Repair Algebras and Regenerative Runtimes

A Runtime Semantics for Biological Systems

Adrian Diamond

AAD Systems
adrian@aadsystems.com

September 7, 2026

Abstract

We introduce *Repair Algebras*, a mathematical framework for modeling biological systems as regenerative runtimes whose behavior is governed by deterministic state transitions, repair operations, and homeostatic invariants.

Building upon our previous work on *Executable Gödel Encodings* and *Sigma Runtimes*, we investigate biological systems from the perspective of runtime semantics rather than molecular implementation. Rather than modeling individual biochemical reactions directly, we introduce higher-level algebraic structures that classify repair, degradation, regeneration, and recovery as formally composable runtime transformations.

The objective is not to replace molecular biology, but to provide a semantic framework that captures the invariant organizational principles governing biological repair. This perspective suggests new methods for reasoning about biological computation, runtime verification, and regenerative processes while providing a common mathematical language for future computational and biological systems.

Contents

1 Introduction

Modern biology provides extraordinarily detailed descriptions of molecular mechanisms, including signaling pathways, gene regulation, protein interactions, and cellular repair processes [12, 14]. These descriptions explain how biological systems operate at the biochemical level, yet they do not provide a unified semantic framework describing the computational principles that preserve the long-term organization of living systems.

In computer science, runtime systems are responsible for maintaining program execution through state transitions, resource management, error recovery, and invariant preservation. Modern verified systems further seek to guarantee that computation proceeds while maintaining formally specified correctness properties. These systems are naturally described using operational semantics, state-transition models, and formal verification methods [4, 5, 6].

This paper proposes that biological systems may be understood from a similar semantic perspective.

Rather than modeling repair as a collection of independent biochemical reactions, we introduce *Repair Algebras*, a mathematical framework in which repair operations are represented as composable runtime transformations acting upon semantic system states. These repair operations preserve higher-level organizational invariants while allowing local damage, degradation, adaptation, and recovery.

Building upon our previous work on *Executable Gödel Encodings* [2] and *Sigma Runtimes* [3], we extend runtime semantics beyond logical computation toward regenerative systems capable of maintaining their own semantic integrity through formally defined repair operations.

The objective of this work is not to replace molecular biology with an abstract computational model. Rather, we seek a semantic framework that characterizes the invariant organizational principles governing biological repair independently of the underlying biochemical implementation. This higher level of abstraction permits biological systems to be studied as verified regenerative runtimes whose behavior is governed by algebraic repair operations and semantic state transitions.

If successful, this framework provides a common mathematical language for reasoning about biological computation, runtime verification, regenerative software systems, and future adaptive computational architectures.

2 Motivation

2.1 Why Biological Systems Require Higher-Level Runtime Models

Biological systems are commonly described through detailed accounts of molecular interactions, biochemical pathways, gene regulation, signaling networks, and cellular processes. These descriptions are essential for understanding biological mechanism, but they do not by themselves provide a general account of how biological organization persists through damage, perturbation, adaptation, and repair.

A living system is not defined solely by the instantaneous configuration of its molecular components. It also exhibits higher-level properties that persist across changes in those components. Cells replace proteins, tissues undergo turnover, regulatory states shift, and local structures are repaired, while the larger organization of the system may remain functionally stable [10, 11, 13].

This suggests that biological description requires a distinction between microscopic configuration and higher-level semantic state.

Let

$$S_{\text{micro}}$$

denote a space of detailed physical or molecular configurations, and let

$$S$$

denote a higher-level state space containing the distinctions relevant to the regenerative behavior under study.

A semantic abstraction may be represented by a map

$$\alpha : S_{\text{micro}} \to S.$$

The map α identifies microscopic configurations that are treated as equivalent at the chosen runtime level of description.

Thus, it is possible that

$$x \neq y$$

in S_{micro}, while

$$\alpha(x) = \alpha(y).$$

The higher-level model therefore does not attempt to reconstruct every microscopic distinction. Instead, it preserves those distinctions that are relevant to viability, organization, function, repair, and regeneration.

This abstraction is particularly important when modeling biological repair. A system that successfully recovers from damage need not return to the exact microscopic configuration that preceded the perturbation. What matters may instead be the restoration of selected organizational properties [14, 15].

Accordingly, we distinguish the internal runtime state

$$s \in S$$

from a semantic observation

$$\eta(s) \in \Omega,$$

where

$$\eta : S \to \Omega$$

records the higher-level properties regarded as significant by the model.

The observation map need not be injective. Distinct states may satisfy

$$\eta(s) = \eta(t)$$

while

$$s \neq t.$$

Such states are different internal realizations of the same selected organizational condition.

This distinction permits regeneration to be represented without requiring microscopic reversal.

A successful recovery may therefore have the form

$$s \xrightarrow{d} d(s) \xrightarrow{r} s',$$

where

$$s' \neq s,$$

but

$$\eta(s') = \eta(s).$$

The system has changed, yet the selected semantic organization has been restored.

This motivates a runtime interpretation of biological organization. Instead of viewing the system only as a collection of molecular reactions, we also model it as a state-transforming process subject to invariants, viability conditions, damage events, repair operations, and recovery criteria.

At this level, the central objects are not individual molecules but relations between states and transformations:

$$\text{state}, \quad \text{damage}, \quad \text{repair}, \quad \text{viability}, \quad \text{invariant}, \quad \text{recovery}.$$

These objects support questions that are difficult to formulate purely at the molecular level. For example:

- Which properties of a biological system remain invariant during ordinary operation?
- Which properties may be temporarily lost after damage?
- Which damaged states remain recoverable?
- Which transformations restore a desired organizational state?
- Can several repair operations be composed into a valid recovery pathway?
- Can a system recover its function without returning to its original internal configuration?

Such questions concern the organization of transformations rather than the identity of individual molecular events.

A higher-level runtime model is therefore not a replacement for molecular biology. It is an additional mathematical layer intended to describe the structure of biological persistence and recovery.

Molecular descriptions answer questions about implementation:

$$\text{How is a biological process physically realized?}$$

A regenerative runtime model instead asks:

$$\boxed{\text{Which state transformations preserve or restore the organization required for continued operation?}}$$

The remainder of this paper develops an algebraic and operational framework for answering this second class of questions.

2.2 Limitations of Purely Molecular Descriptions

Molecular descriptions are indispensable for explaining biological mechanism. They identify the physical components of a system, characterize biochemical interactions, and reveal the causal processes through which biological change is realized.

However, a purely molecular description does not automatically provide a formal account of which aspects of biological organization are essential, which changes are admissible, which perturbations constitute damage, or which transformations count as successful recovery.

The limitation is therefore not one of detail, but of level of description.

A molecular model may represent a biological state by a configuration

$$m \in S_{\text{micro}},$$

where S_{micro} records a large number of physical variables.

Such a representation may be sufficiently detailed to distinguish states that are physically different even when those differences are irrelevant to the organizational property under investigation.

Suppose

$$m_1, m_2 \in S_{\text{micro}}$$

are distinct molecular configurations satisfying

$$m_1 \neq m_2.$$

If both configurations realize the same higher-level biological organization, then a model concerned with repair or regeneration may wish to identify them through an abstraction map

$$\alpha : S_{\text{micro}} \to S$$

such that

$$\alpha(m_1) = \alpha(m_2).$$

The existence of such equivalence is important because biological recovery often permits multiple internal realizations of the same functional or organizational state.

A purely microscopic description does not, by itself, specify which of these differences should be ignored.

The same difficulty appears in the definition of damage.

At the molecular level, every transformation alters some physical variables. But not every molecular alteration is biologically damaging. A system may undergo substantial molecular turnover while remaining viable and preserving its organization.

Consequently, a transformation

$$d : S_{\text{micro}} \to S_{\text{micro}}$$

cannot be classified as damage merely because

$$d(m) \neq m.$$

Damage is necessarily relative to additional criteria.

These criteria may include viability,

$$d(m) \notin V,$$

loss of a semantic property,

$$\eta(d(m)) \neq \eta(m),$$

departure from a homeostatic region, or violation of some other designated invariant.

Thus damage is not simply physical change. It is change interpreted relative to the organizational structure of the model.

The same is true of repair.

A molecular transformation may reverse some physical consequence of damage without restoring the system's selected functional organization. Conversely, a system may regain that organization through a molecular configuration different from the original one.

Therefore the condition

$$r(d(m)) = m$$

is generally too strong as a universal definition of successful repair.

A more appropriate higher-level condition may instead be

$$r(d(s)) \in V$$

together with

$$\eta(r(d(s))) = \eta(s).$$

This distinguishes exact physical restoration from semantic restoration.

Purely molecular descriptions also make it difficult to isolate compositional structure.

Biological recovery frequently consists of sequences of transformations. If

$$r_1, r_2, \ldots, r_n$$

are repair operations, then the relevant mathematical object may be the composite

$$r_n \circ \cdots \circ r_2 \circ r_1.$$

The success of this composite may depend on order, intermediate states, admissibility constraints, and interactions among the operations.

These properties are naturally algebraic and operational. They are not captured merely by listing the molecular mechanisms participating in the process.

A further limitation concerns cross-scale comparison.

The same regenerative phenomenon may be described at molecular, cellular, tissue, or organismal levels. The corresponding state spaces may differ substantially, yet the models may share a common repair structure.

To express this relationship formally, one needs mappings between descriptions that preserve selected features such as viability, repair, semantic equivalence, or recoverability.

This motivates the use of structure-preserving maps rather than direct identity between descriptions.

The distinction can be summarized as follows:

molecular description explains implementation, while runtime semantics explains organizational behavior under transformation.

Neither level eliminates the need for the other.

Molecular biology specifies the mechanisms through which repair is physically realized. A higher-level runtime model specifies the formal conditions under which a sequence of changes counts as damage, compensation, repair, adaptation, or regeneration.

The purpose of the present framework is therefore not to reduce biology to a simpler vocabulary. It is to introduce mathematical structure at a level where biological persistence can be described in terms of states, transformations, equivalence classes, invariants, and recovery pathways.

This higher-level perspective also creates a bridge to verified computation.

Once the relevant biological distinctions are represented by explicit state spaces, transformations, and invariants, questions of biological recovery can be translated into formal problems of reachability, composition, preservation, and verification.

The next section develops that bridge by relating regenerative runtime semantics to executable encodings and the computational frameworks from which the present theory emerged.

3 From Executable Gödel Encodings to Biological Runtimes

3.1 Executable Encodings

The higher-level runtime perspective introduced above requires a precise relationship between abstract structure and machine-executable representation. The immediate mathematical precedent for this relationship is the use of executable encodings: representations in which formal objects are not merely assigned codes, but can be reconstructed, transformed, and verified through computation.

Classical Gödel numbering establishes that syntactic objects may be encoded arithmetically [1]. Let

$$\mathcal{E}$$

denote a class of structured formal objects and let

$$\mathrm{enc} : \mathcal{E} \to \mathbb{N}$$

assign each object a numerical representation.

For execution, however, representation alone is insufficient. A computational system must also recover the represented structure. We therefore require a decoding operation

$$\mathrm{dec} : \mathbb{N} \rightharpoonup \mathcal{E}$$

satisfying the round-trip property

$$\mathrm{dec}(\mathrm{enc}(x)) = x$$

for every admissible

$$x \in \mathcal{E}.$$

The pair

$$(\mathrm{enc}, \mathrm{dec})$$

therefore establishes a verified correspondence between structured objects and their executable representations.

Definition 3.1 (Executable Encoding)**.** Let X be a structured state space and let C be a computational representation space.

An *executable encoding* of X into C consists of maps

$$\mathrm{enc} : X \to C$$

and

$$\mathrm{dec} : C \rightharpoonup X$$

such that

$$\mathrm{dec}(\mathrm{enc}(x)) = x$$

for every

$$x \in X.$$

The significance of executable encoding appears when transformations are transported to the encoded domain.

Let

$$f : X \to X$$

be a transformation on structured states. An encoded realization of f is a transformation

$$\widehat{f} : C \rightharpoonup C$$

satisfying

$$\widehat{f}(\mathrm{enc}(x)) = \mathrm{enc}(f(x))$$

for every admissible $x \in X$ for which the encoded computation is defined.

Applying dec and using the round-trip property then gives

$$\mathrm{dec}\left(\widehat{f}(\mathrm{enc}(x))\right) = f(x).$$

Equivalently, on valid encoded states,

$$\widehat{f} \circ \mathrm{enc} = \mathrm{enc} \circ f.$$

The encoded computation therefore reproduces the transformation defined on the structured domain.

Definition 3.2 (Correct Encoded Realization)**.** Let

$$f : X \to X.$$

An encoded transformation

$$\widehat{f} : C \rightharpoonup C$$

is a *correct encoded realization* of f if

$$\widehat{f}(\mathrm{enc}(x)) = \mathrm{enc}(f(x))$$

for every $x \in X$ on which the encoded computation is defined.

This condition separates representation from semantics. The machine may operate entirely on elements of C, while correctness is stated relative to the structured transformation f.

The same construction applies to compositions.

Proposition 3.3 (Encoded Composition)**.** *Let*

$$f, g : X \to X$$

have correct encoded realizations

$$\widehat{f}, \widehat{g} : C \rightharpoonup C.$$

Suppose the relevant encoded compositions are defined. Then

$$\widehat{g} \circ \widehat{f}$$

correctly realizes

$$g \circ f.$$

Proof. For $x \in X$, correctness of $\widehat{f}$ gives

$$\widehat{f}(\mathrm{enc}(x)) = \mathrm{enc}(f(x))$$

on valid encodings.

Applying $\widehat{g}$,

$$\widehat{g}\left(\widehat{f}(\mathrm{enc}(x))\right) = \widehat{g}(\mathrm{enc}(f(x))).$$

Correctness of $\widehat{g}$ then gives

$$\widehat{g}(\mathrm{enc}(f(x))) = \mathrm{enc}(g(f(x))).$$

Decoding yields

$$\mathrm{dec}\left((\widehat{g} \circ \widehat{f})(\mathrm{enc}(x))\right) = (g \circ f)(x).$$

□

Thus executable encoding preserves not merely individual objects but the behavior of transformation systems.

This observation is central to the development of regenerative runtimes.

Suppose a biological runtime state is represented abstractly by

$$s \in S,$$

with damage and repair transformations

$$d : S \to S, \qquad r : S \to S.$$

If these objects admit executable encodings, then one may seek encoded transformations

$$\widehat{d}, \widehat{r} : C \rightharpoonup C$$

such that

$$\widehat{d}(\mathrm{enc}(s)) = \mathrm{enc}(d(s))$$

and

$$\widehat{r}(\mathrm{enc}(x)) = \mathrm{enc}(r(x)).$$

The composite encoded computation

$$\widehat{r} \circ \widehat{d}$$

then realizes the abstract transformation

$$r \circ d.$$

If successful repair requires

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s),$$

the encoded runtime can in principle be checked against these same conditions after decoding.

The resulting architecture has two distinct correctness requirements.

