Atomic Normalization of SLP-Definable Deterministic Transitions

Abstract

We study deterministic state transitions that admit representation as finite acyclic straight-line computations over a fixed basis of total functions. Structural characteristics of such representations—such as dependency depth, operation count, and potential parallelism—depend on the granularity at which operations are treated as atomic. Without a uniform atomic level, these quantities are representation-dependent and lack structural invariance.

This work introduces a constructive atomic normalization framework for the class of SLP-definable deterministic transitions. Given a transition specified by a finite acyclic dependency DAG, the procedure replaces each composite node by a bounded atomic subgraph over a fixed atomic basis and produces a linear routine whose induced dependency graph refines the original structure. Normalization induces an injective embedding of the original DAG into the atomic DAG, preserves partial order relations, preserves dependency depth up to constant factors determined by the expansion templates, and preserves the computed transition function. The procedure is finite, effective, and deterministic under fixed conventions.

The contribution is structural rather than computational: the framework does not introduce a new computational model, does not claim any computational advantages, and does not apply to arbitrary deterministic state-transition systems. The results establish basis-relative structural invariants for a well-defined subclass of transitions that already admit finite static dependency graphs.

Author Name Sergei Zubov
zubov369@gmail.com

1. Introduction

1.1 Background

Dependency graphs are a fundamental structural representation of computations.

Straight-line programs (SLP), static single assignment (SSA) form, and circuit netlists describe computations as directed acyclic graphs (DAGs) in which nodes represent operations and edges represent data dependencies. Such representations are widely used in compiler theory, circuit design, and formal models of computation.

At the level of deterministic state-transition systems (STS), a transition is a total function $\Delta: X \to X$ on a configuration space X. When such a transition is expressed as a straight-line computation over a fixed basis of operations, it naturally induces a finite acyclic dependency graph.

However, a fundamental structural ambiguity arises from the granularity at which operations are treated as atomic. In standard DAG-based representations, nodes may correspond either to primitive operations or to composite operations whose internal structure is not explicitly

represented. For example, arithmetic addition, memory lookup, or word-level logical instructions may appear as single nodes, even though they correspond to structured subcomputations at a lower level of granularity.

As a consequence, structural characteristics of a transition—such as dependency depth, critical path length, total operation count, or potential parallelism—depend on the chosen representation and on the level at which operations are considered atomic. Without fixing a uniform atomic basis, these structural quantities are representation-dependent and therefore cannot serve as invariant descriptors of transitions at the semantic level.

This observation motivates the need for a normalization framework that makes the atomic level explicit and uniform.

1.2 Problem Statement and Motivation

Given a deterministic transition represented as a finite DAG over a fixed basis, the following questions naturally arise:

1. Can composite nodes be systematically expanded into atomic substructures in a uniform and constructive manner?
2. Can such expansion preserve the dependency structure of the original computation?
3. Can structural parameters such as depth and partial order be related in a controlled way to those of the original DAG?
4. Can the normalization procedure be made deterministic under fixed conventions?

Existing DAG-based representations provide syntactic structure but do not impose a canonical atomic level. Composite operations may hide their internal dependency structure, and different representations of the same transition may expose different structural characteristics.

Atomic normalization addresses this granularity ambiguity by fixing a uniform atomic basis and systematically expanding each composite node into a prescribed atomic substructure. This process makes the dependency structure explicit at a common level of granularity and allows structural characteristics of transitions to be defined relative to that fixed atomic level.

The aim is not to introduce a new representation language for computations, nor to redefine circuit models or compiler normal forms. Rather, the goal is to formalize an atomic normalization at the level of deterministic transition semantics, establishing structural preservation results that hold for a well-defined subclass of transitions.

The framework does not extend to arbitrary deterministic state-transition systems and is not intended to modify or reinterpret classical models of computation.

1.3 Contributions

This paper makes the following contributions.

1. **Formalization of the SLP-Definable Subclass.**
   We identify a subclass of deterministic state transitions that admit representation as finite acyclic straight-line computations over a fixed finite basis of total functions. This subclass is

characterized by structural staticity: finiteness, acyclicity, and absence of data-dependent structural modification within a single transition.

2. **Constructive Atomic Normalization.**
   This work introduces a Canonical Routine Transition System (CRTS) model and provides a constructive procedure that transforms any SLP-definable transition DAG into an atomic linear routine while preserving the underlying dependency structure.
3. **Structure Preservation Theorem.**
   Normalization induces an injective embedding of the original DAG into the atomic dependency DAG. Partial order relations are preserved, and the critical path depth is preserved up to constant factors determined solely by the expansion rules.
4. **Semantic Equivalence.**
   We establish that the normalized CRTS routine implements a transition semantically equivalent to the original SLP-definable transition, up to a fixed encoding convention.
5. **Determinism Under Fixed Conventions.**
   We show that, once traversal order, expansion templates, and naming conventions are fixed, the normalization procedure is deterministic and yields a unique atomic routine up to renaming.
6. **Applicability to Fixed Machine Transitions.**
   We observe that fixed transition rules of RAM or Turing machines, under finite encodings and fixed word sizes, fall within the SLP-definable subclass and therefore admit atomic normalization in the sense defined here.

No claims are made regarding minimality of the resulting atomic form, optimality with respect to time or hardware cost, or universality beyond the SLP-definable subclass. The contribution is structural: it establishes a uniform atomic level for a well-defined class of deterministic transitions and proves preservation properties relative to that level.

The results concern structural normalization only and do not introduce or propose any new machine model or computational paradigm.

2. Deterministic State-Transition Systems (STS)

2.1 Definition

We begin by fixing the general semantic framework within which the present work is formulated.

A **deterministic state-transition system (STS)** is a pair

$$S = (X, \Delta),$$

where:

- X is a non-empty configuration space,
- $\Delta: X \to X$ is a total deterministic transition function.

The configuration space X may represent, for example:

- memory states of a machine,
- register configurations,

- tape configurations of a Turing machine,
- structured tuples encoding control state and data components.

No algebraic or topological structure on X is assumed beyond what is required to define Δ as a total function.
Totality means that for every $x \in X$ the value $\Delta(x)$ is defined.
Determinism means that Δ assigns a unique successor state to each configuration.
The semantics of the system is given by iteration of Δ.
A computation is the sequence:

$$x, \Delta(\mathrm{x}), \Delta^2(\mathrm{x}), \ldots$$

for an initial configuration $x \in X$.
This formulation abstracts away from implementation details and focuses purely on the transition function at the semantic level.

