+B! extension [11], which adds primitive assignment programs $b := 0$ and $b := 1$ to Π_P, with guarded string transformer semantics: $b := 1$ maps an atom α to the unique atom α' that agrees with α on all tests except b, where $\alpha' \models b$. Decidability is preserved for finite flag sets.

2) Continuous Threshold Predicates: Conditions such as `lidar_height < param_land_height` involve real-valued state. We apply a predicate abstraction where each threshold comparison is treated as an atomic Boolean test $t_i \in \Pi_T^{\text{ext}}$ with no internal arithmetic structure visible to KAT. The encoding reasons about control flow conditioned on these atomic tests; their concrete evaluation at runtime is delegated to the underlying system. This is sound because KAT reasoning is monotone with respect to the interpretation of atomic tests. The limitation that KAT cannot prove data-level properties such as "`below_land_threshold` eventually holds" is acknowledged; such properties require a separate data-plane analysis outside the scope of this work.

IV. Algorithmic Compilation Strategies

We present three compilation strategies for reducing $E(\mathcal{M})$ to a target normal form suitable for code emission. All three strategies are sound: the KAT equivalence checker verifies that each output is semantically equivalent to the input.

A. Divide and Conquer: State Elimination

1) Algorithm: Intuitively, to compile a machine with n states, pick one internal state and algebraically fold all paths through it into direct edges between the remaining states. This reduces the problem to a machine with $n-1$ states. Repeat until only the initial and terminal states remain. The McNaughton-Yamada-McCluskey state elimination algorithm [3] operates on the state machine as a labeled directed graph $G = (Q, \lambda)$ where $\lambda : Q \times Q \to \text{KAT}$ assigns each pair of states a KAT expression (initially, the sum of guard-action products for transitions between them, or 0 if none). The algorithm repeatedly selects an internal state $q_k \in Q \setminus (\{q_0\} \cup F)$ and performs the following update for all remaining q_i, q_j:

$$\lambda(q_i, q_j) \;\leftarrow\; \lambda(q_i, q_j) \;+\; \lambda(q_i, q_k) \cdot \lambda(q_k, q_k)^{*} \cdot \lambda(q_k, q_j) \quad (5)$$

State q_k is then removed from the graph. Repeating until only q_0 and states in F remain yields the KAT expression for the machine: $\lambda(q_0, q_f)$ for each $q_f \in F$. We can also represent this procedurally using the pseudocode in Algorithm 1.

Algorithm 1 State Elimination for RSM

Require: RSM $M = (Q, q_0, F, \delta)$
Ensure: KAT expression e such that $L(e) = L(E(M))$

for all $q_i, q_j \in Q$ **do**
  if $\exists (q_i, g, a, q_j) \in \delta$ **then**
    $\lambda[i][j] \leftarrow \sum_{\{(q_i, g, a, q_j) \in \delta\}} (g \cdot \pi_{ij})$
  else
    $\lambda[i][j] \leftarrow 0$
  end if
end for
Internal $\leftarrow Q \setminus (\{q_0\} \cup F)$
Sort Internal by increasing value of $|\text{in}(q_k)| \cdot |\text{out}(q_k)|$
for all $q_k \in$ Internal *in sorted order* **do**
  loop $\leftarrow \lambda[k][k]^{*}$
  for all $q_i \in Q \setminus \{q_k\}$ where $\lambda[i][k] \neq 0$ **do**
    for all $q_j \in Q \setminus \{q_k\}$ where $\lambda[k][j] \neq 0$ **do**
      $\lambda[i][j] \leftarrow \lambda[i][j] + (\lambda[i][k] \cdot \text{loop} \cdot \lambda[k][j])$
    end for
  end for
  for all $q_m \in Q$ **do**
    $\lambda[m][k] \leftarrow 0$
    $\lambda[k][m] \leftarrow 0$
  end for
end for
return $\sum_{q_f \in F} \lambda[\text{idx}(q_0)][\text{idx}(q_f)]$

2) Structure: Each elimination step decomposes the problem: a machine with n states is reduced to an equivalent machine with $n-1$ states and updated edge labels. The conquering step is the label update formula, which folds all paths through q_k into the direct edges between remaining states. The subproblem independence by eliminating q_k before or after q_j yields equivalent results gives the divide-and-conquer character.