First, representation correctness requires

$$\mathrm{dec}(\mathrm{enc}(s)) = s.$$

Second, execution correctness requires

$$\widehat{f}(\mathrm{enc}(s)) = \mathrm{enc}(f(s)).$$

By representation correctness, this implies

$$\mathrm{dec}\left(\widehat{f}(\mathrm{enc}(s))\right) = f(s).$$

Together, these properties ensure that computation on encoded representations agrees with the intended structured semantics.

Definition 3.4 (Semantics-Preserving Executable Encoding)**.** Let

$$\eta : X \to \Omega$$

be a semantic observation map.

An executable encoding is *semantics-preserving* for a transformation f when its encoded realization $\widehat{f}$ satisfies

$$\eta\left(\mathrm{dec}\left(\widehat{f}(\mathrm{enc}(x))\right)\right) = \eta(f(x))$$

for every admissible $x \in X$.

For a correct encoded realization, this equality follows immediately.

Proposition 3.5 (Correct Execution Preserves Observable Semantics)**.** *If $\widehat{f}$ is a correct encoded realization of f, then for every semantic observation map*

$$\eta : X \to \Omega,$$

$$\eta\left(\mathrm{dec}\left(\widehat{f}(\mathrm{enc}(x))\right)\right) = \eta(f(x)).$$

Proof. Correctness gives

$$\mathrm{dec}\left(\widehat{f}(\mathrm{enc}(x))\right) = f(x).$$

Applying η to both sides proves the result. □

The importance of this construction is methodological. Encoding forces the objects, transformations, and semantic conditions of a theory to be stated with sufficient precision for computation.

A machine cannot execute an informal notion of repair. It requires a representation of the state, a representation of the transformation, a rule for applying that transformation, and a criterion by which the resulting state can be evaluated.

Thus the transition from mathematical description to executable encoding introduces a stronger operational requirement:

$$\boxed{\text{representation} \longrightarrow \text{transformation} \longrightarrow \text{execution} \longrightarrow \text{semantic verification.}}$$

The regenerative framework developed in this paper extends this principle from encoded logical syntax to state-transforming systems. The relevant objects are no longer only expressions and their numerical representations, but states, damage transformations, repair operations, invariants, and recovery pathways.

The next subsection introduces the intermediate runtime architecture through which these executable representations are organized into persistent state-transforming systems.

3.2 Sigma Runtimes

Executable encodings provide a correspondence between structured objects and machine-operable representations. A runtime requires an additional layer: encoded objects must participate in persistent state transitions whose behavior is governed by explicit operational rules.

We call this intermediate architecture a *Sigma runtime*.

The purpose of the Sigma runtime is to separate three levels that are often conflated:

$$\text{representation}, \qquad \text{state transformation}, \qquad \text{semantic interpretation}.$$

An encoded state may change during execution while selected properties of its interpretation remain invariant. Conversely, a computation may be syntactically valid while producing a state that violates the semantic conditions required by the system.

The runtime therefore mediates between executable representation and semantic correctness.

Definition 3.6 (Sigma Runtime)**.** A *Sigma runtime* is a tuple

$$\Sigma = (X, C, \mathrm{enc}, \mathrm{dec}, \mathcal{T}, \widehat{\mathcal{T}}, \Omega, \eta)$$

where:

- X is a structured state space;
- C is an executable representation space;
- $$\mathrm{enc} : X \to C$$ is an encoding map;
- $$\mathrm{dec} : C \rightharpoonup X$$ is a decoding map;
- $\mathcal{T}$ is a collection of structured state transformations;
- $\widehat{\mathcal{T}}$ is a corresponding collection of encoded transformations;
- Ω is a semantic observation space;
- $$\eta : X \to \Omega$$ is a semantic observation map.

The runtime is representation-correct when

$$\mathrm{dec}(\mathrm{enc}(x)) = x$$

for every admissible

$$x \in X.$$

For each

$$\tau \in \mathcal{T},$$

the corresponding encoded transformation

$$\widehat{\tau} \in \widehat{\mathcal{T}}$$

is execution-correct when

$$\widehat{\tau}(\mathrm{enc}(x)) = \mathrm{enc}(\tau(x))$$

whenever the transformation is defined.

By representation correctness, execution correctness implies

$$\mathrm{dec}\left(\widehat{\tau}(\mathrm{enc}(x))\right) = \tau(x).$$

Thus the diagram

$$\begin{array}{ccc} X & \xrightarrow{\tau} & X \\ \downarrow_{\mathrm{enc}} & & \downarrow_{\mathrm{enc}} \\ C & \xrightarrow{\widehat{\tau}} & C \end{array}$$

commutes on valid encoded states.

The runtime may therefore execute in the representation space C while its correctness is stated in the structured state space X.

Definition 3.7 (Sigma Runtime State)**.** A *Sigma runtime state* is a pair

$$(x, c) \in X \times C$$

satisfying

$$c = \mathrm{enc}(x).$$

The state is *representation-consistent* when

$$\mathrm{dec}(c) = x.$$

The distinction between x and c is conceptually important. The runtime operates on a representation, while the theory assigns meaning to the structure represented by that value.

A transition

$$c \xrightarrow{\widehat{\tau}} c'$$

therefore corresponds to the structured transition

$$x \xrightarrow{\tau} x'$$

when

$$x = \mathrm{dec}(c), \qquad x' = \mathrm{dec}(c'),$$

and

$$x' = \tau(x).$$

Definition 3.8 (Sigma Execution)**.** Let

$$\tau_1, \ldots, \tau_n \in \mathcal{T}$$

with encoded realizations

$$\widehat{\tau}_1, \ldots, \widehat{\tau}_n.$$

A finite *Sigma execution* from x_0 is an encoded trajectory

$$c_0 \xrightarrow{\widehat{\tau}_1} c_1 \xrightarrow{\widehat{\tau}_2} \cdots \xrightarrow{\widehat{\tau}_n} c_n$$

such that

$$c_0 = \mathrm{enc}(x_0)$$

and every c_i is decodable.

Its associated structured trajectory is

$$x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n,$$

where

$$x_i = \mathrm{dec}(c_i).$$

A correct Sigma runtime must preserve correspondence throughout the entire execution, not merely at its initial and terminal states.

Theorem 3.9 (Execution Correspondence)**.** *Suppose each encoded transformation*

$$\widehat{\tau}_i$$

correctly realizes

$$\tau_i.$$

Then for every finite Sigma execution,

$$c_i = \mathrm{enc}(x_i)$$

and

$$x_i = (\tau_i \circ \cdots \circ \tau_1)(x_0)$$

for every

$$1 \leq i \leq n.$$

Proof. Proceed by induction on i.

For $i = 1$, the definition of Sigma execution gives

$$c_1 = \widehat{\tau}_1(c_0) = \widehat{\tau}_1(\mathrm{enc}(x_0)).$$

Execution correctness therefore gives

$$c_1 = \mathrm{enc}(\tau_1(x_0)).$$

Since

$$x_1 = \mathrm{dec}(c_1),$$

representation correctness yields

$$x_1 = \tau_1(x_0),$$

and hence

$$c_1 = \mathrm{enc}(x_1).$$

Assume the result holds for $i = k$. Then

$$c_k = \mathrm{enc}(x_k).$$

The next encoded transition gives

$$c_{k+1} = \widehat{\tau}_{k+1}(c_k) = \widehat{\tau}_{k+1}(\mathrm{enc}(x_k)).$$

By execution correctness,

$$c_{k+1} = \mathrm{enc}(\tau_{k+1}(x_k)).$$

Therefore

$$x_{k+1} = \mathrm{dec}(c_{k+1}) = \tau_{k+1}(x_k),$$

and

$$c_{k+1} = \mathrm{enc}(x_{k+1}).$$

By the induction hypothesis,

$$x_k = (\tau_k \circ \cdots \circ \tau_1)(x_0),$$

so

$$x_{k+1} = (\tau_{k+1} \circ \tau_k \circ \cdots \circ \tau_1)(x_0).$$

Thus both claims hold for every $1 \leq i \leq n$. □

Execution correspondence establishes structural correctness, but a runtime may require stronger semantic guarantees.

Let

$$\eta : X \to \Omega$$

be the semantic observation map.

A transformation

$$\tau \in \mathcal{T}$$

is semantics-preserving on a region $U \subseteq X$ when

$$\eta(\tau(x)) = \eta(x)$$

for every

$$x \in U$$

for which $\tau(x)$ is defined.

Definition 3.10 (Semantic Runtime Invariant)**.** An observable

$$I : X \to \Lambda$$

is a *semantic runtime invariant* for a transition class

$$\mathcal{T}_I \subseteq \mathcal{T}$$

on $U \subseteq X$ if

$$I(\tau(x)) = I(x)$$

for every

$$\tau \in \mathcal{T}_I$$

and every admissible

$$x \in U.$$

Proposition 3.11 (Invariant Preservation through Sigma Execution)**.** *Suppose*

$$I : X \to \Lambda$$

is invariant under every transformation

$$\tau_i$$

occurring in a Sigma execution.

Then

$$I(x_n) = I(x_0).$$

Proof. For every transition,

$$I(x_i) = I(x_{i-1}).$$

Repeated substitution gives

$$I(x_n) = I(x_{n-1}) = \cdots = I(x_0).$$

□

This result exposes the central architectural role of the runtime.

The encoded states themselves may change:

$$c_0 \neq c_1 \neq \cdots \neq c_n,$$

and the corresponding structured states may also change:

$$x_0 \neq x_1 \neq \cdots \neq x_n.$$

Nevertheless, a selected invariant may satisfy

$$I(x_0) = I(x_1) = \cdots = I(x_n).$$

Thus persistence need not mean persistence of state.

It may instead mean persistence of a property through state transformation.

This principle becomes especially important when the runtime is extended to repair. A damaged state may temporarily leave a desired semantic class and later return to it.

Suppose

$$x_0 \xrightarrow{d} x_1 \xrightarrow{r_1} \cdots \xrightarrow{r_n} x_{n+1}.$$

It is possible that

$$\eta(x_1) \neq \eta(x_0),$$

while

$$\eta(x_{n+1}) = \eta(x_0).$$

The relevant semantic property is therefore not continuously preserved but eventually restored.

This distinction separates two runtime behaviors:

preservation :	an invariant survives every transition,
recovery :	an invariant may be lost and is subsequently restored.

The second behavior provides the conceptual transition from Sigma runtimes to regenerative runtimes.

A regenerative runtime augments the Sigma architecture with explicit notions of viability, damage, repair, recovery, and policies governing state transitions.

The progression is therefore

$$\boxed{\text{executable encoding} \longrightarrow \text{Sigma runtime} \longrightarrow \text{regenerative runtime.}}$$

Executable encoding establishes machine-operable representation. The Sigma runtime establishes verified state transformation. The regenerative runtime adds the structure required to characterize damage and restoration through execution.

The next subsection isolates the semantic layer of this architecture and formalizes the relationship between runtime transitions and the meanings assigned to runtime states.

3.3 Runtime Semantics

A runtime specifies how states change through execution. Runtime semantics specifies what those states and transitions mean.

This distinction is necessary because the operational identity of a state is generally finer than its semantic identity. Two runtime states may differ internally while representing the same relevant organizational condition. Conversely, a small operational change may cross a semantic boundary that is significant to the system.

Let

$$S$$

be a runtime state space and let

$$\Omega$$

be a semantic space.

A semantic interpretation is represented by a map

$$\eta : S \to \Omega.$$

The value

$$\eta(s)$$

is the semantic signature associated with runtime state s.

Definition 3.12 (Runtime Semantic Equivalence)**.** For

$$s, t \in S,$$

define

$$s \sim_\eta t$$

if and only if

$$\eta(s) = \eta(t).$$

The relation $\sim_\eta$ partitions the runtime state space into semantic equivalence classes.

For

$$s \in S,$$

write

$$[s]_\eta = \{t \in S : \eta(t) = \eta(s)\}.$$

The class $[s]_\eta$ contains all runtime states that are semantically indistinguishable relative to the observation map η.

Proposition 3.13 (Semantic Equivalence Is an Equivalence Relation)**.** *The relation*

$$\sim_\eta$$

is reflexive, symmetric, and transitive.

Proof. For reflexivity,

$$\eta(s) = \eta(s),$$

so

$$s \sim_\eta s.$$

For symmetry, if

$$s \sim_\eta t,$$

then

$$\eta(s) = \eta(t).$$

Hence

$$\eta(t) = \eta(s),$$

and therefore

$$t \sim_\eta s.$$

For transitivity, suppose

$$s \sim_\eta t$$

and

$$t \sim_\eta u.$$

Then

$$\eta(s) = \eta(t)$$

and

$$\eta(t) = \eta(u),$$

so

$$\eta(s) = \eta(u).$$

Therefore

$$s \sim_\eta u.$$

□

The quotient

$$S/\sim_\eta$$

therefore represents the runtime state space after distinctions irrelevant to the chosen semantics have been removed.

This quotient perspective is important for regenerative systems. Recovery need not return a damaged system to its original runtime state. It may instead return the system to the semantic equivalence class of that state.

Thus

$$s' \neq s$$

may coexist with

$$[s']_\eta = [s]_\eta.$$

The runtime has changed structurally while preserving or recovering the selected semantics.

Runtime transformations interact with this quotient structure.

Let

$$\tau : S \rightharpoonup S$$

be a runtime transformation.

Definition 3.14 (Semantics-Preserving Transformation)**.** A transformation τ is *semantics-preserving on* $U \subseteq S$ if

$$\eta(\tau(s)) = \eta(s)$$

for every

$$s \in U$$

for which $\tau(s)$ is defined.

Equivalently,

$$\tau(s) \sim_\eta s.$$

A semantics-preserving transformation may therefore change the runtime state without changing its semantic equivalence class.

Proposition 3.15 (Composition of Semantics-Preserving Transformations)**.** *Let*

$$\tau_1, \tau_2 : S \rightharpoonup S$$

be semantics-preserving on a region U*, and suppose their composition is defined and remains within the region on which semantic preservation holds.*

Then

$$\tau_2 \circ \tau_1$$

is semantics-preserving.

Proof. For any admissible $s \in U$,

$$\eta(\tau_1(s)) = \eta(s).$$

Since τ_2 is also semantics-preserving,

$$\eta(\tau_2(\tau_1(s))) = \eta(\tau_1(s)).$$

Therefore

$$\eta((\tau_2 \circ \tau_1)(s)) = \eta(s).$$

□

Hence semantics-preserving transformations are closed under admissible composition.

Not every runtime transformation must preserve semantics.

A damage transformation may satisfy

$$\eta(d(s)) \neq \eta(s),$$

and a subsequent repair transformation may satisfy

$$\eta(r(d(s))) = \eta(s).$$

The complete execution

$$s \xrightarrow{d} d(s) \xrightarrow{r} r(d(s))$$

therefore leaves and later re-enters the original semantic equivalence class.

This motivates a distinction between semantic preservation and semantic restoration.