2.2 Structural Generality of STS

The STS definition intentionally imposes **no structural constraints** on the internal form of Δ.
The transition function may be defined in arbitrarily complex ways.
In particular, the following forms of internal behavior are fully permitted within the STS framework:

- **Dynamic addressing.**
  Memory locations or structural components of the state may be accessed through indices computed at runtime.
- **Data-dependent branching.**
  The internal structure of the computation performed by Δ may depend on the input state.
- **Interpretation.**
  The transition may execute instructions encoded within the state itself, effectively interpreting programs stored in memory.
- **Variable-length internal execution.**
  The amount of internal work required to compute $\Delta(x)$ may vary with x, even though the external transition is a single semantic step.
- **Self-modifying structure.**
  The transition may alter components that influence future control flow in ways that are not structurally fixed in advance.

From the standpoint of the STS definition, all of these behaviors are admissible, since Δ is only required to be a total function on X.
This structural generality implies that the class of deterministic STS is extremely broad. It includes, for example:

- deterministic RAM models,
- deterministic Turing machines,
- register machines,
- stack machines,

- interpreters of encoded programs,
- and arbitrary deterministic algorithms represented extensionally as total functions.

Crucially, the STS abstraction does not constrain how the transition is internally realized.
It does not require:

- finiteness of a dependency graph,
- acyclicity,
- absence of branching,
- fixed structural templates.

The present work does **not** attempt to normalize arbitrary deterministic STS.
Instead, we will identify a structurally restricted subclass of STS for which atomic normalization is well-defined and provable.
The distinction between the general STS framework and the structurally static subclass introduced later is essential.
The normalization results established in subsequent sections apply only to transitions whose internal structure satisfies specific static constraints.

Conceptual Role of This Section

This section serves two purposes:

1. It clearly separates semantic determinism (total function Δ) from structural properties of its internal realization.
2. It emphasizes that the normalization framework developed later is not intended to apply to arbitrary STS, but only to a well-defined structurally restricted subclass.

By doing so, we avoid conflating general computability with structural normalizability.
The present framework therefore does not attempt to characterize or normalize arbitrary STS, and no claims are made regarding computability or expressive power.

3. The SLP-Definable Transition Subclass

3.1 Fixed Computational Basis

To introduce a structurally normalizable subclass of deterministic transitions, we first fix a computational basis.
Let D be a finite or finitely encoded domain of values.
Let

$$G = \{g_1, g_2, \dots, g_m\}$$

be a fixed finite set of total functions

$$g_i : D^{k_i} \to D,$$

where each arity k_i is fixed and finite.
The basis G is assumed to be:

- finite,
- fixed throughout the normalization procedure,
- composed of total deterministic functions.

The choice of G determines the level of granularity at which operations are considered primitive. All structural statements in this paper are relative to this fixed basis.

3.2 Straight-Line (SLP) Representation of Transitions

Let $S = (X, \Delta)$, be a deterministic STS.

We say that Δ is **SLP-definable over basis** G if there exists a finite acyclic directed graph

$$H = (V, E)$$

with the following properties:

1. **Finite node set.**
   V is finite.
2. **Acyclicity.**
   H contains no directed cycles.
3. **Node labeling.**
   Each node $v \in V$ is labeled by some function $g_v \in G$.
4. **Input arity consistency.**
   For each node v, the number of incoming edges equals the arity of g_v.
5. **Designated input nodes.**
   Certain nodes correspond to components of the input state $x \in X$.
6. **Designated output nodes.**
   Certain nodes define components of $\Delta(x)$.

The evaluation of H on input $x \in X$ proceeds in topological order.

The resulting output values define $\Delta(x)$.

Thus, Δ is represented as a finite straight-line computation (SLP) over basis G.

3.3 Structural Staticity

An SLP-definable transition satisfies the following structural properties:

(1) Finiteness

The dependency graph H is finite and independent of the runtime input.

(2) Acyclicity

No cyclic data dependencies are present within a single transition.

(3) Absence of Data-Dependent Structural Modification

The structure of H does not depend on the input x.

The same graph is used for all inputs.

(4) No Dynamic Indexing of Argument Positions

Edges in H connect statically determined argument positions.

The graph does not contain nodes whose input positions are selected dynamically at runtime.

(5) No Interpretation of Encoded Programs

The transition does not interpret an encoded description of another program stored in X.

All operations are statically fixed by H.

These properties define what we call **structural staticity** of a transition.

3.4 Functional Indirect Addressing

The restriction against dynamic indexing requires clarification.
We distinguish between:

- **Structural dynamic indexing:**
  Runtime-dependent selection of which subgraph to execute or which argument position to bind.
- **Value-level lookup:**
  A function lookup: $D \times D^n \to D$ treated as a total function in G.

Functional indirect addressing (e.g., memory lookup in fixed-size memory encoded as a vector) is permitted provided it is represented as a fixed function in the basis and does not alter the structure of the dependency graph.
Formally:
If lookup $\in G$, then its internal behavior is encapsulated within a single node (or later expanded atomically).
The dependency graph itself remains independent of runtime data.
Thus, SLP-definability allows value-level indirection but excludes structural variability.

3.5 Applicability to Fixed Machine Transitions

Consider a RAM or Turing machine with:

- fixed word size,
- fixed finite control,
- fixed finite encoding of configurations.

Each individual transition rule defines a total function over a finite domain.
Such a function admits a finite straight-line representation over an appropriate basis (e.g., Boolean or arithmetic primitives).
Therefore, fixed machine transitions under finite encodings fall within the SLP-definable subclass.
This statement does not claim that unrestricted RAM or Turing machine models, viewed as infinite families parameterized by word size or memory size, are globally SLP-definable.
The claim applies only to fixed transitions under fixed encodings.

3.6 Boundary of the SLP Subclass

The SLP-definable subclass excludes transitions exhibiting:

- Data-dependent branching that changes the executed structure within a single semantic step.
- Variable numbers of executed operations depending on the input.
- Interpretation of arbitrary encoded programs.
- Self-modifying structural behavior affecting the current transition.

Formally, if the internal computation of $\Delta(x)$ cannot be represented by a single finite acyclic graph independent of x, then Δ does not belong to the SLP-definable subclass.

3.7 Maximality Within Structural Staticity

The SLP-definable subclass captures the broadest class of transitions satisfying structural staticity:

- fixed finite dependency structure,
- no runtime structural modification,
- fixed computational basis.

Any attempt to enlarge the class to include data-dependent structural variation or interpretation would require either:

- extending the computational basis to encode interpreters,
- or abandoning acyclicity or finiteness.

Thus, SLP-definability represents the natural boundary for normalization results that rely on finite static dependency graphs.

No claim of formal maximality is made beyond this structural observation.

3.8 Conceptual Role of the Subclass

The SLP-definable subclass serves as the structural layer at which:

- transitions admit finite static dependency graphs,
- composite operations can be systematically expanded,
- structural properties are meaningfully defined,
- atomic normalization can be proven.