The update formula is an application of the KAT star identity: paths from q_i to q_j through q_k are exactly $\lambda(i,k) \cdot \lambda(k,k)^* \cdot \lambda(k,j)$. Adding this to $\lambda(i,j)$ and removing q_k preserves the guarded string language of the machine, by induction on the number of eliminated states. After eliminating states $q_{k_1}, \ldots, q_{k_t}$, the residual labeled graph G_t satisfies $\mathcal{L}(E(G_t)) = \mathcal{L}(E(\mathcal{M}))$.

The elimination order affects intermediate expression size. We adopt the heuristic of ordering states by $\min\big(|\mathrm{in}(q_k)| \cdot |\mathrm{out}(q_k)|\big)$, eliminating states with the smallest in-degree-times-out-degree product first, which minimizes the number of new terms introduced per step. The output is a flat sum of products representing enumerated execution paths: well-suited to a flat switch-case dispatcher, where each case corresponds to one path from initial to terminal state. The algorithm has $O(n^3)$ edge update operations. Each intermediate label can grow in size; in the worst case (adversarial graph structure), expression sizes grow exponentially with the number of states, but for the sparse, low-fan-in machines typical of control logic, growth is polynomial in practice.

B. Dynamic Programming: Matrix Kleene Closure

1) Algorithm: For this algorithm, we try an approach by representing the machine as a transition matrix over KAT then computing its Kleene closure using a recurrence over intermediate states, exactly as Floyd-Warshall computes all-pairs reachability, but over the KAT semiring instead of the Boolean or tropical semiring. More formally, we represent $\mathcal{M}$ as an $m \times m$ matrix M over KAT where

$$M[i][j] \;=\; \sum_{(q_i,\, g,\, p,\, q_j) \,\in\, \delta} g \cdot \tau_{ij}$$

(or 0 if no transition exists from q_i to q_j). The Kleene closure $M^*[i][j]$ (the KAT expression for all paths from q_i to q_j) is computed by the recurrence relation

$$\begin{aligned} D^{(0)} &= M \\ D^{(k)}[i][j] &= D^{(k-1)}[i][j] + \\ &\qquad D^{(k-1)}[i][k] \cdot (D^{(k-1)}[k][k])^* \cdot D^{(k-1)}[k][j] \end{aligned}$$

for $k = 1, \ldots, m$, with $D^{(m)} = M^*$. The KAT expression for the full machine is $\sum_{q_f \in F} D^{(m)}[0][f]$. Computationally, we can write the pseudocode for this approach in Algorithm 2.

2) Structure: $D^{(k)}[i][j]$ represents the KAT expression for all paths from q_i to q_j using only intermediate states $q_1, \ldots, q_k$. Optimal substructure holds because KAT composition is associative and distributes over path concatenation. This is the exact generalization of Floyd-Warshall's all-pairs shortest-path algorithm [4] from the Boolean semiring to an arbitrary Kleene algebra, which is the algebraic connection Kleene identified in the semiring treatment of reachability.

The output is a closed-form KAT expression per source-target pair, with Kleene stars encoding loops through intermediate states. This form is compact for machines with complex loop structure and maps naturally to a transition table implementation, where the table cell $T[i][j]$ holds the compiled action sequence for the path. Complexity is strictly $O(n^3)$ matrix update operations, regardless of graph structure. Each operation involves one KAT multiplication and one KAT addition. Unlike state elimination, the complexity profile is uniform: dense and sparse graphs incur the same asymptotic cost. Individual expression sizes are bounded by the star of the largest diagonal entry, which can grow but is predictable in structure.