Definition 3.16 (Semantic Preservation)**.** An execution

$$s_0 \to s_1 \to \cdots \to s_n$$

preserves semantics when

$$\eta(s_i) = \eta(s_0)$$

for every

$$0 \leq i \leq n.$$

Definition 3.17 (Semantic Restoration)**.** An execution

$$s_0 \to s_1 \to \cdots \to s_n$$

restores the semantics of s_0 when

$$\eta(s_n) = \eta(s_0),$$

without requiring

$$\eta(s_i) = \eta(s_0)$$

for intermediate states.

Semantic preservation is therefore stronger than terminal semantic restoration.

Proposition 3.18 (Semantic Preservation Implies Semantic Restoration)**.** *If an execution preserves the semantics of its initial state, then it restores those semantics at its terminal state.*

Proof. Semantic preservation gives

$$\eta(s_i) = \eta(s_0)$$

for every i. In particular,

$$\eta(s_n) = \eta(s_0).$$

□

The converse does not generally hold.

This distinction provides the semantic foundation for repair. Damage may break an invariant; repair may restore it. The defining feature of regeneration is therefore not necessarily uninterrupted invariance, but recoverability of selected invariants through transformation.

The semantic map may itself contain several observables.

Suppose

$$\eta = (\eta_1, \ldots, \eta_k),$$

where

$$\eta_j : S \to \Omega_j.$$

A transformation may preserve some components while changing others.

For example,

$$\eta_1(\tau(s)) = \eta_1(s)$$

may hold while

$$\eta_2(\tau(s)) \neq \eta_2(s).$$

Thus semantic preservation is relative to the observables selected by the model.

Definition 3.19 (Semantic Projection)**.** For

$$J \subseteq \{1, \ldots, k\},$$

define

$$\eta_J(s) = (\eta_j(s))_{j \in J}.$$

A transformation is *J-semantics-preserving* when

$$\eta_J(\tau(s)) = \eta_J(s)$$

on its admissible domain.

This permits the runtime to distinguish essential invariants from properties that may vary during adaptation or repair.

The semantic structure therefore determines which differences between runtime states matter.

If

$$\eta(s) = \eta(t),$$

then the model deliberately treats s and t as equivalent with respect to the selected semantics, even if they differ in other respects.

This does not assert that the physical states are identical. It asserts only that the distinctions between them are outside the semantic resolution of the current model.

The relationship between runtime execution and semantics can consequently be expressed as

runtime states $\longrightarrow$ state transformations $\longrightarrow$ semantic observation $\longrightarrow$ invariance or restoration.

The runtime describes what changes.

The semantic map specifies which properties of those changes are relevant.

The invariants identify what must remain unchanged or be recovered.

This separation allows the same transformation system to be studied under different semantic interpretations and allows different internal states to realize the same higher-level organization.

The next subsection develops this relationship across levels of representation. It introduces semantic descent: the problem of determining which structures and invariants survive when a theory is translated from a higher-level semantic description into progressively more concrete representations and executable forms.

3.4 Semantic Descent

The preceding sections distinguish structured states from executable representations and runtime states from their semantic interpretations. These distinctions lead to a more general question: what happens to meaning when a theory is translated through successive levels of representation?

A high-level theory may begin with objects whose interpretation is explicit. To execute that theory computationally, those objects must eventually be represented by lower-level structures, encoded data, and machine-operable states.

We call this progression *semantic descent*.

Definition 3.20 (Semantic Descent)**.** Let

$$X_0, X_1, \ldots, X_n$$

be state or representation spaces and let

$$\phi_i : X_i \to X_{i+1} \qquad (0 \leq i < n)$$

be representation maps.

A *semantic descent* is the sequence

$$X_0 \xrightarrow{\phi_0} X_1 \xrightarrow{\phi_1} \cdots \xrightarrow{\phi_{n-1}} X_n,$$

where X_0 represents a higher-level semantic description and successive spaces provide progressively more concrete representations of the same modeled system.

The composite descent map is

$$\Phi = \phi_{n-1} \circ \cdots \circ \phi_1 \circ \phi_0.$$

Thus an abstract state

$$x \in X_0$$

is represented at the terminal level by

$$\Phi(x) \in X_n.$$

Semantic descent becomes mathematically significant when one asks which properties of x remain recoverable from, or preserved by, its lower-level representation.

Let

$$\eta_0 : X_0 \to \Omega$$

be a semantic observation at the abstract level.

A corresponding observation

$$\eta_i : X_i \to \Omega$$

may exist at each lower level.

Definition 3.21 (Semantic Compatibility)**.** A representation map

$$\phi_i : X_i \to X_{i+1}$$

is *semantically compatible* with observation maps

$$\eta_i : X_i \to \Omega$$

and

$$\eta_{i+1} : X_{i+1} \to \Omega$$

if

$$\eta_{i+1}(\phi_i(x)) = \eta_i(x)$$

for every admissible

$$x \in X_i.$$

Semantic compatibility means that the representation may change while the selected semantic observable remains invariant.

Proposition 3.22 (Semantic Compatibility under Descent)**.** *Suppose every representation map*

$$\phi_i : X_i \to X_{i+1}$$

is semantically compatible. Then

$$\eta_n(\Phi(x)) = \eta_0(x)$$

for every admissible

$$x \in X_0.$$

Proof. Semantic compatibility gives

$$\eta_1(\phi_0(x)) = \eta_0(x).$$

Applying compatibility at the next level,

$$\eta_2(\phi_1(\phi_0(x))) = \eta_1(\phi_0(x)).$$

Continuing through the descent yields

$$\eta_n\left(\phi_{n-1} \circ \cdots \circ \phi_0(x)\right) = \eta_0(x).$$

Since

$$\Phi = \phi_{n-1} \circ \cdots \circ \phi_0,$$

the result follows. □

Thus semantic meaning can remain invariant even while its representation changes repeatedly. The same principle applies to equivalence relations.
Suppose

$$x \sim_{\eta_0} y.$$

If semantic compatibility holds throughout the descent, then

$$\eta_n(\Phi(x)) = \eta_0(x) = \eta_0(y) = \eta_n(\Phi(y)).$$

Therefore

$$\Phi(x) \sim_{\eta_n} \Phi(y).$$

Theorem 3.23 (Preservation of Semantic Equivalence under Descent)**.** *If every stage of a semantic descent is semantically compatible, then*

$$x \sim_{\eta_0} y \quad \Longrightarrow \quad \Phi(x) \sim_{\eta_n} \Phi(y).$$

Proof. If

$$x \sim_{\eta_0} y,$$

then

$$\eta_0(x) = \eta_0(y).$$

By semantic compatibility of the complete descent,

$$\eta_n(\Phi(x)) = \eta_0(x)$$

and

$$\eta_n(\Phi(y)) = \eta_0(y).$$

Hence

$$\eta_n(\Phi(x)) = \eta_n(\Phi(y)),$$

so

$$\Phi(x) \sim_{\eta_n} \Phi(y).$$

□

The converse generally requires stronger conditions.

A lower-level representation may collapse distinctions that were visible at the higher level. Consequently,

$$\Phi(x) \sim_{\eta_n} \Phi(y)$$

need not imply

$$x \sim_{\eta_0} y$$

unless the descent reflects the relevant semantics.

Definition 3.24 (Semantic Reflection)**.** A representation map

$$\phi_i : X_i \to X_{i+1}$$

reflects semantic equivalence if

$$\phi_i(x) \sim_{\eta_{i+1}} \phi_i(y) \quad \Longrightarrow \quad x \sim_{\eta_i} y.$$

A representation that both preserves and reflects semantic equivalence provides a stronger correspondence between levels.

This distinction becomes important when executable representations are used for verification. Preservation ensures that semantically equivalent high-level states remain equivalent after translation. Reflection ensures that semantic equivalence observed at the lower level is not merely an artifact of information loss.

Semantic descent also applies to transformations.

Let

$$\tau_i : X_i \rightharpoonup X_i$$

represent a transformation at level i.

A lower-level transformation

$$\tau_{i+1} : X_{i+1} \rightharpoonup X_{i+1}$$

correctly realizes τ_i when

$$\phi_i(\tau_i(x)) = \tau_{i+1}(\phi_i(x))$$

for every admissible x.

The corresponding diagram

$$\begin{array}{ccc} X_i & \xrightarrow{\tau_i} & X_i \\ \downarrow{\scriptstyle \phi_i} & & \downarrow{\scriptstyle \phi_i} \\ X_{i+1} & \xrightarrow{\tau_{i+1}} & X_{i+1} \end{array}$$

then commutes.

Definition 3.25 (Transformation-Compatible Descent)**.** A semantic descent is *transformation-compatible* for a family of transformations when every high-level transformation has a corresponding lower-level realization satisfying

$$\phi_i \circ \tau_i = \tau_{i+1} \circ \phi_i$$

on the relevant domain.

This condition ensures that descent preserves not only states but their dynamics.

Theorem 3.26 (Execution Preservation under Semantic Descent)**.** *Suppose a semantic descent is transformation-compatible and semantically compatible.*

Let

$$x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_m} x_m$$

be a high-level execution.

Then the descended states

$$\Phi(x_0), \Phi(x_1), \ldots, \Phi(x_m)$$

form a corresponding lower-level execution, and

$$\eta_n(\Phi(x_j)) = \eta_0(x_j)$$

for every

$$0 \leq j \leq m.$$

Proof. Transformation compatibility ensures that each high-level transition is realized by the corresponding descended transformation. Hence the descended states form an execution.

Semantic compatibility gives

$$\eta_n(\Phi(x_j)) = \eta_0(x_j)$$

for each state in the execution. □

The theorem identifies the condition required for a semantic theory to survive implementation: both its states and its transformations must descend correctly.

This provides a general interpretation of executable encoding.

An encoding map

$$\mathrm{enc} : X \to C$$

is one stage of semantic descent.

A compiler lowering a structured representation into another intermediate representation is another.

A machine representation provides a still lower level.

The complete computational path may therefore be modeled as

$$X_{\text{semantic}} \longrightarrow X_{\text{formal}} \longrightarrow X_{\text{encoded}} \longrightarrow X_{\text{runtime}} \longrightarrow X_{\text{machine}}.$$

The representations change at every stage.

The central verification problem is whether the required invariants do.

This motivates the principle

semantic descent is correct when the distinctions required by the theory survive the transformations required for

For regenerative systems, this principle acquires an additional dimension.

Damage may intentionally break an invariant:

$$\eta(d(s)) \neq \eta(s).$$

Repair must then restore it:

$$\eta(r(d(s))) = \eta(s).$$

A correct descent of the regenerative theory must therefore preserve not only invariant states but the distinction between preservation, violation, and restoration.

If

$$\widehat{d}$$

and

$$\widehat{r}$$

are descended realizations of damage and repair, then a verified executable implementation should satisfy

$$\operatorname{dec}\left(\widehat{r}\left(\widehat{d}(\operatorname{enc}(s))\right)\right) = r(d(s)).$$

Consequently,

$$\eta\left(\operatorname{dec}\left(\widehat{r}\left(\widehat{d}(\operatorname{enc}(s))\right)\right)\right) = \eta(s)$$

whenever the abstract repair is semantically successful.

This is the bridge between executable encoding and regenerative computation.

The central question of the present framework can therefore be stated as follows:

What survives semantic descent through a repair algebra and a regenerative runtime?

The answer developed in the remainder of this paper is expressed through viability, semantic equivalence, repair composition, morphisms, recovery paths, runtime invariants, and verification conditions.

These structures identify what may change, what must remain invariant, what may temporarily fail, and what must ultimately be restored.

The next section introduces the algebraic structure governing those transformations.

4 Repair Algebras

4.1 Definition

A repair algebra describes a system whose states may be perturbed, damaged, and subsequently transformed toward a viable semantic configuration. The framework separates the internal state of the system from the higher-level properties that the system is required to preserve.

Let S be a nonempty set of runtime states. A state $s \in S$ represents the complete configuration of the system at the semantic level selected by the model. Depending on the application, a state may describe a cell, tissue, organ, organism, computational process, or other regenerative system.

Not every state need represent a viable configuration. We therefore distinguish a subset

$$V \subseteq S,$$

called the *viability region*. States in V satisfy the minimal conditions required for continued system operation.

The properties that characterize homeostatic organization are represented by a semantic observation map

$$\eta : S \to \Omega,$$

where Ω is the space of homeostatic signatures. The value $\eta(s)$ records the higher-level properties of s that are relevant to the identity, organization, or function of the system.

More generally, if

$$h_i : S \to \Omega_i$$

is a family of observable invariants indexed by $i \in I$, then the homeostatic signature may be defined as

$$\eta(s) = \big(h_i(s)\big)_{i \in I},$$

with

$$\Omega = \prod_{i \in I} \Omega_i.$$

The map η is not required to be injective. Distinct internal states may therefore realize the same homeostatic organization. This permits repair to restore function and organization without requiring an exact reversal of every microscopic change.

Let

$$\mathrm{End}(S) = \{f \mid f : S \to S\}$$

denote the set of deterministic state transformations on S.

Definition 4.1 (Repair Algebra)**.** A *repair algebra* is a tuple

$$\mathcal{A} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \circ, \mathrm{id}_S)\,,$$

where:

1. S is a nonempty state space;
2. $V \subseteq S$ is the viability region;
3. Ω is a space of homeostatic signatures;
4. $\eta : S \to \Omega$ is the semantic observation map;
5. $\mathcal{D} \subseteq \mathrm{End}(S)$ is a collection of *damage transformations*;
6. $\mathcal{R} \subseteq \mathrm{End}(S)$ is a collection of *repair transformations*;
7. $\mathrm{id}_S \in \mathcal{R}$;
8. whenever $r_1, r_2 \in \mathcal{R}$, their composition satisfies

$$r_2 \circ r_1 \in \mathcal{R}.$$

Consequently,

$$(\mathcal{R}, \circ, \mathrm{id}_S)$$

is a transformation monoid acting on S.

Closure under composition expresses the fact that repair may consist of multiple sequential transformations. A complex recovery pathway may therefore be represented as

$$r_n \circ r_{n-1} \circ \cdots \circ r_1,$$

where each $r_i \in \mathcal{R}$ represents an individual repair stage.

The damage transformations in $\mathcal{D}$ are not assumed to be closed under composition. Damage may arise from heterogeneous causes and need not possess the algebraic regularity required of repair processes. Nevertheless, compositions such as

$$d_n \circ d_{n-1} \circ \cdots \circ d_1$$

remain valid state transformations whenever several perturbations act sequentially.

Definition 4.2 (Semantic Equivalence)**.** For states $s, t \in S$, define

$$s \sim_\eta t \quad \Longleftrightarrow \quad \eta(s) = \eta(t).$$

The relation $\sim_\eta$ is called *semantic equivalence*. It partitions the state space into equivalence classes of states having the same homeostatic signature.

The semantic equivalence class of a state s is

$$[s]_\eta = \{t \in S \mid \eta(t) = \eta(s)\}.$$

A regenerative system need not return to the exact internal state that existed before damage. It may instead reach a different state within the same semantic equivalence class. This distinction separates microscopic restoration from higher-level recovery.