The remainder of this paper operates strictly within this subclass.

This subclass forms the natural structural domain to which the normalization procedure applies; extending beyond it would require abandoning static structure, which lies outside the scope of this work.

4. Atomic Normalization Framework

The purpose of this section is to introduce the atomic level at which normalization is defined. We first fix an atomic basis, then define the expansion of composite nodes, and finally formalize the Canonical Routine Transition System (CRTS) model that realizes the normalized transition.

4.1 Atomic Basis Selection

Let D be the underlying value domain fixed in Section 3.

We now fix a finite atomic basis

$$A = \{a_1, a_2, \dots, a_r\},$$

where each

$$a_i : D^{l_i} \to D$$

is a total deterministic function of fixed finite arity l_i.

The atomic basis A satisfies:

1. **Finiteness.**
   The set A is finite.
2. **Fixed arity.**
   Each atomic operation has fixed finite input arity.
3. **Totality.**
   Each a_i, is defined for all inputs in D^{l_i}.
4. **Structural primitiveness.**
   Atomic operations are treated as indivisible at the level of normalization.

The choice of A determines the granularity of atomic normalization.
All structural properties established later are relative to this fixed atomic basis.
No assumption is made about minimality of ; it is simply fixed for the normalization procedure.

4.2 Atomic Expansion Rule

Let

$$H = (V, E)$$

be an SLP-definable transition DAG over basis G.
Each node $v \in V$, labeled by some function $g_v \in G$, is replaced by a finite atomic DAG

$$H_v = (V_v, E_v)$$

over the atomic basis A.
The atomic expansion must satisfy:

(1) Local Correctness

For every node v of H the atomic subgraph H_v computes the same output value using operations from A.

(2) Bounded Size

There exists a constant k, depending only on the chosen expansion templates and basis A, such that

$$|V_v| \leq k$$

for all $v \in V$.
Thus, expansion increases size at most linearly in the number of original nodes.

(3) Interface Preservation

The inputs of H_v correspond exactly to the inputs of node v in H, and its output corresponds to the output of v.

(4) Structural Locality

Dependencies internal to H_v are confined to its own substructure.
Edges between distinct nodes $u, v \in V$ are replaced by edges between corresponding atomic substructures.

Global Atomic DAG Construction

The normalized atomic DAG

$$H' = (V', E')$$

is obtained by:

1. Replacing each node $v \in V$ by its atomic subgraph H_v,
2. Rewiring edges to connect output nodes of H_u to input nodes of H_v whenever $u, v \in E$,
3. Designating final output nodes corresponding to the original transition output.

The resulting graph H' is finite and acyclic.

4.3 The Canonical Routine Transition System (CRTS)

The atomic DAG H' defines a structurally expanded transition.
We now define the execution model used to realize this atomic structure.

4.3.1 CRTS Structure

A Canonical Routine Transition System (CRTS) consists of:

- A finite atomic basis A,
- A finite routine R, which is a linear sequence of atomic operations,
- Deterministic execution semantics.

Formally, a CRTS routine is a sequence

$$R = \{r_1, r_2, \dots, r_m\},$$

where each r_i is an application of some atomic function $a \in A$ to previously computed values or input components.

4.3.2 Linear Routine vs Dependency Graph

Although the routine R is linear, it induces a dependency DAG H':

- Each atomic operation corresponds to a node in H',
- Edges represent data dependencies between atomic operations.

The sequential order of R is a linear extension of the partial order defined by H'.
All structural properties discussed in this paper — including:

- depth,
- critical path,
- parallelism,
- dependency preservation —

refer to the induced atomic dependency DAG H', not to the sequential execution order of R.

4.3.3 Deterministic Execution

Given an input state $x \in X$, execution of CRTS proceeds as follows:

1. Initialize input values.
2. Execute atomic operations in the order specified by R.
3. Produce output values corresponding to the designated output nodes.

Because atomic operations are total and deterministic, the CRTS transition

$$\Delta_{CRTS,R}: X \to X$$

is total and deterministic.

4.3.4 Relationship to STS

CRTS itself defines a deterministic STS:

$$S_{CRTS} = (X, \Delta_{CRTS,R}).$$

CRTS is not introduced as a universal machine model.
It serves as a normalization schema for SLP-definable transitions.

4.4 Conceptual Role of Atomic Normalization

Atomic normalization performs two structurally distinct operations:

1. **Granularity Fixing.**
   It fixes a uniform atomic level relative to basis A.
2. **Structure Exposure.**
   It makes the internal dependency structure of composite operations explicit.

This process does not alter semantic behavior but refines the structural representation of the transition.
The following sections establish formal guarantees regarding preservation of dependencies, depth, and semantics under this transformation.
CRTS is not introduced as a computational model but solely as a structural normalization schema for SLP-definable transitions.

5. Construction of Normalization

In this section we formally define the normalization procedure

$$Norm(H),$$

which transforms an SLP-definable transition DAG H into an atomic CRTS routine R.
We establish:

- finiteness and effectiveness of the construction,
- preservation of dependency structure,
- linear size bound,
- depth preservation up to constants,
- semantic equivalence.

5.1 Expansion Algorithm

Let

$$H = (V, E)$$

be a finite acyclic DAG over basis G, representing an SLP-definable transition.
We assume a fixed atomic basis A and fixed expansion templates for each $g \in G$, as defined in Section 4.

Algorithm: Norm(H)

Input:
Finite acyclic DAG $H = (V, E)$.
Output:
Atomic CRTS routine R.

Step 1: Fix Traversal Order

Compute a fixed topological ordering of V:

$$v_1, v_2, \ldots, v_n,$$

according to a predetermined deterministic rule (e.g., lexicographic on node identifiers).
Because H is finite and acyclic, such an ordering exists.

Step 2: Expand Nodes

For each node $v_i \in V$:

1. Retrieve its label $g_{v_i} \in G$.
2. Replace v_i by the fixed atomic subgraph H_{v_i} over basis A.
3. Assign fresh identifiers to all atomic nodes inside H_{v_i}.

Each H_{v_i} has size bounded by a constant k independent of H.

Step 3: Rewire Dependencies

For each edge $(u, v) \in E$:

- Let o_u denote the designated output node of H_u.
- Let i_v denote the appropriate input node of H_v.
- Add an edge (o_u, i_v) in the atomic graph.

This preserves the original dependency relation.

Step 4: Produce Atomic DAG

Let

$$H' = (V', E')$$

be the resulting atomic DAG constructed as the union of all H_v with rewired edges.