Algorithm 2 Matrix Kleene Closure for RSM

Require: RSM $M = (Q, q_0, F, \delta)$ with $m = |Q|$
Ensure: KAT expressions $D[0][f]$ for each $q_f \in F$ such that $L(D[0][f]) = L_f(E(M))$

for all $q_i, q_j \in Q$ **do**
  if $\exists (q_i, g, a, q_j) \in \delta$ **then**
    $D[i][j] \leftarrow \sum_{\{(q_i,g,a,q_j)\in\delta\}} (g \cdot \pi_{ij})$
  else
    $D[i][j] \leftarrow 0$
  end if
end for
for $k \leftarrow 0$ **to** $m-1$ **do**
  $\mathrm{loop}_k \leftarrow D[k][k]^*$
  for $i \leftarrow 0$ **to** $m-1$ **do**
    if $i \neq k$ **then**
      for $j \leftarrow 0$ **to** $m-1$ **do**
        if $j \neq k$ **then**
          $\mathrm{rest} \leftarrow (D[i][k] \cdot \mathrm{loop}_k \cdot D[k][j])$
          $D[i][j] \leftarrow D[i][j] + \mathrm{rest}$
        end if
      end for
    end if
  end for
end for
return $\sum_{q_f \in F} D[\mathrm{idx}(q_0)][\mathrm{idx}(q_f)]$

V. Conclusion

We presented a formal compilation framework for reactive state machines using Kleene Algebra with Tests, in which compilation is a KAT reduction algorithm and correctness is certified by the KAT equational theory. We proposed two algorithmic strategies: Divide and Conquer via state elimination, and Dynamic Programming via matrix Kleene closure. Each of the algorithm produce a semantically equivalent implementation from a shared KAT-encoded specification, with structurally distinct normal forms that differ in code shape and performance characteristics. We showed that both strategies are instances of a common algebraic elimination scheme on the KAT transition matrix, distinguished by the ordering of elimination steps.

The two-layer mutable state scheme (KAT+B! for Boolean flags, predicate abstraction for thresholds) handles control-flow correctness but not data-level liveness or safety properties. Atom count scales as $m \cdot 2^{|\Pi_T^{\mathrm{ext}}|}$, which can become impractical for machines with many external predicates.

Future work expanding this topic may about extending the framework to quantitative extensions of KA (for reasoning over continuous dynamics), supporting synthesis (generating a machine from a KAT specification rather than compiling from one), and integrating the equivalence checker with a hardware-targeted backend for certified bare-metal deployment remain open directions.

APPENDIX

For further study, the author create an experiment setup and a YouTube video explaining the topics. Feel free to check out the link below:

- GitHub repository: https://github.com/m-akma1/if2211-paper
- YouTube video: https://www.youtube.com/@IF24099/videos

ACKNOWLEDGMENT

The author would like to express gratitude to Allah SWT for His blessings and guidance in completing this paper. Appreciation is also extended to Prof. Dr. Ir. Rinaldi, M.T and the Lecturer Team of IF2211 Algorithm Strategy for their dedication and the knowledge shared throughout the course and to the Lecturer Team of IF2224 Formal Language and Automata for the insight about Theory of Computation that opened a new world of Theoretical Computer Science the author that inspired this paper. Special thanks also given to the VTOL team at Aksantara ITB, especially for the RSC Department, for providing the author with high quality code of the precision landing state machine. The author hopes this paper may serve as a foundation for further development beyond its initial submission.

REFERENCES

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[2] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pp. 427–443, 1997.

[3] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Transactions on Electronic Computers*, vol. EC-9, no. 1, pp. 39–47, 1960.

[4] S. Warshall, "A theorem on Boolean matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, 1962.

[5] D. Kozen, "A completeness theorem for Kleene algebras and the algebra of regular events," *Information and Computation*, vol. 110, no. 2, pp. 366–390, 1994.

[6] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. ACM, 2014, pp. 113–126.

[7] D. Kozen and M.-C. Patron, "Certification of compiler optimizations using Kleene algebra with tests," in *Computational Logic (CL 2000)*, ser. Lecture Notes in Computer Science, vol. 1861. Springer, 2000, pp. 568–582.

[8] G. Berry and G. Gonthier, "The esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[10] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouderoux, and M. Rittri, "A formally verified compiler for Lustre," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017, pp. 586–601.

[11] A. Angus and D. Kozen, "Kleene algebra with tests and program schematology," Cornell University, Tech. Rep. TR2001-1844, 2001.

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 19 June 2026

Muhammad Akmal
13524099