Definition 4.3 (Successful Repair)**.** Let $s \in V$, let $d \in \mathcal{D}$, and let $r \in \mathcal{R}$. The transformation r is a *successful repair of d at s* when

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s).$$

Equivalently,

$$r(d(s)) \in V \cap [s]_\eta.$$

Successful repair therefore has two requirements:

1. the resulting state must return to the viability region; and
2. the resulting state must recover the selected homeostatic signature.

These conditions distinguish repair from a transformation that merely changes a damaged state without restoring its required semantic organization.

Definition 4.4 (Repair Set and Recoverability)**.** For $s \in V$ and $d \in \mathcal{D}$, define the repair set

$$\mathrm{Rep}_{\mathcal{A}}(s, d) = \{r \in \mathcal{R} \mid r(d(s)) \in V \text{ and } \eta(r(d(s))) = \eta(s)\}.$$

The pair (s, d) is called *recoverable* when

$$\mathrm{Rep}_{\mathcal{A}}(s, d) \neq \varnothing.$$

It is called *unrecoverable relative to $\mathcal{A}$* when

$$\mathrm{Rep}_{\mathcal{A}}(s, d) = \varnothing.$$

Recoverability is therefore relative to the available repair algebra. A damaged state that is unrecoverable under one collection of repair transformations may be recoverable under a richer algebra containing additional repair operators.

Definition 4.5 (Exact and Semantic Repair)**.** A successful repair r of d at s is an *exact repair* when

$$r(d(s)) = s.$$

It is a *semantic repair* when

$$r(d(s)) \sim_\eta s,$$

even when

$$r(d(s)) \neq s.$$

Exact repair restores the original runtime state. Semantic repair restores the properties represented by the homeostatic signature while permitting the internal realization of those properties to change.

Remark 4.6. The distinction between exact and semantic repair is essential for biological systems. A repaired system may recover its organization or function without reconstructing every molecular configuration that existed before damage. The repair algebra therefore models preservation at the level of selected invariants rather than assuming microscopic identity.

The definitions above supply the foundational objects used throughout the remainder of the paper. Primitive repair operations are introduced next, followed by their composition rules, morphisms between repair algebras, and the algebraic properties governing regenerative runtime behavior.

4.2 Primitive Operations

The repair algebra introduced above specifies collections of damage and repair transformations, but it does not yet distinguish the elementary operations from which larger repair processes are constructed. We therefore introduce a primitive basis for damage and repair.

Definition 4.7 (Primitive Basis)**.** Let

$$\mathcal{A} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \circ, \mathrm{id}_S)$$

be a repair algebra.

A *primitive basis* for $\mathcal{A}$ is a pair

$$\mathcal{P} = (\mathcal{P}_\mathcal{D}, \mathcal{P}_\mathcal{R}),$$

where

$$\mathcal{P}_\mathcal{D} \subseteq \mathcal{D} \qquad \text{and} \qquad \mathcal{P}_\mathcal{R} \subseteq \mathcal{R}.$$

The elements of $\mathcal{P}_\mathcal{D}$ are called *primitive damage operations*, and the elements of $\mathcal{P}_\mathcal{R}$ are called *primitive repair operations*.

The repair basis is generating when

$$\mathcal{R} = \langle \mathcal{P}_\mathcal{R} \rangle,$$

where

$$\langle \mathcal{P}_\mathcal{R} \rangle$$

denotes the smallest submonoid of $\mathrm{End}(S)$ containing $\mathcal{P}_\mathcal{R}$.

Thus, every repair transformation in a generating repair basis can be expressed as a finite composition of primitive repair operations. When no smaller generating family is distinguished, one may take

$$\mathcal{P}_{\mathcal{R}} = \mathcal{R}.$$

Primitive damage operations represent elementary perturbations of the runtime state. Depending on the level of abstraction, they may represent loss of structure, degradation of function, interruption of a process, corruption of stored information, or displacement from a homeostatic region.

Primitive repair operations represent elementary corrective transformations. They do not necessarily restore the system in a single step. Their role is to provide the basic transformations from which more complex recovery pathways may be composed.

The identity transformation

$$\mathrm{id}_S : S \to S$$

acts as the null repair operation. It represents the absence of an active change and provides the neutral element required by the repair monoid.

Definition 4.8 (Viability Test)**.** The *viability test* associated with V is the function

$$\chi_V : S \to \{0, 1\}$$

defined by

$$\chi_V(s) = \begin{cases} 1, & s \in V, \\ 0, & s \notin V. \end{cases}$$

The viability test determines whether a state remains within the region required for continued system operation. It does not determine whether the state preserves the same homeostatic organization as an earlier state. That question is determined separately by the observation map

$$\eta : S \to \Omega.$$

Consequently, every transformed state may be evaluated along two distinct semantic dimensions:

$$\chi_V(s) \qquad \text{and} \qquad \eta(s).$$

The first measures viability, while the second records the selected homeostatic signature.

Definition 4.9 (Primitive Damage–Repair Response)**.** Let $s \in V$, let

$$d \in \mathcal{P}_{\mathcal{D}},$$

and let

$$r \in \mathcal{P}_{\mathcal{R}}.$$

The corresponding *primitive damage–repair response* is the state

$$(r \circ d)(s) = r(d(s)).$$

A primitive repair operation is not classified independently of context. The same transformation may restore one damaged state, compensate for another, and fail on a third. Its semantic role must therefore be evaluated relative to an initial viable state and a particular damage operation.

Definition 4.10 (Exact Primitive Restoration)**.** Let $s \in V$, $d \in \mathcal{P}_{\mathcal{D}}$, and $r \in \mathcal{P}_{\mathcal{R}}$. The operation r is an *exact primitive restoration* for d at s when

$$r(d(s)) = s.$$

Exact primitive restoration reverses the selected damage operation at the level of the modeled runtime state.

Definition 4.11 (Semantic Primitive Restoration)**.** Let $s \in V$, $d \in \mathcal{P}_{\mathcal{D}}$, and $r \in \mathcal{P}_{\mathcal{R}}$. The operation r is a *semantic primitive restoration* for d at s when

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s).$$

Every exact primitive restoration is also a semantic primitive restoration.

Proposition 4.12. *If*

$$r(d(s)) = s,$$

then

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s).$$

Proof. Since $s \in V$ and $r(d(s)) = s$, it follows immediately that

$$r(d(s)) \in V.$$

Applying η to the equality $r(d(s)) = s$ gives

$$\eta(r(d(s))) = \eta(s).$$

□

The converse need not hold. A primitive repair may restore viability and the required homeostatic signature without reproducing the original internal state.

Definition 4.13 (Non-Exact Semantic Restoration)**.** Let $s \in V$, $d \in \mathcal{P}_{\mathcal{D}}$, and $r \in \mathcal{P}_{\mathcal{R}}$. The operation r is a *primitive compensation* for d at s when

$$r(d(s)) \in V,$$

$$\eta(r(d(s))) = \eta(s),$$

and

$$r(d(s)) \neq s.$$

Primitive compensation restores the selected semantic organization through a state different from the original one. It therefore models functional recovery without exact structural reversal.

Definition 4.14 (Viability-Only Response)**.** Let $s \in V$, $d \in \mathcal{P}_{\mathcal{D}}$, and $r \in \mathcal{P}_{\mathcal{R}}$. The operation r is a *viability-only response* for d at s when

$$r(d(s)) \in V$$

but

$$\eta(r(d(s))) \neq \eta(s).$$

A viability-only response prevents immediate runtime failure but does not restore the original homeostatic signature. Such a state may represent a stable but altered organization.

Definition 4.15 (Failed Primitive Response)**.** Let $s \in V$, $d \in \mathcal{P}_{\mathcal{D}}$, and $r \in \mathcal{P}_{\mathcal{R}}$. The operation r is a *failed primitive response* for d at s when

$$r(d(s)) \notin V.$$

The four response classes may therefore be summarized as follows:

Response class	$r(d(s)) \in V$	$\eta(r(d(s))) = \eta(s)$	$r(d(s)) = s$
Exact restoration	yes	yes	yes
Compensation	yes	yes	no
Viability-only response	yes	no	not required
Failed response	no	not required	no

Example 4.16 (Minimal Regenerative State Model)**.** Let

$$S = \{s_{\text{healthy}}, s_{\text{damaged}}, s_{\text{compensated}}, s_{\text{failed}}\}$$

and let

$$V = \{s_{\text{healthy}}, s_{\text{compensated}}\}.$$

Suppose that

$$\eta(s_{\text{healthy}}) = \eta(s_{\text{compensated}}) = \omega_{\text{functional}},$$

while

$$\eta(s_{\text{damaged}}) = \eta(s_{\text{failed}}) = \omega_{\text{nonfunctional}}.$$

Let d satisfy

$$d(s_{\text{healthy}}) = s_{\text{damaged}}.$$

An exact restorative operation r_{exact} satisfies

$$r_{\text{exact}}(s_{\text{damaged}}) = s_{\text{healthy}}.$$

A compensatory operation r_{comp} satisfies

$$r_{\text{comp}}(s_{\text{damaged}}) = s_{\text{compensated}}.$$

Both transformations recover the functional homeostatic signature, but only r_{exact} restores the original state.

Primitive operations provide the local vocabulary of repair. Biological recovery, however, frequently requires several such operations to occur in a specific order. The next subsection therefore defines the composition rules governing multi-stage repair processes.

4.3 Composition Rules

Repair processes generally consist of ordered sequences of primitive operations rather than isolated transformations. The algebraic structure of repair must therefore distinguish an individual repair operator from a *repair pathway*: a finite program of operators executed in a specified order.

Definition 4.17 (Repair Word)**.** Let

$$\mathcal{P}_{\mathcal{R}} \subseteq \mathcal{R}$$

be a primitive repair basis.

A *repair word* is a finite sequence

$$w = (r_1, r_2, \ldots, r_n), \qquad r_i \in \mathcal{P}_{\mathcal{R}}.$$

The set of all repair words is denoted by

$$\mathcal{P}_{\mathcal{R}}^{*}.$$

The empty repair word is denoted by

$$\varepsilon.$$

The length of a repair word

$$w = (r_1, \ldots, r_n)$$

is

$$|w| = n,$$

and the empty repair word satisfies

$$|\varepsilon| = 0.$$

Definition 4.18 (Evaluation of a Repair Word)**.** The evaluation map

$$[\![\cdot]\!] : \mathcal{P}_{\mathcal{R}}^{*} \to \mathcal{R}$$

is defined by

$$[\![\varepsilon]\!] = \mathrm{id}_S$$

and

$$[\![(r_1, \ldots, r_n)]\!] = r_n \circ r_{n-1} \circ \cdots \circ r_1.$$

Thus, r_1 acts first and r_n acts last.

Because

$$\mathcal{P}_{\mathcal{R}} \subseteq \mathcal{R},$$

and because $\mathcal{R}$ is closed under composition, every finite repair word evaluates to an element of $\mathcal{R}$.

Proposition 4.19 (Closure of Repair-Word Evaluation)**.** *For every*

$$w \in \mathcal{P}_{\mathcal{R}}^{*},$$

we have

$$[\![w]\!] \in \mathcal{R}.$$

Proof. The proof proceeds by induction on the length of w.

For the empty word,

$$[\![\varepsilon]\!] = \mathrm{id}_S \in \mathcal{R}.$$

Suppose

$$[\![(r_1, \ldots, r_n)]\!] \in \mathcal{R}.$$

For any

$$r_{n+1} \in \mathcal{P}_{\mathcal{R}} \subseteq \mathcal{R},$$

closure of $\mathcal{R}$ under composition gives

$$r_{n+1} \circ [\![(r_1, \ldots, r_n)]\!] \in \mathcal{R}.$$

This transformation is precisely

$$[\![(r_1, \ldots, r_n, r_{n+1})]\!].$$

□

Definition 4.20 (Concatenation)**.** Let

$$u = (r_1, \ldots, r_m)$$

and

$$v = (q_1, \ldots, q_n)$$

be repair words. Their concatenation is

$$u \cdot v = (r_1, \ldots, r_m, q_1, \ldots, q_n).$$

Operationally, the pathway u is executed first, followed by the pathway v.

The evaluation of a concatenated repair pathway satisfies

$$[\![u \cdot v]\!] = [\![v]\!] \circ [\![u]\!].$$

The reversal in the displayed order reflects the standard convention for function composition: the transformation written on the right acts first.

Proposition 4.21 (Associativity of Repair-Pathway Composition)**.** *For all*

$$u, v, w \in \mathcal{P}_{\mathcal{R}}^*,$$

we have

$$(u \cdot v) \cdot w = u \cdot (v \cdot w).$$

Moreover,

$$\varepsilon \cdot w = w = w \cdot \varepsilon.$$

Proof. These identities follow directly from associativity of finite-sequence concatenation and the defining role of the empty sequence. □

The set

$$(\mathcal{P}_{\mathcal{R}}^*, \cdot, \varepsilon)$$

is therefore the free monoid generated by the primitive repair operations.

Definition 4.22 (Repair Trace)**.** Let

$$s \in S, \qquad d \in \mathcal{D},$$

and let

$$w = (r_1, \ldots, r_n) \in \mathcal{P}_{\mathcal{R}}^*.$$

The *repair trace* generated by s, d, and w is the finite sequence

$$x_0, x_1, \ldots, x_n$$

defined by

$$x_0 = d(s)$$

and

$$x_k = r_k(x_{k-1}), \qquad 1 \leq k \leq n.$$

The terminal state of the trace is

$$x_n = [\![w]\!](d(s)).$$

A repair trace records the intermediate states through which the system passes. This distinction is important because two repair words may produce the same terminal state while traversing different intermediate configurations.

Definition 4.23 (Admissibility Region)**.** An *admissibility region* is a subset

$$U \subseteq S$$

containing the viability region:

$$V \subseteq U.$$

The set U represents states that may be traversed during a repair process without violating the selected runtime constraints.

The viability region V characterizes completed states capable of continued operation. The potentially larger region U allows a model to represent temporary repair states that are not yet homeostatic but remain admissible during recovery.

Definition 4.24 (Admissible Repair Pathway)**.** Let

$$x_0, x_1, \ldots, x_n$$

be the repair trace generated by s, d, and w.

The repair word w is *U-admissible for d at s* when

$$x_k \in U$$

for every

$$0 \leq k \leq n.$$

Admissibility is a property of the complete pathway, not merely its terminal state. A pathway may restore the required homeostatic signature while passing through an impermissible intermediate state. Such a pathway is terminally successful but not admissible relative to U.

Definition 4.25 (Successful Repair Pathway)**.** Let

$$s \in V, \qquad d \in \mathcal{D},$$

and

$$w \in \mathcal{P}_{\mathcal{R}}^{*}.$$

The word w is a *successful repair pathway for d at s* when

$$[\![w]\!](d(s)) \in V$$

and

$$\eta\big([\![w]\!](d(s))\big) = \eta(s).$$

Equivalently,

$$[\![w]\!](d(s)) \in V \cap [s]_{\eta}.$$

Definition 4.26 (Strongly Successful Repair Pathway)**.** A successful repair pathway w is *strongly successful relative to U* when it is also U-admissible.

Strong success therefore combines two requirements:

1. the terminal state restores viability and the original homeostatic signature; and
2. every intermediate state remains within the designated admissibility region.