Step 5: Linearize into CRTS Routine

Compute a topological ordering of H' using the fixed deterministic rule.

$$R = (r_1, r_2, \ldots, r_N),$$

where $N = |V'|$, and each r_j corresponds to one atomic node in topological order.

Termination and Effectiveness

Since:

- H is finite,
- each expansion H_v is finite,
- rewiring is finite,

the algorithm terminates after finitely many steps.
Thus, normalization is constructive and effective.

5.2 Injective Embedding and Dependency Preservation

We now formalize structural preservation.

Definition (Embedding Mapping)

Let $H = (V, E)$ and $H' = (V', E')$.
Define

$$\varphi: V \to V'$$

by mapping each original node $v \in V$ to the designated output node of its atomic subgraph H_v.

Lemma 5.1 (Injectivity)

The mapping φ is injective.

Proof.

Each node $v \in V$ is expanded into a distinct atomic subgraph H_v with fresh identifiers.
Their designated output nodes are distinct.
Therefore, $\varphi(v_1) = \varphi(v_2)$ implies $v_1 = v_2$. ■

Lemma 5.2 (Partial Order Preservation)

If

$$u \prec_{\mathrm{H}} v,$$

then

$$\varphi(u) \prec_{\mathrm{H}'} \varphi(v).$$

Proof.

If $u \prec_{\mathrm{H}} v$, then there exists a directed path from u to v in H.
Under expansion:

- each edge is replaced by edges between output nodes of H_u and input nodes of H_v,
- internal paths inside H_v preserve local dependencies.

Thus, there exists a path from $\varphi(u)$ to $\varphi(v)$ in H'. ■

5.3 Size Bound

Let $n = |V|$.
Since each node $v \in V$ is replaced by a subgraph H_v with at most k nodes,

$$|V'| \leq kn.$$

Thus, normalization increases size at most linearly.

5.4 Depth Preservation

Let $depth(H)$ denote the maximum length of a dependency chain in H.

Theorem 5.3 (Depth Bound)

There exist constants c , c_0 depending only on expansion templates, such that

$$depth(H') \leq c \cdot depth(H) + c_0.$$

Proof Sketch.

Consider a maximal chain

$$v_{i_1} \prec_{\mathrm{H}} v_{i_2} \prec_{\mathrm{H}} \dots \prec_{\mathrm{H}} v_{i_d}.$$

Each v_{i_j} is replaced by a subgraph $H_{v_{i_j}}$ of bounded depth d_0.
Thus, the corresponding atomic chain has length at most

$$d_0 \cdot d + c_0,$$

where constants depend only on local expansion depth. ■

Remark

Depth refers strictly to dependency depth in H', not to the sequential execution length of routine R.

5.5 Semantic Preservation

Theorem 5.4 (Semantic Equivalence)

For all input states $x \in X$,

$$\Delta_{CRTS,R}(x) = \Delta(\mathrm{x}),$$

up to fixed encoding conventions.

Proof Sketch

We proceed by induction over the topological order of H.

- Each atomic subgraph H_v correctly computes the function g_v.
- Dependencies ensure that inputs to each H_v are identical to those in H.
- Final outputs correspond exactly to outputs of H.

Thus, the overall computed function matches Δ. ■

5.6 Determinism of Normalization

Given fixed:

- topological traversal rule,
- expansion templates,
- naming scheme,

the algorithm $Norm(H)$ produces a uniquely determined atomic routine.

Theorem 5.5 (Determinism)

Under fixed conventions, normalization is deterministic:

$$Norm(H)$$

is uniquely defined up to renaming of intermediate variables.

Proof.

All steps of the algorithm are deterministic under fixed conventions.

No nondeterministic choice remains. ■

Summary of Section 5

The normalization procedure:

- is finite and effective,
- preserves partial order,
- preserves depth up to constants,
- increases size linearly,
- preserves semantics,
- is deterministic under fixed conventions.

This completes the formal construction.

6. Structural Properties of Atomic Normalization

In this section we formalize the structural guarantees induced by atomic normalization.

Let:

- $H = (V, E)$ be the original SLP-definable dependency DAG,
- $H' = (V', E')$ be the atomic DAG constructed by normalization.

All statements refer to dependency graphs, not to the sequential execution order of the CRTS routine.

6.1 Injective Embedding

We first establish that the original DAG embeds into the atomic DAG in a structure-preserving way.

Definition 6.1 (Embedding Mapping)

For each node $v \in V$, let H_v be its atomic expansion subgraph.

Let $o_v \in V'$ denote the designated output node of H_v.

Define:

$$\varphi: V \to V'$$

by

$$\varphi(v) = o_v.$$

Theorem 6.1 (Injective Embedding)

The mapping φ is injective.

Proof

Each node $v \in V$ is replaced by a distinct atomic subgraph H_v with fresh identifiers.

The designated output nodes o_v are therefore distinct for distinct v.

Hence,

$$\varphi(v_1) = \varphi(v_2) \Rightarrow v_1 = v_2.$$

Thus, φ is injective. ■

Interpretation

The embedding ensures that every original node has a uniquely identifiable counterpart in the atomic graph.

6.2 Partial Order Preservation

We now show that the dependency relation of the original DAG is preserved.

Recall that:

$$u \prec_{\mathrm{H}} v$$

means there exists a directed path from u to v in H.

Theorem 6.2 (Dependency Preservation)

If

$$u \prec_{\mathrm{H}} v,$$

then

$$\varphi(u) \prec_{\mathrm{H}'} \varphi(v).$$

Proof

Assume $u \prec_{\mathrm{H}} v$.

Then there exists a path:

$$u = w_0 \to w_1 \to \cdots \to w_m = v$$

in H.
Under normalization:

- each w_i is replaced by H_{w_i},
- edges (w_i, w_{i+1}) are replaced by edges connecting o_{w_i} to the appropriate input node of $H_{w_{i+1}}$,
- internal paths inside each H_{w_i} preserve local dependencies.

Thus, there exists a path:

$$\varphi(u) \to \cdots \to \varphi(v)$$

in H'.
Hence,

$$\varphi(u) \prec_{H'} \varphi(v). \blacksquare$$

Corollary 6.3
The partial order of H embeds into that of H'.

6.3 Size Bound

Let

$$n = |V|.$$

Each node $v \in V$ is replaced by a subgraph H_v satisfying:

$$|V(H_v)| \leq k,$$

for some constant k independent of H.
Theorem 6.4 (Linear Size Bound)

$$|V'| \leq kn.$$

Proof

$$|V'| = \sum_{v \in V} |V(H_v)| \leq \sum_{v \in V} k = kn. \blacksquare$$

Remark
Normalization increases graph size at most linearly.