Definition 4.27 (Successful Pathway Set)**.** For

$$s \in V$$

and

$$d \in \mathcal{D},$$

define

$$\mathrm{Path}_{\mathcal{A}}(s,d) = \{w \in \mathcal{P}_{\mathcal{R}}^{*} \mid [\![w]\!](d(s)) \in V \cap [s]_{\eta}\}.$$

The pair (s,d) is recoverable precisely when

$$\mathrm{Path}_{\mathcal{A}}(s,d) \neq \varnothing,$$

provided that the primitive basis generates the available repair monoid.

Definition 4.28 (Repair Depth)**.** Let

$$s \in V$$

and

$$d \in \mathcal{D}.$$

If

$$\mathrm{Path}_{\mathcal{A}}(s,d) \neq \varnothing,$$

the *repair depth* of d at s is

$$\delta_{\mathcal{A}}(s,d) = \min\{|w| \mid w \in \mathrm{Path}_{\mathcal{A}}(s,d)\}.$$

If no successful repair pathway exists, define

$$\delta_{\mathcal{A}}(s,d) = \infty.$$

Repair depth measures the minimum number of primitive repair stages required to restore the selected homeostatic organization. It is an algebraic measure of recovery complexity rather than elapsed physical time.

Different primitive bases may assign different repair depths to the same damage event. Repair depth is therefore relative both to the repair algebra and to the chosen decomposition of repair into primitive operations.

Definition 4.29 (Invariant-Preserving Repair Transformation)**.** Let

$$W \subseteq V.$$

A repair transformation

$$r \in \mathcal{R}$$

is *invariant-preserving on* W when, for every $x \in W$,

$$r(x) \in W$$

and

$$\eta(r(x)) = \eta(x).$$

Let

$$\mathrm{Pres}_\eta(W) = \{r \in \mathcal{R} \mid r(W) \subseteq W \text{ and } \eta(r(x)) = \eta(x) \text{ for all } x \in W\}.$$

Proposition 4.30 (Closure of Invariant-Preserving Transformations)**.** *For every*

$$W \subseteq V,$$

the structure

$$(\mathrm{Pres}_\eta(W), \circ, \mathrm{id}_S)$$

is a submonoid of

$$(\mathcal{R}, \circ, \mathrm{id}_S).$$

Proof. The identity transformation satisfies

$$\mathrm{id}_S(W) = W$$

and

$$\eta(\mathrm{id}_S(x)) = \eta(x)$$

for every $x \in W$. Hence

$$\mathrm{id}_S \in \mathrm{Pres}_\eta(W).$$

Now let

$$r_1, r_2 \in \mathrm{Pres}_\eta(W).$$

For every $x \in W$, we have

$$r_1(x) \in W$$

and therefore

$$r_2(r_1(x)) \in W.$$

Furthermore,

$$\eta(r_2(r_1(x))) = \eta(r_1(x)) = \eta(x).$$

Thus,

$$r_2 \circ r_1 \in \mathrm{Pres}_\eta(W).$$

Associativity is inherited from function composition. □

This result formalizes the compositional stability of homeostatic maintenance: a finite sequence of transformations that each preserves a homeostatic region also preserves that region as a whole.

Proposition 4.31 (Stable Extension of a Successful Pathway)**.** *Let*

$$w \in \mathrm{Path}_{\mathcal{A}}(s, d),$$

and define

$$W = V \cap [s]_{\eta}.$$

If

$$v \in \mathcal{P}^{*}_{\mathcal{R}}$$

satisfies

$$[\![v]\!] \in \mathrm{Pres}_{\eta}(W),$$

then

$$w \cdot v \in \mathrm{Path}_{\mathcal{A}}(s, d).$$

Proof. Because w is successful,

$$x = [\![w]\!](d(s)) \in W.$$

Since

$$[\![v]\!] \in \mathrm{Pres}_{\eta}(W),$$

we have

$$[\![v]\!](x) \in W.$$

Using the concatenation rule,

$$[\![w \cdot v]\!](d(s)) = [\![v]\!]\left([\![w]\!](d(s))\right) \in W.$$

Therefore,

$$w \cdot v \in \mathrm{Path}_{\mathcal{A}}(s, d).$$

□

The order of primitive repair operations may affect the resulting state.

Definition 4.32 (Commuting Repair Operations)**.** Let

$$r, q \in \mathcal{R}$$

and let

$$U \subseteq S.$$

The transformations r and q *commute on* U when

$$r(q(x)) = q(r(x))$$

for every

$$x \in U.$$

When two repair operations commute on the relevant region, their execution order does not affect the resulting state. In general, however,

$$r \circ q \neq q \circ r.$$

Repair composition is therefore generally noncommutative. This captures the fact that biological repair pathways may depend critically on sequencing: activation, stabilization, replacement, and recovery operations may succeed in one order and fail in another.

The composition rules above distinguish the syntax of repair pathways, their net transformations, their intermediate traces, and their terminal semantic outcomes. The next subsection develops morphisms between repair algebras, allowing repair structure to be compared across different state spaces and levels of biological organization.

4.4 Repair Morphisms

Repair algebras may describe the same regenerative process at different levels of abstraction. One algebra may represent molecular configurations, another may represent cellular states, and a third may represent tissue-level organization. To compare such models, we require mappings that preserve their repair structure.

Let

$$\mathcal{A} = (S_A, V_A, \Omega_A, \eta_A, \mathcal{D}_A, \mathcal{R}_A, \circ, \mathrm{id}_{S_A})$$

and

$$\mathcal{B} = (S_B, V_B, \Omega_B, \eta_B, \mathcal{D}_B, \mathcal{R}_B, \circ, \mathrm{id}_{S_B})$$

be repair algebras.

Definition 4.33 (Repair-Algebra Morphism)**.** A *repair-algebra morphism*

$$\mathfrak{F} : \mathcal{A} \longrightarrow \mathcal{B}$$

is a tuple

$$\mathfrak{F} = (f, g, \Phi_{\mathcal{D}}, \Phi_{\mathcal{R}}),$$

where

$$f : S_A \to S_B, \qquad g : \Omega_A \to \Omega_B,$$
$$\Phi_{\mathcal{D}} : \mathcal{D}_A \to \mathcal{D}_B,$$

and

$$\Phi_{\mathcal{R}} : \mathcal{R}_A \to \mathcal{R}_B,$$

such that the following conditions hold.

1. *Viability preservation:*
$$f(V_A) \subseteq V_B.$$

2. *Semantic compatibility:*
$$\eta_B \circ f = g \circ \eta_A.$$

   Equivalently, for every $s \in S_A$,
$$\eta_B(f(s)) = g(\eta_A(s)).$$

3. *Repair-monoid preservation:*

$$\Phi_{\mathcal{R}}\left(\mathrm{id}_{S_A}\right) = \mathrm{id}_{S_B},$$

and, for all

$$r_1, r_2 \in \mathcal{R}_A,$$
$$\Phi_{\mathcal{R}}(r_2 \circ r_1) = \Phi_{\mathcal{R}}(r_2) \circ \Phi_{\mathcal{R}}(r_1).$$

4. *Repair-action compatibility:* for every

$$r \in \mathcal{R}_A$$

and

$$s \in S_A,$$
$$f(r(s)) = \Phi_{\mathcal{R}}(r)(f(s)).$$

5. *Damage-action compatibility:* for every

$$d \in \mathcal{D}_A$$

and

$$s \in S_A,$$
$$f(d(s)) = \Phi_{\mathcal{D}}(d)(f(s)).$$

The semantic compatibility condition is expressed by the commutative diagram

$$\begin{array}{ccc} S_A & \xrightarrow{f} & S_B \\ \eta_A \downarrow & & \downarrow \eta_B \\ \Omega_A & \xrightarrow{g} & \Omega_B. \end{array}$$

Similarly, repair-action compatibility requires

$$\begin{array}{ccc} S_A & \xrightarrow{r} & S_A \\ f \downarrow & & \downarrow f \\ S_B & \xrightarrow{\Phi_{\mathcal{R}}(r)} & S_B \end{array}$$

to commute for every $r \in \mathcal{R}_A$.

Damage-action compatibility imposes the corresponding condition

$$\begin{array}{ccc} S_A & \xrightarrow{d} & S_A \\ f \downarrow & & \downarrow f \\ S_B & \xrightarrow{\Phi_{\mathcal{D}}(d)} & S_B \end{array}$$

for every $d \in \mathcal{D}_A$.

These conditions ensure that a state transformation may be performed before or after translation between the two repair algebras with the same resulting abstract state.

Proposition 4.34 (Preservation of Semantic Equivalence)**.** *Let*

$$\mathfrak{F} : \mathcal{A} \to \mathcal{B}$$

be a repair-algebra morphism. If

$$s \sim_{\eta_A} t,$$

then

$$f(s) \sim_{\eta_B} f(t).$$

Proof. The assumption

$$s \sim_{\eta_A} t$$

means

$$\eta_A(s) = \eta_A(t).$$

Applying g gives

$$g(\eta_A(s)) = g(\eta_A(t)).$$

By semantic compatibility,

$$\eta_B(f(s)) = g(\eta_A(s))$$

and

$$\eta_B(f(t)) = g(\eta_A(t)).$$

Therefore,

$$\eta_B(f(s)) = \eta_B(f(t)),$$

so

$$f(s) \sim_{\eta_B} f(t).$$

□

Theorem 4.35 (Preservation of Successful Repair)**.** *Let*

$$\mathfrak{F} : \mathcal{A} \to \mathcal{B}$$

be a repair-algebra morphism.

Suppose

$$s \in V_A, \qquad d \in \mathcal{D}_A, \qquad r \in \mathcal{R}_A,$$

and suppose that r is a successful repair of d at s. Then

$$\Phi_{\mathcal{R}}(r)$$

is a successful repair of

$$\Phi_{\mathcal{D}}(d)$$

at

$$f(s).$$

Proof. Because r is a successful repair of d at s,

$$r(d(s)) \in V_A$$

and

$$\eta_A(r(d(s))) = \eta_A(s).$$

Viability preservation gives

$$f(r(d(s))) \in V_B.$$

Using damage-action compatibility and repair-action compatibility,

$$f(r(d(s))) = \Phi_{\mathcal{R}}(r)\left(\Phi_{\mathcal{D}}(d)(f(s))\right).$$

Therefore,

$$\Phi_{\mathcal{R}}(r)\left(\Phi_{\mathcal{D}}(d)(f(s))\right) \in V_B.$$

For the homeostatic signature, semantic compatibility gives

$$\eta_B(f(r(d(s)))) = g(\eta_A(r(d(s)))).$$

Since

$$\eta_A(r(d(s))) = \eta_A(s),$$

we obtain

$$g(\eta_A(r(d(s)))) = g(\eta_A(s)).$$

Applying semantic compatibility again,

$$g(\eta_A(s)) = \eta_B(f(s)).$$

Hence

$$\eta_B\left(\Phi_{\mathcal{R}}(r)\left(\Phi_{\mathcal{D}}(d)(f(s))\right)\right) = \eta_B(f(s)).$$

Thus,

$$\Phi_{\mathcal{R}}(r)$$

is a successful repair of

$$\Phi_{\mathcal{D}}(d)$$

at

$$f(s).$$

□

This theorem establishes that repair correctness is preserved when a system is translated through a repair-algebra morphism.

Corollary 4.36 (Preservation of Recoverability)**.** *If*

$$(s, d)$$

is recoverable in $\mathcal{A}$*, then*

$$(f(s), \Phi_{\mathcal{D}}(d))$$

is recoverable in $\mathcal{B}$*.*

Proof. Recoverability in $\mathcal{A}$ implies that there exists

$$r \in \mathcal{R}_A$$

that successfully repairs d at s. By the preservation theorem,

$$\Phi_{\mathcal{R}}(r)$$

successfully repairs

$$\Phi_{\mathcal{D}}(d)$$

at

$$f(s).$$

□

Corollary 4.37 (Preservation of Exact Repair)**.** *Suppose*

$$r(d(s)) = s.$$

Then

$$\Phi_{\mathcal{R}}(r)\,(\Phi_{\mathcal{D}}(d)(f(s))) = f(s).$$

Proof. Using repair-action and damage-action compatibility,

$$\Phi_{\mathcal{R}}(r)\,(\Phi_{\mathcal{D}}(d)(f(s))) = f(r(d(s))).$$

Since

$$r(d(s)) = s,$$

we obtain

$$f(r(d(s))) = f(s).$$

□

Thus exact restoration remains exact after translation through a repair morphism, although distinct source states may become identified when the state map f is not injective.

Definition 4.38 (Signature-Faithful Morphism)**.** A repair-algebra morphism

$$\mathfrak{F} = (f, g, \Phi_{\mathcal{D}}, \Phi_{\mathcal{R}})$$

is *signature-faithful* when

$$g : \Omega_A \to \Omega_B$$

is injective.

A signature-faithful morphism does not collapse distinct homeostatic signatures.

Proposition 4.39 (Reflection of Semantic Equivalence)**.** *Let*

$$\mathfrak{F} : \mathcal{A} \to \mathcal{B}$$

be signature-faithful. Then

$$f(s) \sim_{\eta_B} f(t)$$

implies

$$s \sim_{\eta_A} t.$$

Proof. The assumption

$$f(s) \sim_{\eta_B} f(t)$$

means

$$\eta_B(f(s)) = \eta_B(f(t)).$$

By semantic compatibility,

$$g(\eta_A(s)) = g(\eta_A(t)).$$

Since g is injective,

$$\eta_A(s) = \eta_A(t).$$

Therefore,

$$s \sim_{\eta_A} t.$$

□

Definition 4.40 (Viability-Reflecting Morphism)**.** A repair-algebra morphism is *viability-reflecting* when

$$f(s) \in V_B \quad \Longrightarrow \quad s \in V_A$$

for every

$$s \in S_A.$$

A morphism that is both viability-preserving and viability-reflecting satisfies

$$s \in V_A \quad \Longleftrightarrow \quad f(s) \in V_B.$$

Definition 4.41 (Repair-Algebra Embedding)**.** A repair-algebra morphism is a *repair-algebra embedding* when

$$f, \qquad g, \qquad \Phi_{\mathcal{D}}, \qquad \Phi_{\mathcal{R}}$$

are injective and the morphism is viability-reflecting.

A repair-algebra embedding represents one repair system as a structurally faithful subsystem of another.

By contrast, a noninjective state map may intentionally collapse multiple microscopic configurations into a single higher-level state. Such morphisms provide a formal mechanism for semantic abstraction and for relating biological models across scales.

Definition 4.42 (Primitive-Basis-Preserving Morphism)**.** Suppose that $\mathcal{A}$ and $\mathcal{B}$ are equipped with primitive bases

$$\mathcal{P}_{\mathcal{R}}^{A}$$

and

$$\mathcal{P}_{\mathcal{R}}^{B}.$$

A repair-algebra morphism is *primitive-basis-preserving* when

$$\Phi_{\mathcal{R}}\left(\mathcal{P}_{\mathcal{R}}^{A}\right) \subseteq \mathcal{P}_{\mathcal{R}}^{B}.$$

For a repair word

$$w = (r_1, \ldots, r_n) \in \left(\mathcal{P}_{\mathcal{R}}^{A}\right)^*,$$

define its image word by

$$\Phi_{\mathcal{R}}^{*}(w) = \left(\Phi_{\mathcal{R}}(r_1), \ldots, \Phi_{\mathcal{R}}(r_n)\right).$$

For the empty word,

$$\Phi_{\mathcal{R}}^{*}(\varepsilon) = \varepsilon.$$

Proposition 4.43 (Preservation of Repair-Word Evaluation)**.** *For every repair word*

$$w \in \left(\mathcal{P}_{\mathcal{R}}^{A}\right)^*$$

and every

$$s \in S_A,$$

we have

$$f\left(\llbracket w \rrbracket_A(s)\right) = \llbracket \Phi_{\mathcal{R}}^{*}(w) \rrbracket_B(f(s)).$$

Proof. The proof proceeds by induction on the length of w.

For the empty word,

$$f\left([\![\varepsilon]\!]_A(s)\right) = f(s),$$

while

$$[\![\Phi^*_{\mathcal{R}}(\varepsilon)]\!]_B(f(s)) = \mathrm{id}_{S_B}(f(s)) = f(s).$$

Suppose the result holds for

$$w = (r_1, \ldots, r_n).$$

Let

$$w' = (r_1, \ldots, r_n, r_{n+1}).$$

Then

$$[\![w']\!]_A = r_{n+1} \circ [\![w]\!]_A.$$

Therefore,

$$f\left([\![w']\!]_A(s)\right) = f\left(r_{n+1}\left([\![w]\!]_A(s)\right)\right).$$

By repair-action compatibility,

$$f\left(r_{n+1}\left([\![w]\!]_A(s)\right)\right) = \Phi_{\mathcal{R}}(r_{n+1})\left(f\left([\![w]\!]_A(s)\right)\right).$$

Using the induction hypothesis,

$$f\left([\![w]\!]_A(s)\right) = [\![\Phi^*_{\mathcal{R}}(w)]\!]_B(f(s)).$$

Hence

$$f\left([\![w']\!]_A(s)\right) = [\![\Phi^*_{\mathcal{R}}(w')]\!]_B(f(s)).$$

□

Corollary 4.44 (Preservation of Repair Pathways)**.** *If*

$$w \in \mathrm{Path}_{\mathcal{A}}(s, d),$$

then

$$\Phi^*_{\mathcal{R}}(w) \in \mathrm{Path}_{\mathcal{B}}\left(f(s), \Phi_{\mathcal{D}}(d)\right).$$

Corollary 4.45 (Repair-Depth Inequality)**.** *Assume the morphism is primitive-basis-preserving. Then*

$$\delta_{\mathcal{B}}\left(f(s), \Phi_{\mathcal{D}}(d)\right) \leq \delta_{\mathcal{A}}(s, d).$$

Proof. If

$$\delta_{\mathcal{A}}(s, d) = \infty,$$

then the inequality is immediate in

$$\mathbb{N} \cup \{\infty\}.$$

Suppose instead that

$$\delta_{\mathcal{A}}(s, d) < \infty.$$

Choose a shortest successful pathway

$$w \in \mathrm{Path}_{\mathcal{A}}(s, d)$$

with

$$|w| = \delta_{\mathcal{A}}(s, d).$$

Because the morphism is primitive-basis-preserving,

$$\Phi^*_{\mathcal{R}}(w)$$

has the same length as w. By preservation of repair pathways,

$$\Phi^*_{\mathcal{R}}(w) \in \mathrm{Path}_{\mathcal{B}}\left(f(s), \Phi_{\mathcal{D}}(d)\right).$$

Therefore,

$$\delta_{\mathcal{B}}\left(f(s), \Phi_{\mathcal{D}}(d)\right) \leq |\Phi^*_{\mathcal{R}}(w)| = |w| = \delta_{\mathcal{A}}(s, d).$$

□

Repair-algebra morphisms compose naturally.

Suppose

$$\mathfrak{F} = (f, g, \Phi_{\mathcal{D}}, \Phi_{\mathcal{R}}) : \mathcal{A} \to \mathcal{B}$$

and

$$\mathfrak{G} = \left(f', g', \Psi_{\mathcal{D}}, \Psi_{\mathcal{R}}\right) : \mathcal{B} \to \mathcal{C}$$

are repair-algebra morphisms.

Define

$$\mathfrak{G} \circ \mathfrak{F} = \left(f' \circ f, g' \circ g, \Psi_{\mathcal{D}} \circ \Phi_{\mathcal{D}}, \Psi_{\mathcal{R}} \circ \Phi_{\mathcal{R}}\right).$$

Proposition 4.46 (Composition of Repair Morphisms)**.** *The composite*

$$\mathfrak{G} \circ \mathfrak{F} : \mathcal{A} \to \mathcal{C}$$

is a repair-algebra morphism.

Proof. Viability preservation follows from

$$f(V_A) \subseteq V_B$$

and

$$f'(V_B) \subseteq V_C.$$

Semantic compatibility follows from

$$\eta_C \circ f' = g' \circ \eta_B$$

and

$$\eta_B \circ f = g \circ \eta_A.$$

Thus,

$$\eta_C \circ f' \circ f = g' \circ g \circ \eta_A.$$

Preservation of identities and composition follows because the composite of monoid homomorphisms is a monoid homomorphism. Repair-action and damage-action compatibility follow by applying the corresponding compatibility equations for $\mathfrak{F}$ and $\mathfrak{G}$ successively. □

For every repair algebra $\mathcal{A}$, the tuple

$$\mathrm{Id}_{\mathcal{A}} = (\mathrm{id}_{S_A}, \mathrm{id}_{\Omega_A}, \mathrm{id}_{\mathcal{D}_A}, \mathrm{id}_{\mathcal{R}_A})$$

is an identity repair-algebra morphism.

Theorem 4.47 (Category of Repair Algebras)**.** *Repair algebras and repair-algebra morphisms form a category, denoted*

$$\mathbf{RepAlg}.$$

Proof. Morphisms are closed under composition by the preceding proposition. Composition is associative because each component is composed using ordinary function composition. The morphisms

$$\mathrm{Id}_{\mathcal{A}}$$

act as left and right identities for every repair-algebra morphism. □

The category

$$\mathbf{RepAlg}$$

provides a formal setting in which regenerative systems can be translated, embedded, abstracted, and compared. It also permits repair semantics to be related across molecular, cellular, tissue, organismal, and computational levels without requiring those levels to share the same internal state space.

The next subsection develops the algebraic properties of repair operations within an individual repair algebra.

4.5 Algebraic Properties

Repair operations may satisfy additional algebraic laws beyond the monoid axioms required in the definition of a repair algebra. These laws can strongly constrain the structure of recovery pathways.

Of particular importance are repair operations whose effect stabilizes after a single application and whose order of execution is irrelevant on a designated region of the state space. Such operations model corrective transformations that establish a condition rather than repeatedly accumulating an effect.

Definition 4.48 (Idempotent Repair Transformation)**.** Let

$$r \in \mathcal{R}$$

and let

$$U \subseteq S.$$

The transformation r is *idempotent on* U when

$$r(r(x)) = r(x)$$

for every

$$x \in U$$

for which

$$r(x) \in U.$$

When $r(U) \subseteq U$, idempotence on U may equivalently be written

$$r \circ r = r$$

after restriction to U.

An idempotent repair operation represents a transformation whose corrective effect is already complete after its first successful application. Repeating the same operation does not produce a further state change within the region on which the law holds.

Definition 4.49 (Stable Repair Region)**.** Let

$$\mathcal{Q} \subseteq \mathcal{R}.$$

A subset

$$U \subseteq S$$

is *stable under* $\mathcal{Q}$ when

$$r(U) \subseteq U$$

for every

$$r \in \mathcal{Q}.$$

Stability ensures that algebraic identities imposed on U remain applicable throughout every pathway generated by operations in $\mathcal{Q}$.

Definition 4.50 (Commuting Idempotent Repair Family)**.** A finite family

$$\mathcal{Q} = \{r_1, \ldots, r_n\} \subseteq \mathcal{R}$$

is a *commuting idempotent repair family on* U when:

1. U is stable under $\mathcal{Q}$;
2. every r_i is idempotent on U;
3. for every i, j,
$$r_i \circ r_j = r_j \circ r_i$$
on U.

The three conditions imply that repetitions and reorderings of the primitive repair operations do not affect the resulting transformation on U.

Let

$$w = (q_1, \ldots, q_m)$$

be a repair word whose letters lie in $\mathcal{Q}$. Define the *support* of w by

$$\operatorname{supp}(w) = \{r_i \in \mathcal{Q} \mid r_i \text{ occurs in } w\}.$$

Thus, the support records which repair operations occur while discarding both their multiplicities and their order.

Theorem 4.51 (Canonicalization of Commuting Idempotent Repair Pathways)**.** *Let*

$$\mathcal{Q} = \{r_1, \dots, r_n\}$$

be a commuting idempotent repair family on U*.*

For every repair word

$$w \in \mathcal{Q}^*,$$

there exists a word

$$\mathrm{can}(w)$$

having the following properties:

$$|\,\mathrm{can}(w)| = |\,\mathrm{supp}(w)|,$$

each element of

$$\mathrm{supp}(w)$$

appears exactly once in $\mathrm{can}(w)$*, and*

$$[\![w]\!](x) = [\![\mathrm{can}(w)]\!](x)$$

for every

$$x \in U.$$

In particular,

$$|\,\mathrm{can}(w)| \le n.$$

Proof. Fix the ordering

$$r_1 < r_2 < \cdots < r_n.$$

Starting with w, use pairwise commutativity to reorder the letters so that all occurrences of r_1 appear first, followed by all occurrences of r_2, and so on.

Because U is stable under every element of $\mathcal{Q}$, all intermediate states remain in U. Hence the assumed commutativity and idempotence identities remain applicable throughout the evaluation.

For each r_i, repeated occurrences satisfy

$$r_i^k(x) = r_i(x)$$

for every integer $k \ge 1$ and every relevant $x \in U$, by idempotence.

Therefore all repeated copies of r_i may be deleted without changing the evaluated transformation on U.

The resulting word contains exactly one copy of every generator occurring in w, arranged according to the fixed ordering. Define this word to be

$$\mathrm{can}(w).$$

It follows that

$$[\![w]\!](x) = [\![\mathrm{can}(w)]\!](x)$$

for all $x \in U$.

Since $\mathrm{can}(w)$ contains at most one copy of each of the n generators,

$$|\,\mathrm{can}(w)| \le n.$$

□

This theorem shows that a potentially unbounded syntactic space of repair words may collapse to a finite family of effective repair transformations.

Although

$$\mathcal{Q}^*$$

contains infinitely many words whenever

$$\mathcal{Q} \neq \varnothing,$$

the number of distinct transformations induced on U is finite.

Corollary 4.52 (Finite Effective Repair Monoid)**.** *Let*

$$\mathcal{Q} = \{r_1, \ldots, r_n\}$$

be a commuting idempotent repair family on U.

Then the submonoid generated by $\mathcal{Q}$, after restriction to U, contains at most

$$2^n$$

distinct transformations.

Proof. By the canonicalization theorem, every repair word is equivalent on U to a canonical word containing each generator at most once.

Such a canonical word is completely determined by the subset

$$J \subseteq \{1, \ldots, n\}$$

of generators that occur.

There are exactly

$$2^n$$

such subsets.

Different subsets may still induce the same transformation, so the number of distinct induced transformations is at most

$$2^n.$$

□

The bound need not be attained. Algebraic dependencies among primitive repair operations may cause two distinct subsets of generators to induce the same transformation. Thus the effective repair monoid may be substantially smaller than the Boolean family of all subsets.

Definition 4.53 (Effective Repair Equivalence)**.** For repair words

$$u, v \in \mathcal{Q}^*,$$

define

$$u \equiv_U v$$

when

$$[\![u]\!](x) = [\![v]\!](x)$$

for every

$$x \in U.$$

The quotient

$$\mathcal{Q}^*/\equiv_U$$

therefore identifies syntactically different repair pathways that induce the same state transformation on the relevant region.

Under the commuting-idempotent hypotheses, this quotient is finite whenever $\mathcal{Q}$ is finite.

Proposition 4.54 (Support Determines Effective Repair)**.** *Let $\mathcal{Q}$ be a commuting idempotent repair family on U.*

If

$$\mathrm{supp}(u) = \mathrm{supp}(v),$$

then

$$u \equiv_U v.$$

Proof. Both u and v canonicalize to the same ordered list of distinct generators. Hence

$$\mathrm{can}(u) = \mathrm{can}(v).$$

By the canonicalization theorem,

$$[\![u]\!](x) = [\![\mathrm{can}(u)]\!](x) = [\![\mathrm{can}(v)]\!](x) = [\![v]\!](x)$$

for every

$$x \in U.$$

□

The converse need not hold. Two different supports may still induce the same transformation because one repair operation may be algebraically redundant in the presence of others.

The canonicalization result has an immediate consequence for recovery complexity.

Theorem 4.55 (Bounded Repair Depth for Finite Commuting Idempotent Systems)**.** *Let*

$$\mathcal{P}_{\mathcal{R}} = \{r_1, \ldots, r_n\}$$

be a finite primitive repair basis that forms a commuting idempotent repair family on U.

Let

$$s \in V, \qquad d \in \mathcal{D},$$

and suppose

$$d(s) \in U.$$

If

$$\mathrm{Path}_{\mathcal{A}}(s, d) \neq \varnothing,$$

and the successful pathway remains within U, then

$$\delta_{\mathcal{A}}(s, d) \leq n.$$

Proof. Let

$$w \in \mathrm{Path}_{\mathcal{A}}(s, d)$$

be a successful repair pathway whose trace remains in U.

By the canonicalization theorem, there exists

$$\mathrm{can}(w)$$

such that

$$[\![w]\!](d(s)) = [\![\mathrm{can}(w)]\!](d(s))$$

and

$$|\,\mathrm{can}(w)| \leq n.$$

Since w is successful,

$$[\![w]\!](d(s)) \in V \cap [s]_\eta.$$

Therefore,

$$[\![\mathrm{can}(w)]\!](d(s)) \in V \cap [s]_\eta,$$

so

$$\mathrm{can}(w) \in \mathrm{Path}_{\mathcal{A}}(s, d).$$

By the definition of repair depth,

$$\delta_{\mathcal{A}}(s, d) \leq |\,\mathrm{can}(w)| \leq n.$$

□

This theorem separates two different sources of repair complexity.

In a general repair algebra, pathways may be arbitrarily long because primitive operations may need to be repeated and because their execution order may matter.

In a finite commuting idempotent repair system, neither phenomenon contributes to minimal repair depth. Every successful pathway has an equivalent representation using each relevant primitive operation at most once.

Consequently, the search for a successful repair can be reduced from an infinite word space to a finite subset-selection problem.