6.4 Depth Preservation

Let:

$$D(H) = depth(H),$$
$$D(H') = depth(H').$$

Depth refers to the maximum length of a directed dependency chain.
Lemma 6.5 (Local Depth Bound)
Let

$$d_0 = \max_{v \in V} depth(H_v).$$

Then d_0 is a constant depending only on expansion templates.

Theorem 6.6 (Depth Preservation)

There exist constants c, c_0 such that

$$D(H') \leq c \cdot D(H) + c_0.$$

Proof

Consider a maximal chain in H:

$$v_1 \prec_H v_2 \prec_H \dots \prec_H v_d ,$$

where $d = D(H)$.

Each v_i expands into H_{v_i} of depth at most d_0.

A maximal chain in H' corresponding to this chain consists of:

- at most d0 internal steps per H_{v_i},
- one transition edge between consecutive expansions.

Thus,

$$D(H') \leq d_0 \cdot d + c_0.$$

Setting $c = d_0$ gives the result. ■

Important Clarification

Depth refers to dependency depth of the atomic DAG H',
not to the sequential execution length of the CRTS routine.

6.5 Structural Parallelism

We now analyze structural parallelism.

Define:

$$W(H) = |V| \text{ (Work)},$$
$$D(H) = depth(H) \text{ (Depth)},$$
$$P(H) = \frac{W(H)}{D(H)} \text{ (Structural Parallelism)}.$$

These quantities describe the theoretical parallelism available under an idealized unlimited-processor model.

Theorem 6.7 (Parallelism Preservation up to Constants)

$$P(H') = \Theta(P(H)).$$

Proof

From Section 6.3:

$$W(H') \leq kW(H).$$

From Section 6.4:

$$D(H') \leq cD(H) + c_0.$$

Since each original dependency chain in H maps to a dependency chain in H',
we obtain:

$$W(H') \geq W(H),$$
$$D(H') \geq D(H).$$

Thus,

$$\frac{W(H')}{D(H')} = \Theta\left(\frac{W(H)}{D(H)}\right). \blacksquare$$

Interpretation

Atomic normalization:

- preserves asymptotic structural parallelism,
- does not asymptotically collapse or inflate dependency structure.

This statement is purely structural and does not imply hardware execution speed.

Summary of Section 6

Atomic normalization satisfies:

1. Injective embedding of original nodes.
2. Preservation of partial order.
3. Linear size expansion.
4. Depth preservation up to constants.
5. Preservation of structural parallelism up to constants.

These results formalize that atomic normalization is a structure-preserving refinement of the original SLP transition.

7. Semantic Equivalence

In this section we prove that atomic normalization preserves the semantic behavior of the original transition.

Let:

- H be the original SLP-definable dependency DAG representing transition Δ,
- H' be the atomic DAG constructed by normalization,
- R be the CRTS routine obtained by linearization of H',
- $\Delta_{\mathrm{CRTS},\mathrm{R}}$ be the transition implemented by executing R.

We assume a fixed encoding scheme identifying components of the state X with input and output nodes of the DAG.

7.1 Semantic Preservation Theorem

Theorem 7.1 (Semantic Preservation)

For all input states $x \in X$,

$$\Delta_{\mathrm{CRTS},\mathrm{R}}(\mathrm{x}) = \Delta(\mathrm{x}),$$

up to the fixed encoding convention used to map DAG outputs to state components.

Clarification

Equality "up to encoding" means:

- The output values computed by H' correspond exactly to the output values of H,
- The mapping between output nodes and components of X is fixed and identical in both representations.

No semantic transformation of the state space occurs during normalization.

7.2 Proof

We prove the theorem constructively.

Step 1: Semantics of the Original DAG

Let $H = (V, E)$ be evaluated on input $x \in X$.

Since H is finite and acyclic, there exists a topological order:

$$v_1, v_2, \dots, v_n.$$

Each node v_i computes:

$$y_{v_i} = g_{v_i}(y_{u_1}, \dots, y_{u_k}),$$

where $u_1, \dots, u_k$. are its predecessors in H.

The output of H is defined by designated output nodes.

Step 2: Local Correctness of Expansion

For each node $v \in V$, we constructed an atomic subgraph H_v satisfying:

- It computes exactly the same function g_v,
- It takes as input the same predecessor values,
- It produces the same output value.

Formally, if

$$y_v = g_v(y_{u_1}, \dots, y_{u_k}),$$

then evaluation of H_v yields the same value at its designated output node.

This follows directly from the construction of atomic expansion templates.

Step 3: Induction over Topological Order

We prove by induction over the topological order of H that for each node $v \in V$:

$$value\ computed\ at\ \varphi(v)\ in\ H'\ equals\ y_v.$$

Base Case

For input nodes corresponding directly to components of x,

values in H and H' are identical by construction.

Inductive Step

Assume the statement holds for all predecessors of v.

By construction:

- The inputs to H_v in H' are exactly the outputs of atomic subgraphs corresponding to predecessors of v.
- By induction hypothesis, those values equal the values computed in H.

By local correctness of expansion:

- H_v computes exactly g_v on those inputs.

Therefore:

$$value\ at\ \varphi(v) = y_v.$$

Step 4: Output Consistency

Since designated output nodes of H correspond to designated output nodes in $'$, and their values coincide, the final outputs of both graphs are identical.

Thus,

$$\Delta_{\mathrm{CRTS},R}(\mathrm{x}) = \Delta(\mathrm{x}).\ \blacksquare$$

7.3 Corollaries

Corollary 7.2 (Functional Equivalence)

The normalized CRTS routine defines the same total function on X as the original SLP-definable transition.

Corollary 7.3 (STS Preservation)

If

$$S = (X, \Delta)$$

is an SLP-definable STS, and

$$S' = (X, \Delta_{CRTS,R}),$$

then

$$\text{S and } S'$$

are semantically identical deterministic STS.

7.4 Important Distinctions

This theorem establishes:

- semantic equivalence,
- preservation of transition function.

It does **not** claim:

- minimality of the atomic form,
- optimality of operation count,
- equivalence of sequential execution length,
- preservation of implementation cost.

The result concerns semantic equality and structural refinement only.

Summary of Section 7

Atomic normalization:

- preserves computed values at every node,
- preserves final state transformation,
- yields a transition function identical to the original,
- maintains totality and determinism.

Thus, normalization is a semantics-preserving structural refinement.

8. Determinism of Normalization

In previous sections we established existence, structural preservation, and semantic equivalence of the atomic normalization procedure.

We now address an additional structural property: determinism of the normalization process itself.

Since normalization involves:

- traversal of the original DAG,
- expansion of nodes,
- generation of fresh identifiers,

we must specify conventions that eliminate nondeterministic choices.

8.1 Fixed Conventions

To ensure determinism, we fix the following conventions globally.

8.1.1 Traversal Order

Let

$$H = (V, E)$$

be a finite acyclic DAG.

We fix a deterministic topological ordering rule.

For example:

- nodes are assigned canonical identifiers,
- ties are broken lexicographically,
- topological sorting uses a deterministic priority policy.

Thus, for each DAG H, there exists a uniquely determined topological sequence:

$$v_1, v_2, \dots, v_n.$$

No arbitrary choice remains in traversal.

8.1.2 Expansion Templates

For each function symbol

$$g \in G,$$

we fix a single expansion template:

$$T_g,$$

which defines the atomic subgraph H_v for any node labeled by g.

These templates specify:

- internal atomic nodes,
- internal dependency edges,
- designated input interface,
- designated output node.

No alternative expansions are permitted once templates are fixed.

8.1.3 Naming Rules

We fix a deterministic naming scheme for intermediate atomic values.

For example:

- atomic nodes are numbered sequentially in traversal order,
- identifiers incorporate the index of the original node,
- no reuse of identifiers is allowed.

This ensures that the generated atomic DAG and the resulting routine are syntactically determined.

8.1.4 Summary of Fixed Conventions

The following elements are fixed:

1. Topological traversal rule,
2. Expansion template for each basis function,
3. Identifier generation policy.

Once these are fixed, the normalization procedure contains no nondeterministic choices.

8.2 Determinism Theorem

We now formalize the determinism property.

Definition 8.1 (Normalization Procedure)

Let

$$Norm(H)$$

denote the CRTS routine obtained by applying the expansion algorithm of Section 5 under the fixed conventions of Section 8.1.

Theorem 8.2 (Determinism of Normalization)

Under fixed conventions, for any SLP-definable DAG H,

$$Norm(H)$$

is uniquely defined up to renaming of intermediate identifiers.

Proof

The normalization algorithm consists of the following deterministic steps:

1. Compute the topological ordering of H using the fixed rule.
   Since the rule is deterministic, the ordering is uniquely determined.
2. For each node v in this order:
   - select the fixed expansion template T_g,
   - generate the corresponding atomic subgraph,
   - assign identifiers according to the fixed naming policy.
3. Rewire dependencies according to the original edge set E.
4. Produce the final routine by applying the fixed topological ordering rule to the atomic DAG.

Since:

- traversal order is uniquely determined,
- expansion templates are fixed,
- naming rules are deterministic,
- rewiring is mechanically defined by E,

the resulting atomic DAG and corresponding routine are uniquely determined.

Any remaining syntactic difference can only arise from consistent renaming of intermediate variables.

Such renaming does not alter structure or semantics.

Therefore, normalization is deterministic up to renaming. ■

8.3 Canonical Form Under Fixed Conventions

While we do not claim absolute minimality or uniqueness independent of conventions, we obtain the following structural property:

Given fixed basis A and fixed conventions, the atomic form of any SLP-definable transition is canonical relative to those conventions.

Formally:

If two DAGs H_1 and H_2 are syntactically identical, then

$$Norm(H_1) = Norm(H_2)$$

up to renaming.

This property ensures reproducibility of the normalization procedure.

8.4 Scope of Determinism

The determinism theorem:

- does not imply minimality of atomic size,
- does not imply uniqueness across different choices of atomic basis,
- does not imply uniqueness if expansion templates differ,
- does not define a canonical form independent of conventions.

Determinism is relative to fixed, globally specified normalization rules.

Summary of Section 8

Atomic normalization:

- contains no inherent nondeterminism once conventions are fixed,
- yields a reproducible atomic routine,
- provides a structurally canonical form relative to a chosen basis and expansion policy.

This completes the formal development of atomic normalization for SLP-definable transitions.

9. Induced Structural Metric

Atomic normalization not only refines the structural representation of a transition, but also induces a natural family of structural measures. These measures are defined at a fixed atomic granularity and therefore provide representation-invariant characteristics relative to the chosen atomic basis.

Let:

- H be the original SLP-definable dependency DAG,
- H' be its atomic normalization,
- A be the fixed atomic basis.

All quantities below refer to the atomic dependency DAG H'.

9.1 Atomic Length

Definition 9.1 (Atomic Length)

The **atomic length** of a transition is defined as

$$L_A(H) := |V(H')|.$$

This quantity measures the total number of atomic operations required to realize the transition under basis A.

Interpretation

Atomic length captures:

- total structural work,
- granularity-fixed operation count,
- size of the atomic refinement.

It is independent of:

- syntactic grouping of composite nodes,
- high-level instruction naming,
- alternative SLP encodings using the same basis.

Relative Invariance

Atomic length is invariant under:

- syntactic reordering of nodes preserving dependency structure,
- alternative but equivalent SLP representations that differ only in grouping of composite operations,
- any transformation that does not change the dependency DAG up to isomorphism.

Atomic length depends only on:

- the chosen atomic basis A,
- the expansion templates.

No claim of basis-independence is made.

9.2 Dependency Depth

Definition 9.2 (Atomic Depth)

The **atomic depth** of a transition is defined as

$$D_A(H) := depth(H').$$

This quantity measures the maximum length of a dependency chain in the atomic DAG.

Structural Meaning

Atomic depth reflects:

- critical path length at atomic granularity,
- structural sequential dependency,
- inherent depth of value propagation.

It does not measure:

- sequential execution time of the linear routine,
- hardware latency,
- wall-clock performance.

Stability Under Normalization

From Section 6:

$$D_A(H) \leq c \cdot depth(H) + c_0.$$

Thus, atomic depth preserves structural layering up to constants.

9.3 Structural Parallelism

We define structural parallelism as the ratio between total work and dependency depth.

Definition 9.3 (Structural Parallelism)

Let

$$W_A(H) := |V(H')|,$$
$$D_A(H) := depth(H').$$

Then

$$P_A(H) := \frac{W_A(H)}{D_A(H)}.$$

Interpretation

$P_A(H)$ approximates the maximum potential parallelism of the transition assuming unlimited processors and respecting dependencies.

It is a purely structural measure.

Preservation Property

From Theorem 6.7:

$$P(H') := \Theta\big(\mathrm{P}(\mathrm{H})\big).$$

Since $P_A(H)$ is defined as $P(H')$, it follows that

$$P_A(H) = \Theta(P(H)).$$

Thus, atomic normalization preserves asymptotic structural parallelism up to constant factors.

9.4 Representation-Level Invariance

We now clarify the sense in which these measures are invariant.

Let two SLP representations H_1 and H_2 define the same transition Δ, and suppose they differ only in grouping of composite operations but induce isomorphic dependency graphs modulo node labeling.