For

$$\mathcal{P}_{\mathcal{R}} = \{r_1, \ldots, r_n\},$$

one need only consider subsets

$$J \subseteq \{1, \ldots, n\}.$$

Each subset determines the canonical repair transformation

$$R_J = \prod_{j \in J} r_j,$$

where the product may be evaluated in any fixed order because the generators commute on U.

Thus recoverability reduces to the existence of a subset J such that

$$R_J(d(s)) \in V \cap [s]_\eta.$$

This finite formulation will later permit repair depth to be studied not only as an algebraic invariant, but also as a computational optimization problem.

4.5.1 Semantic Repair Semilattices

The finite canonicalization theory above admits a stronger structural interpretation. When primitive repair operations commute and are idempotent, composition no longer behaves as an arbitrary monoid operation. Instead, after quotienting by effective equivalence, composition becomes commutative and idempotent.

This places the resulting repair structure within the theory of semilattices.

Let

$$\mathcal{Q} = \{r_1, \ldots, r_n\} \subseteq \mathcal{R}$$

be a finite commuting idempotent repair family on a stable region

$$U \subseteq S.$$

Recall the effective repair equivalence relation

$$u \equiv_U v$$

when

$$[\![u]\!](x) = [\![v]\!](x)$$

for every

$$x \in U.$$

Denote the equivalence class of a repair word w by

$$[w]_U.$$

Proposition 4.56 (Congruence of Effective Repair Equivalence)**.** *The relation*

$$\equiv_U$$

is a monoid congruence on

$$\mathcal{Q}^*.$$

That is, if

$$u \equiv_U u'$$

and

$$v \equiv_U v',$$

then

$$u \cdot v \equiv_U u' \cdot v'.$$

Proof. Let

$$x \in U.$$

Because U is stable under every generator in $\mathcal{Q}$, evaluation of any repair word over $\mathcal{Q}$ maps U into U.

Using the concatenation rule,

$$[\![u \cdot v]\!](x) = [\![v]\!]\left([\![u]\!](x)\right).$$

Since

$$u \equiv_U u',$$

we have

$$[\![u]\!](x) = [\![u']\!](x).$$

Let this common state be y. Stability gives

$$y \in U.$$

Since

$$v \equiv_U v',$$

we therefore obtain

$$[\![v]\!](y) = [\![v']\!](y).$$

Hence

$$[\![u \cdot v]\!](x) = [\![u' \cdot v']\!](x).$$

Since this holds for every $x \in U$,

$$u \cdot v \equiv_U u' \cdot v'.$$

□

The quotient

$$\mathsf{ERep}_U(\mathcal{Q}) = \mathcal{Q}^*/\equiv_U$$

therefore inherits a well-defined multiplication

$$[u]_U * [v]_U = [u \cdot v]_U.$$

We call

$$\mathsf{ERep}_U(\mathcal{Q})$$

the *effective repair algebra generated by* $\mathcal{Q}$ *on* U.

Theorem 4.57 (Semilattice Structure of Effective Repair)**.** *Let*

$$\mathcal{Q} = \{r_1, \ldots, r_n\}$$

be a finite commuting idempotent repair family on U.

Then

$$(\mathsf{ERep}_U(\mathcal{Q}), *, [\varepsilon]_U)$$

is a finite commutative idempotent monoid.

Equivalently, its multiplication defines a join-semilattice structure.

Proof. Associativity and the identity element are inherited from the quotient of the free monoid

$$\mathcal{Q}^*.$$

It remains to prove commutativity and idempotence.

Let

$$u, v \in \mathcal{Q}^*.$$

By canonicalization, the effective transformation induced by a repair word depends only upon its support. Thus

$$u \cdot v$$

and

$$v \cdot u$$

have the same support:

$$\operatorname{supp}(u \cdot v) = \operatorname{supp}(u) \cup \operatorname{supp}(v) = \operatorname{supp}(v \cdot u).$$

Therefore,

$$u \cdot v \equiv_U v \cdot u,$$

and hence

$$[u]_U * [v]_U = [v]_U * [u]_U.$$

Similarly,

$$\operatorname{supp}(u \cdot u) = \operatorname{supp}(u),$$

so

$$u \cdot u \equiv_U u.$$

Thus

$$[u]_U * [u]_U = [u]_U.$$

Hence the quotient is a finite commutative idempotent monoid. □

The preceding theorem makes precise a useful interpretation of repair composition.

In a general repair algebra, composition records an ordered temporal sequence of corrective actions.

In a commuting idempotent repair system, however, the effective repair class records only the collection of distinct corrective effects that have been established.

Repeated application contributes nothing new, and execution order becomes irrelevant.

This motivates an intrinsic ordering on effective repair classes.

Definition 4.58 (Repair-Effect Order)**.** For

$$a, b \in \mathsf{ERep}_U(\mathcal{Q}),$$

define

$$a \preceq b$$

when

$$a * b = b.$$

Operationally,

$$a \preceq b$$

means that all effective repair information represented by a is already contained in b.

Proposition 4.59 (Repair-Effect Order is a Partial Order)**.** *The relation*

$$\preceq$$

is a partial order on

$$\mathsf{ERep}_U(\mathcal{Q}).$$

Proof. For reflexivity, idempotence gives

$$a * a = a,$$

so

$$a \preceq a.$$

For antisymmetry, suppose

$$a \preceq b$$

and

$$b \preceq a.$$

Then

$$a * b = b$$

and

$$b * a = a.$$

By commutativity,

$$a * b = b * a,$$

so

$$a = b.$$

For transitivity, suppose

$$a \preceq b$$

and

$$b \preceq c.$$

Then

$$a * b = b$$

and

$$b * c = c.$$

Using associativity,

$$a * c = a * (b * c) = (a * b) * c = b * c = c.$$

Hence

$$a \preceq c.$$

□

Theorem 4.60 (Join Characterization)**.** *For any*

$$a, b \in \mathsf{ERep}_U(\mathcal{Q}),$$

the product

$$a * b$$

is the least upper bound of a *and* b *with respect to*

$$\preceq .$$

Thus

$$a \vee b = a * b.$$

Proof. First,

$$a * (a * b) = (a * a) * b = a * b,$$

so

$$a \preceq a * b.$$

Similarly,

$$b \preceq a * b.$$

Thus $a * b$ is an upper bound.

Now suppose c is another upper bound, so

$$a \preceq c \qquad \text{and} \qquad b \preceq c.$$

Hence

$$a * c = c$$

and

$$b * c = c.$$

Then

$$(a * b) * c = a * (b * c) = a * c = c.$$

Therefore,

$$a * b \preceq c.$$

Hence $a * b$ is the least upper bound. □

The identity class

$$[\varepsilon]_U$$

is the least element of this semilattice because

$$[\varepsilon]_U * a = a$$

for every effective repair class a.

Thus the effective repair algebra has the structure

$$(\mathsf{ERep}_U(\mathcal{Q}), \preceq, \vee, [\varepsilon]_U)\,,$$

where composition corresponds to accumulation of effective repair structure.

The semilattice need not be isomorphic to the full Boolean lattice

$$2^{\mathcal{Q}}.$$

The support map

$$\mathrm{supp} : \mathcal{Q}^* \to 2^{\mathcal{Q}}$$

is surjective at the syntactic level, but distinct subsets of $\mathcal{Q}$ may induce the same effective transformation on U.

Consequently, the effective repair semilattice is naturally a quotient of the Boolean join-semilattice

$$\left(2^{\mathcal{Q}}, \cup, \varnothing\right).$$

Proposition 4.61 (Boolean Quotient Representation)**.** *Define*

$$\Theta : 2^{\mathcal{Q}} \to \mathsf{ERep}_U(\mathcal{Q})$$

by choosing any word containing each element of

$$J \subseteq \mathcal{Q}$$

exactly once and setting

$$\Theta(J) = [w_J]_U.$$

Then Θ *is a surjective join-semilattice homomorphism satisfying*

$$\Theta(J \cup K) = \Theta(J) \vee \Theta(K).$$

Proof. Because the primitive operations commute on U, the equivalence class

$$[w_J]_U$$

does not depend upon the order chosen for the elements of J. Thus Θ is well defined.

Every effective repair class has a canonical representative whose letters are distinct, so Θ is surjective.

Finally, concatenating representatives for J and K, followed by canonicalization, yields precisely the effective repair associated with

$$J \cup K.$$

Hence

$$\Theta(J \cup K) = \Theta(J) * \Theta(K) = \Theta(J) \vee \Theta(K).$$

□

This representation separates two levels of repair structure.

The Boolean semilattice

$$2^{\mathcal{Q}}$$

records which primitive repair operations have been selected.

The effective repair semilattice

$$\mathsf{ERep}_U(\mathcal{Q})$$

records which distinct state transformations those selections actually produce.

The kernel of

$$\Theta$$

therefore measures algebraic redundancy among primitive repair operations.

Definition 4.62 (Repair Redundancy)**.** A primitive repair operation

$$r \in \mathcal{Q}$$

is *redundant relative to* $J \subseteq \mathcal{Q}$ on U when

$$\Theta(J \cup \{r\}) = \Theta(J).$$

Thus r is redundant relative to J when adding r contributes no new effective transformation beyond those already generated by J.

Definition 4.63 (Irredundant Repair Set)**.** A subset

$$J \subseteq \mathcal{Q}$$

is *irredundant on* U when

$$\Theta(J \setminus \{r\}) \neq \Theta(J)$$

for every

$$r \in J.$$

Irredundant repair sets provide minimal algebraic descriptions of effective repair transformations.

A canonical repair word need not be irredundant: canonicalization removes repetition, but it does not remove generators whose effect is implied by other generators.

This distinction separates two different forms of simplification:

$$\text{syntactic canonicalization} \quad \text{and} \quad \text{semantic minimization.}$$

The first follows solely from commutativity and idempotence.

The second depends upon the specific algebraic relations among the repair transformations.

This distinction becomes important when repair depth is interpreted as an optimization problem. A successful pathway may be canonical while still containing semantically redundant primitive operations.

Accordingly, minimal repair depth corresponds not merely to finding a successful subset of primitive repairs, but to finding a smallest subset whose effective join reaches the required viable semantic class.

4.5.2 Semantic Minimization and Minimal Repair Sets

Canonicalization reduces a repair word to a representative containing each primitive repair operation at most once. This reduction is syntactic: it follows from commutativity and idempotence alone.

A second reduction is possible when distinct subsets of primitive repairs induce the same effective transformation. This motivates a notion of semantic minimality.

Let

$$\mathcal{Q} = \{r_1, \ldots, r_n\}$$

be a finite commuting idempotent repair family on a stable region

$$U \subseteq S,$$

and let

$$\Theta : 2^{\mathcal{Q}} \to \mathsf{ERep}_U(\mathcal{Q})$$

be the Boolean quotient map defined above.

Definition 4.64 (Semantically Redundant Repair)**.** Let

$$J \subseteq \mathcal{Q}$$

and let

$$r \in J.$$

The repair operation r is *semantically redundant in J on U* when

$$\Theta(J) = \Theta(J \setminus \{r\}).$$

Equivalently, removing r from the repair set does not change the effective repair transformation on U.

Definition 4.65 (Irredundant Repair Set)**.** A subset

$$J \subseteq \mathcal{Q}$$

is *irredundant on U* when

$$\Theta(J) \neq \Theta(J \setminus \{r\})$$

for every

$$r \in J.$$

Thus every primitive operation in an irredundant repair set contributes essentially to the effective repair transformation represented by that set.

Proposition 4.66 (Existence of Irredundant Representatives)**.** *For every*

$$J \subseteq \mathcal{Q},$$

there exists an irredundant subset

$$J_{\min} \subseteq J$$

such that

$$\Theta(J_{\min}) = \Theta(J).$$

Proof. Because J is finite, repeatedly remove any element

$$r \in J$$

satisfying

$$\Theta(J) = \Theta(J \setminus \{r\}).$$

Each removal strictly decreases the cardinality of the current subset while preserving its image under Θ. Since the original set is finite, the process terminates after finitely many steps.

The resulting subset $J_{\min}$ contains no semantically redundant element and therefore is irredundant. By construction,

$$\Theta(J_{\min}) = \Theta(J).$$

□

The irredundant representative need not be unique. Distinct primitive repair sets may contain different generators while nevertheless inducing the same effective repair transformation.

This leads naturally to a numerical invariant measuring the smallest primitive description of an effective repair.

Definition 4.67 (Semantic Repair Rank)**.** For

$$a \in \mathsf{ERep}_U(\mathcal{Q}),$$

define the *semantic repair rank* of a by

$$\rho_U(a) = \min \{|J| \mid J \subseteq \mathcal{Q},\ \Theta(J) = a\}.$$

Because $\mathcal{Q}$ is finite and Θ is surjective, the minimum exists.

The semantic repair rank measures the smallest number of primitive repair operations required to realize a given effective repair transformation.

Proposition 4.68 (Rank Bounds)**.** *For every*

$$a \in \mathsf{ERep}_U(\mathcal{Q}),$$

one has

$$0 \leq \rho_U(a) \leq n.$$

Moreover,

$$\rho_U([\varepsilon]_U) = 0.$$

Proof. Every effective repair class has a representative arising from some subset

$$J \subseteq \mathcal{Q}.$$

Since

$$|J| \leq n,$$

the upper bound follows.

The lower bound is immediate from cardinality.

Finally,

$$\Theta(\varnothing) = [\varepsilon]_U,$$

so

$$\rho_U([\varepsilon]_U) = 0.$$

□

Semantic repair rank is a property of an effective transformation. Repair depth, by contrast, is defined relative to an initial viable state and a damage operation. The two notions are therefore related but distinct.

We now connect them.

Definition 4.69 (Successful Repair Sets)**.** For

$$s \in V$$

and

$$d \in \mathcal{D},$$

define

$$\mathsf{Succ}_{\mathcal{A}}(s, d) = \{J \subseteq \mathcal{Q} \mid R_J(d(s)) \in V \cap [s]_\eta\},$$

where

$$R_J$$

denotes the effective transformation obtained by composing the primitive repair operations in J.

Because the generators commute on U, the order chosen to evaluate R_J does not affect its effective action on U.

Recoverability in the finite commuting idempotent setting is therefore equivalent to

$$\mathsf{Succ}_{\mathcal{A}}(s, d) \neq \varnothing.$$

Theorem 4.70 (Minimal-Subset Characterization of Repair Depth)**.** *Suppose*

$$d(s) \in U$$

and every repair pathway relevant to (s, d) *remains within the stable region* U*.*

If

$$(s, d)$$

is recoverable, then

$$\delta_{\mathcal{A}}(s, d) = \min \{|J| \mid J \in \mathsf{Succ}_{\mathcal{A}}(s, d)\}.$$

Proof. Let

$$w$$

be a successful repair word of minimal length. By finite canonicalization, there exists a canonical repair word

$$\mathrm{can}(w)$$

whose letters are distinct and whose effective action on U agrees with that of w.