Under fixed atomic basis A and fixed normalization conventions:

$$L_A(H_1) = L_A(H_2),$$
$$D_A(H_1) = D_A(H_2),$$
$$P_A(H_1) = P_A(H_2).$$

Thus, atomic normalization removes granularity ambiguity introduced by composite nodes.

These quantities become invariant with respect to:

- representation granularity,
- syntactic factorization of operations,
- topological reordering.

9.5 Basis-Relativity

All structural metrics introduced here are relative to the chosen atomic basis A.

Changing the atomic basis changes:

- atomic length,
- atomic depth,
- structural parallelism.

Therefore, invariance holds only under fixed basis and fixed expansion templates.

This relativity is intentional: atomic normalization fixes a structural level rather than claiming universal canonical minimality.

9.6 Comparison Across Architectures (Conceptual Remark)

Given two SLP-definable transitions Δ_1 and Δ_2 representing alternative realizations of a computation, atomic normalization enables structural comparison at a uniform level:

- compare L_A,
- compare D_A,
- compare P_A.

This comparison is structural rather than performance-based.
It does not depend on instruction naming or high-level abstraction.

9.7 Conceptual Summary

Atomic normalization induces three fundamental structural quantities:

1. **Atomic length L_A,**
2. **Atomic depth D_A,**
3. **Structural parallelism P_A.**

These quantities:

- are well-defined at a fixed atomic granularity,
- are invariant under syntactic representation changes,
- preserve asymptotic structure of the original DAG,
- do not depend on sequential execution order.

They provide a structurally grounded measurement framework for SLP-definable transitions.

10. Scope, Limitations, and Conceptual Positioning

The results established in this paper concern a structurally restricted subclass of deterministic state-transition systems.
This section consolidates the scope of applicability, clarifies the limitations of the framework, and positions the contribution conceptually.

10.1 Restriction to SLP-Definable Transitions

Atomic normalization applies strictly to transitions representable as finite acyclic dependency graphs over a fixed basis of total functions.
In particular, the framework does **not** extend to deterministic transitions exhibiting:

- data-dependent structural branching within a single transition,
- variable numbers of executed operations depending on input,
- dynamic loop structures inside a single semantic step,
- interpreter-driven execution of encoded programs,
- runtime structural modification of the dependency graph.

If the internal computation of a transition cannot be represented by a single finite static DAG independent of runtime data, the normalization framework presented here does not apply.
This restriction is structural rather than semantic: a transition may remain total and deterministic while failing to satisfy structural staticity.

10.2 No Claim of Universality

CRTS is not introduced as:
• a universal computation model,
• a simulator for arbitrary RAM or Turing machines,
• a replacement for classical machine models,

• a foundation of computability theory.
The normalization procedure applies only to fixed transitions under fixed encodings that admit SLP representation. No claim is made that arbitrary deterministic STS can be normalized in this manner, and no claim is made regarding the computational power of CRTS relative to classical machine models. The framework establishes structural normalization only for the SLP-definable subclass and does not extend to transitions whose internal structure varies with runtime data. The framework therefore has no implications for computability, complexity classes, or the expressive power of machine models.

10.3 Basis Relativity
All structural properties and metrics defined in this work are relative to a fixed atomic basis A. Changing the atomic basis may change:

- atomic length,
- atomic depth,
- structural parallelism.

Therefore:

- no basis-independent minimality is claimed,
- no canonical minimal atomic form across different bases is asserted.

The framework establishes invariance relative to a chosen atomic granularity, not absolute invariance across all possible granularities.

10.4 No Minimality or Optimality Guarantees
The atomic form produced by normalization is:

- semantics-preserving,
- structure-preserving,
- deterministic under fixed conventions.

However, the procedure does **not** guarantee:

- minimal number of atomic operations,
- minimal dependency depth,
- minimal critical path,
- optimal circuit representation,
- optimal hardware realization,
- optimal instruction scheduling or memory usage.

Expansion templates are fixed and uniform; they are not optimized per instance.

10.5 No Time Complexity or Performance Claims
The structural quantities introduced in Section 9 — atomic length, atomic depth, and structural parallelism — are purely structural measures of the normalized dependency DAG.
They are not statements about:

- running time,
- computational complexity classes,
- asymptotic time bounds,
- resource-bounded computation,
- hardware latency, energy, or memory footprint.

Bridging structural metrics to performance or complexity theory requires additional modeling assumptions beyond the scope of this work.

10.6 Finite Encoding Assumption

Applicability to machine models (e.g., RAM or Turing machines) relies on:

- fixed finite encodings of configurations,
- fixed word size,
- bounded memory within a single transition.

The results do not extend to:

- asymptotic families of machines parameterized by unbounded word size,
- dynamically expanding memory structures inside a single transition.

Normalization operates at the level of a single fixed transition rule under fixed encoding.

10.7 Scope of Structural Invariance

Within the SLP-definable subclass, atomic normalization guarantees:

- injective embedding of the original DAG,
- preservation of partial order,
- preservation of dependency depth up to constants,
- preservation of structural parallelism up to constants,
- semantic equivalence of transitions.

It does **not** claim:

- equivalence of sequential execution length,
- invariance under arbitrary refactorizations of the computational basis,
- structural equivalence across unrelated bases.

All invariance results are basis-relative and structurally scoped.

10.8 Conceptual Position

The contribution of this work is structural in nature.
It does not seek to extend classical computability theory, establish complexity-theoretic separations, construct universal models, or derive lower bounds. No claims are made regarding expressive power or asymptotic computational strength.
The framework instead operates at the level of structural representation. It:

- fixes a uniform atomic granularity for structurally static transitions,
- provides a constructive normalization procedure that refines composite operations into atomic substructures while preserving dependency relations,
- induces basis-relative structural metrics defined at that atomic level.

The results therefore concern structural invariance under a fixed atomic basis rather than computational advancement. Within the SLP-definable subclass, atomic normalization yields a deterministic, semantics-preserving refinement of transition representations. No broader theoretical claims are intended.

11. Relation to Existing Representations

This section clarifies how the proposed atomic normalization framework relates to existing representations of computations and dependency structures.
The goal is not to replace or subsume these representations, but to position CRTS within the landscape of structural formalisms. The framework does not introduce a new circuit model or execution semantics; rather, it provides a normalization schema applied to transitions that already admit a finite static dependency graph.

CRTS complements existing representations by refining structural granularity without altering semantics or proposing alternative computational foundations.

11.1 Topological Sorting

A topological sort of a DAG produces a linear extension of its partial order.
Formally, given a dependency $DAG\ H = (V, E)$, a topological ordering yields a sequence

$$v_1, v_2, \ldots, v_n$$

such that all dependencies are respected.
However, topological sorting:

- does not alter the granularity of nodes,
- does not expand composite operations,
- does not refine internal structure,
- does not introduce a uniform atomic basis.

It provides a linear ordering of existing nodes but does not normalize their internal structure.
In contrast, CRTS:

- replaces each composite node with an explicit atomic substructure,
- refines the dependency graph to a fixed atomic granularity,
- then produces a linear routine consistent with that refined structure.

Thus, topological sorting is an ordering procedure, whereas CRTS normalization is a structural refinement followed by ordering.

11.2 Straight-Line Programs (SLP)

Straight-Line Programs (SLP) represent computations as finite acyclic DAGs over a functional basis.
An SLP specifies:

- nodes labeled by operations from a fixed set,
- explicit data dependencies,
- no branching or loops.

SLP is the structural class to which the present normalization applies.
However, SLP permits:

- composite nodes at arbitrary granularity,
- high-level operations treated as atomic symbols,
- variability in internal structural detail.

CRTS does not replace SLP as a representation language.
Rather, it refines SLP by:

- fixing an atomic basis,
- systematically expanding composite nodes,
- enforcing uniform atomic granularity.

In this sense, CRTS operates as a normalization layer over SLP-defined transitions.

11.3 Circuits and Netlists

Boolean circuits and arithmetic circuits represent computations as networks of primitive gates.

Circuits provide:

- an explicit dependency graph,
- low-level operational granularity,
- object-level representation of combinational computation.

However, circuits are typically:

- representations of functions,
- hardware-oriented abstractions,
- not framed as normalization procedures for transition systems.

CRTS differs in that:

- it starts from an abstract transition specification,
- performs a systematic expansion relative to a chosen basis,
- embeds the result into a deterministic transition system framework.

While circuits and atomic DAGs share structural similarity, CRTS is positioned as a normalization method for transition semantics rather than as a hardware description formalism.

11.4 Static Single Assignment (SSA)

Static Single Assignment (SSA) form is a widely used intermediate representation in compilers. SSA provides:

- explicit data dependencies,
- unique assignment per variable,
- a structured representation suitable for optimization.

However, SSA:

- operates at the syntactic program level,
- does not enforce atomic granularity,
- allows composite operations to remain opaque,
- may contain operations whose internal structure is not expanded.

CRTS differs in scope:

- it does not operate at the level of program syntax,
- it does not introduce φ-functions or control-flow constructs,
- it enforces a fixed atomic basis,
- it expands composite operations into atomic substructures.

Thus, while SSA makes data dependencies explicit at the variable level, CRTS refines computational nodes themselves to a uniform atomic level.

11.5 Transition-Level Normalization

The distinguishing feature of CRTS is its positioning at the level of transition semantics. Specifically, CRTS:

- treats a deterministic transition $\Delta: X \to X$ as the primary object,
- refines its dependency structure to atomic granularity,
- preserves semantics and dependency relations,

- induces structural metrics relative to a fixed atomic basis.

Unlike:

- topological sorting (which orders),
- SLP (which represents),
- circuits (which describe objects),
- SSA (which normalizes syntax),

CRTS performs atomic normalization of the transition function itself.

11.6 Summary

The relationship can be summarized as follows:

- **Topological sort**: linear extension of an existing DAG.
- **SLP**: structural representation allowing composite nodes.
- **Circuits**: object-level low-level network representation.
- **SSA**: syntactic normalization at the program level.
- **CRTS**: atomic normalization of transition semantics under a fixed basis.

CRTS does not replace existing representations.
It provides a structural refinement layer that operates at a fixed atomic granularity for SLP-definable deterministic transitions.

12. Conclusion

This work introduced a constructive atomic normalization framework for a structurally restricted subclass of deterministic state-transition systems, namely SLP-definable transitions.
We established that for every transition representable as a finite acyclic dependency graph over a fixed basis, there exists a corresponding atomic representation realized within the CRTS framework. The normalization procedure is finite, effective, and semantics-preserving.
The main contributions can be summarized as follows:

- **Uniform Atomic Level.**
  A fixed atomic basis provides a uniform structural granularity for transition representation. Composite operations are systematically expanded into atomic substructures.
- **Preservation of Structural Dependencies.**
  The original dependency DAG embeds injectively into the atomic DAG. Partial order relations are preserved.
- **Depth Preservation up to Constants.**
  Dependency depth is preserved up to linear factors determined by fixed expansion templates.
- **Semantic Preservation.**
  The normalized CRTS routine computes exactly the same transition function as the original SLP-definable specification.

- **Determinism Under Fixed Conventions.**
  Given fixed traversal rules, expansion templates, and naming conventions, the normalization procedure yields a uniquely determined atomic representation up to renaming.

The induced atomic form enables the definition of structural quantities—atomic length, atomic dependency depth, and structural parallelism—that are invariant relative to the chosen atomic basis. These quantities provide a structurally consistent measurement layer independent of syntactic granularity.

Importantly, the scope of the results is explicitly bounded. The framework:

- does not establish minimality or optimality,
- does not claim universality,
- does not address time complexity or hardware performance,
- does not extend to transitions with dynamic branching or runtime structural variation.

Normalization applies strictly to the SLP-definable subclass of deterministic transitions characterized by finite static dependency structure.

Within this well-defined scope, atomic normalization provides a structurally invariant and semantically faithful refinement of transition representations. It establishes a fixed structural layer at which heterogeneous transition descriptions become comparable under a common atomic granularity.

This concludes the development of the atomic normalization framework for SLP-definable deterministic transitions.

References

1. Bürgisser, P., Clausen, M., Shokrollahi, M. A.
Algebraic Complexity Theory. Springer, 1997.
2. von zur Gathen, J., Gerhard, J.
Modern Computer Algebra. Cambridge University Press, 2013.
3. Wegener, I.
The Complexity of Boolean Functions. Wiley-Teubner, 1987.
4. Arora, S., Barak, B.
Computational Complexity: A Modern Approach. Cambridge University Press, 2009.
5. Brent, R. P.
"The Parallel Evaluation of General Arithmetic Expressions."
Journal of the ACM, 21(2), 1974, pp. 201–206.
6. Valiant, L. G.
"A Bridging Model for Parallel Computation."
Communications of the ACM, 33(8), 1990, pp. 103–111.
7. Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K.
"Efficiently Computing Static Single Assignment Form and the Control Dependence Graph."
ACM Transactions on Programming Languages and Systems, 13(4), 1991.
8. Sipser, M.
Introduction to the Theory of Computation. Cengage, 2012.

9. Cook, S. A., Reckhow, R. A.
“The Relative Efficiency of Propositional Proof Systems.”
Journal of Symbolic Logic, 1979.
10. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.
Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley, 2006.