Let

$$J_w \subseteq \mathcal{Q}$$

be the support of $\mathrm{can}(w)$. Then

$$|J_w| = |\,\mathrm{can}(w)| \leq |w| = \delta_{\mathcal{A}}(s, d),$$

and

$$J_w \in \mathsf{Succ}_{\mathcal{A}}(s, d).$$

Hence

$$\min \{|J| \mid J \in \mathsf{Succ}_{\mathcal{A}}(s, d)\} \leq \delta_{\mathcal{A}}(s, d).$$

Conversely, let

$$J \in \mathsf{Succ}_{\mathcal{A}}(s, d).$$

Choose a repair word containing each element of J exactly once. This word has length

$$|J|$$

and is successful by the definition of

$$\mathsf{Succ}_{\mathcal{A}}(s, d).$$

Therefore,

$$\delta_{\mathcal{A}}(s, d) \leq |J|.$$

Taking the minimum over all successful repair sets gives

$$\delta_{\mathcal{A}}(s, d) \leq \min \{|J| \mid J \in \mathsf{Succ}_{\mathcal{A}}(s, d)\}.$$

Combining the two inequalities proves the result. □

The theorem converts minimal repair depth from a search over arbitrary repair words into a finite combinatorial optimization problem.

Instead of searching

$$\mathcal{Q}^*,$$

one searches

$$2^{\mathcal{Q}}$$

for a smallest subset whose effective join returns the damaged state to the required viable semantic class.

Equivalently,

$$\delta_{\mathcal{A}}(s,d) = \min\left\{|J| \mid R_J(d(s)) \in V \cap [s]_\eta\right\}.$$

This formulation exposes the computational content of repair depth. It also suggests a natural decision problem.

Definition 4.71 (Bounded Repair-Set Recovery)**.** The *bounded repair-set recovery problem* asks:

Given a finite repair algebra, a viable state

$$s \in V,$$

a damage operation

$$d \in \mathcal{D},$$

and an integer

$$k \geq 0,$$

does there exist a repair set

$$J \subseteq \mathcal{Q}$$

such that

$$|J| \leq k$$

and

$$R_J(d(s)) \in V \cap [s]_\eta?$$

Thus bounded repair-set recovery asks whether semantic recovery can be achieved using a repair set of cardinality at most k.

The optimization form seeks

$$\delta_{\mathcal{A}}(s,d),$$

while the decision form asks whether

$$\delta_{\mathcal{A}}(s,d) \leq k.$$

In the commuting-idempotent setting, bounded repair-set recovery is equivalent to the corresponding bounded repair-depth question because every effective repair pathway has a representative using each primitive repair at most once. This provides the bridge from the algebraic theory of repair to the computational complexity of finding minimal regenerative pathways.

4.5.3 Computational Complexity of Bounded Repair Depth

The definition of Bounded Repair Depth becomes a complexity-theoretic decision problem only after specifying how a finite repair algebra is represented as input.

An explicit representation of every element of a transformation monoid may be exponentially larger than its primitive generating family. We therefore work with a compact representation in which the primitive repair operations are given directly and their action on states is efficiently computable.

Definition 4.72 (Polynomially Evaluable Finite Repair Instance)**.** A *polynomially evaluable finite repair instance* consists of:

1. a finite state representation S;
2. a finite primitive repair basis
$$\mathcal{Q} = \{r_1, \dots, r_n\};$$
3. a designated viable state
$$s \in V;$$
4. a designated damage transformation
$$d;$$
5. a viability predicate
$$\chi_V : S \to \{0, 1\};$$
6. a semantic observation map
$$\eta : S \to \Omega;$$
7. an integer bound
$$k \geq 0;$$

such that the following operations can be computed in time polynomial in the size of the encoded instance:

$$d(x), \qquad r_i(x), \qquad \chi_V(x), \qquad \eta(x).$$

The primitive transformations themselves are represented by polynomial-size descriptions rather than by complete transition tables over the entire state space.

This representation permits exponentially large state spaces or repair monoids to be described compactly, provided that individual transformations can be evaluated efficiently.

Definition 4.73 (Bounded Repair Depth Decision Problem)**.** The *Bounded Repair Depth problem*, abbreviated
$$\text{BRD},$$
is the following decision problem.

Input: A polynomially evaluable finite repair instance
$$(\mathcal{A}, \mathcal{Q}, s, d, k).$$

Question: Does there exist a repair word

$$w \in \mathcal{Q}^*$$

such that

$$|w| \leq k$$

and

$$[\![w]\!](d(s)) \in V \cap [s]_\eta?$$

Equivalently, does

$$\delta_{\mathcal{A}}(s, d) \leq k?$$

For unrestricted repair systems, the relationship between the numerical bound k, repeated operations, and the encoding size must be handled carefully. The strongest complexity result below therefore concerns a restricted class for which canonicalization guarantees certificates of polynomial length.

Definition 4.74 (Commuting-Idempotent Bounded Repair Depth)**.** The problem

CI-BRD

is the restriction of BRD to instances in which:

1. the primitive repair basis
$$\mathcal{Q} = \{r_1, \ldots, r_n\}$$
is finite;
2. the primitive repairs are pairwise commuting on the relevant stable region;
3. every primitive repair is idempotent on that region.

By canonicalization, every effective repair pathway then has a representative using each primitive operation at most once.

Proposition 4.75 (CI-BRD is in NP)**.** *The decision problem*

CI-BRD

belongs to the complexity class

NP.

Proof. Suppose an instance contains

$$n$$

primitive repair operations.

A certificate consists of a subset

$$J \subseteq \mathcal{Q}$$

with

$$|J| \leq k.$$

Because the primitive operations commute and are idempotent, no operation need occur more than once. Hence the certificate contains at most

$$n$$

primitive-operation indices and therefore has size polynomial in the input description.

To verify the certificate, begin with

$$x_0 = d(s)$$

and apply each primitive repair in J, obtaining

$$x_J = R_J(d(s)).$$

By the assumption of polynomial evaluability, this computation requires polynomial time.

The verifier then checks

$$x_J \in V$$

using

$$\chi_V,$$

and checks

$$\eta(x_J) = \eta(s).$$

Thus a proposed successful repair set of size at most k can be verified in polynomial time.

Therefore,

$$\text{CI-BRD} \in \mathbf{NP}.$$

□

We next show that the problem remains computationally difficult even under substantial algebraic restrictions.

Definition 4.76 (Boolean Monotone Repair Algebra)**.** A *Boolean monotone repair algebra* has state space

$$S = \{0,1\}^m.$$

For a subset

$$C \subseteq \{1, \dots, m\},$$

define the repair operation

$$r_C : S \to S$$

by

$$(r_C(x))_j = \begin{cases} 1, & j \in C, \\ x_j, & j \notin C. \end{cases}$$

Thus r_C changes the coordinates indexed by C to 1 and leaves all other coordinates unchanged.

Each such transformation is monotone with respect to the coordinatewise order on

$$\{0,1\}^m.$$

Moreover,

$$r_C \circ r_C = r_C,$$

so every Boolean monotone repair is idempotent.

For any two subsets C, D,

$$r_C \circ r_D = r_D \circ r_C,$$

because both compositions set precisely the coordinates in

$$C \cup D$$

to 1.

Hence these primitive repairs form a commuting idempotent family.

Theorem 4.77 (NP-Completeness of Bounded Repair Depth)**.** *The problem*

CI-BRD

is NP-complete.

More strongly, it remains NP-complete when restricted to Boolean monotone repair algebras whose primitive repair operations are pairwise commuting and idempotent.

Proof. Membership in

NP

was established above.

It remains to prove NP-hardness.

We give a polynomial-time reduction from

Set Cover,

using the classical NP-complete covering problem [7].

Let an instance of Set Cover be given by a finite universe

$$X = \{1, \ldots, m\},$$

a family of subsets

$$\mathcal{C} = \{C_1, \ldots, C_n\}, \qquad C_i \subseteq X,$$

and an integer

$$k.$$

The question is whether there exists a subfamily of at most k sets whose union is all of X.

Construct the state space

$$S = \{0, 1\}^m.$$

Let the designated viable state be

$$s = \mathbf{1} = (1, \ldots, 1).$$

Define the viability region by

$$V = \{\mathbf{1}\}.$$

Let the semantic observation map be the identity:

$$\eta(x) = x.$$

Define a damage transformation

$$d : S \to S$$

by

$$d(x) = \mathbf{0} = (0, \ldots, 0)$$

for every

$$x \in S.$$

In particular,

$$d(s) = \mathbf{0}.$$

For each subset

$$C_i \in \mathcal{C},$$

introduce a primitive repair operation

$$r_i = r_{C_i}$$

defined coordinatewise by

$$(r_i(x))_j = \begin{cases} 1, & j \in C_i, \\ x_j, & j \notin C_i. \end{cases}$$

Let

$$\mathcal{Q} = \{r_1, \ldots, r_n\}.$$

As observed above, every r_i is monotone and idempotent, and all pairs r_i, r_j commute.

Consider any subset

$$J \subseteq \{1, \ldots, n\}.$$

Applying the repairs indexed by J to the damaged state gives a vector whose j-th coordinate is 1 precisely when

$$j \in \bigcup_{i \in J} C_i.$$

Consequently,

$$R_J(\mathbf{0}) = \mathbf{1}$$

if and only if

$$\bigcup_{i \in J} C_i = X.$$

Since

$$V = \{\mathbf{1}\}$$

and

$$\eta = \mathrm{id},$$

the repair set J is successful precisely when

$$R_J(d(s)) = \mathbf{1}.$$

Therefore,

$$J$$

is a successful repair set of cardinality at most k if and only if

$$\{C_i \mid i \in J\}$$

is a set cover of X of cardinality at most k.

Hence

$$(X, \mathcal{C}, k) \in \textsc{Set Cover}$$

if and only if the constructed repair instance satisfies

$$\delta_{\mathcal{A}}(s, d) \leq k.$$

The construction uses m-bit states and m-bit descriptions of each primitive repair operation, so its size is polynomial in the size of the original Set Cover instance.

Thus

$$\textsc{Set Cover} \leq_p \text{CI-BRD}.$$

Therefore

$$\text{CI-BRD}$$

is NP-hard.

Together with membership in

$$\mathbf{NP},$$

we conclude that

$$\boxed{\text{CI-BRD is NP-complete.}}$$

□

The reduction yields a stronger structural conclusion than NP-completeness alone.

The computational difficulty does not arise from noncommutativity, repeated repair operations, complicated semantic observations, or arbitrary state transitions.

Indeed, the reduction uses repair operations satisfying all of the following:

$$\text{deterministic}, \quad \text{monotone}, \quad \text{idempotent}, \quad \text{pairwise commuting}.$$

The semantic observation map is simply the identity, and the viability region contains a single target state.

Thus the difficulty of minimal repair may persist even when the algebraic dynamics themselves are extremely simple.

The source of complexity is instead combinatorial:

$$\boxed{\text{Which primitive repair effects should be selected?}}$$

This distinguishes the complexity of executing a repair pathway from the complexity of finding a smallest successful pathway.

For the Boolean construction above, execution of any proposed repair set is polynomial-time, while choosing a smallest successful repair set is NP-hard.

Corollary 4.78 (NP-Hardness of Minimum Repair Depth)**.** *Computing*

$$\delta_{\mathcal{A}}(s, d)$$

for compactly represented finite commuting-idempotent repair systems is NP-hard.

Proof. If minimum repair depth could be computed in polynomial time, then CI-BRD could be decided in polynomial time by computing

$$\delta_{\mathcal{A}}(s, d)$$

and testing whether

$$\delta_{\mathcal{A}}(s, d) \leq k.$$

Since CI-BRD is NP-hard, computing minimum repair depth is also NP-hard. □

The Set Cover construction also reveals that, in this restricted Boolean setting, repair depth coincides exactly with minimum cover size.

If

$$\tau(X, \mathcal{C})$$

denotes the cardinality of a smallest set cover, then the associated repair instance satisfies

$$\delta_{\mathcal{A}}(s, d) = \tau(X, \mathcal{C}).$$

Thus a classical combinatorial optimization quantity appears directly as a repair-theoretic invariant.

This correspondence provides the first explicit complexity-theoretic boundary within the repair-algebra framework:

$$\text{verification of a proposed repair} \quad \text{is efficient,}$$

while

$$\text{optimization of repair depth} \quad \text{can be computationally intractable.}$$

4.5.4 Tractable Repair Classes

The NP-completeness result above shows that commutativity, idempotence, and monotonicity do not by themselves make minimal repair computationally easy. The Boolean reduction remains difficult because different primitive repair operations may overlap in the coordinates they restore.

This suggests that computational hardness is associated not merely with the number of available repair operations, but with dependencies among their effects.

We now identify a restricted class in which these dependencies disappear and minimum repair depth becomes efficiently computable.

Definition 4.79 (Disjoint-Support Boolean Repair System)**.** Let

$$S = \{0, 1\}^m,$$

and let

$$\mathcal{Q} = \{r_{C_1}, \ldots, r_{C_n}\}$$

be a Boolean monotone repair family as defined above.

The family has *disjoint support* when

$$C_i \cap C_j = \varnothing$$

for every

$$i \neq j.$$

In a disjoint-support system, each state coordinate can be modified by at most one primitive repair operation.

Consequently, no primitive repair can substitute for another on a coordinate that only the latter controls.

Definition 4.80 (Required Repair Set)**.** Let

$$x \in \{0,1\}^m$$

be a damaged state and let

$$t \in \{0,1\}^m$$

be a target state satisfying

$$x_j \leq t_j$$

for every coordinate j.

For a disjoint-support repair family

$$\mathcal{Q} = \{r_{C_1}, \ldots, r_{C_n}\},$$

define

$$\mathrm{Req}(x,t) = \{r_{C_i} \in \mathcal{Q} \mid \exists j \in C_i \text{ such that } x_j = 0 \text{ and } t_j = 1\}.$$

Thus

$$\mathrm{Req}(x,t)$$

contains exactly those primitive repair operations whose supports contain at least one coordinate that must change from 0 to 1.

Proposition 4.81 (Necessity of Required Repairs)**.** *Let*

$$\mathcal{Q} = \{r_{C_1}, \ldots, r_{C_n}\}$$

be a disjoint-support Boolean repair family.

If a repair set

$$J \subseteq \mathcal{Q}$$

transforms x into t, then

$$\mathrm{Req}(x,t) \subseteq J.$$

Proof. Let

$$r_{C_i} \in \mathrm{Req}(x,t).$$

By definition, there exists a coordinate

$$j \in C_i$$

such that

$$x_j = 0 \qquad \text{and} \qquad t_j = 1.$$

Because the supports are pairwise disjoint, no primitive repair operation other than

$$r_{C_i}$$

can modify coordinate j.

Therefore any repair set transforming x into t must contain

$$r_{C_i}.$$

Since this holds for every member of

$$\mathrm{Req}(x,t),$$

we obtain

$$\mathrm{Req}(x,t) \subseteq J.$$

□

Necessity alone does not guarantee recoverability. A required repair may also set a coordinate to 1 that the target requires to remain 0.

This motivates an admissibility condition.

Definition 4.82 (Target-Compatible Repair)**.** A primitive Boolean repair

$$r_C$$

is *target-compatible with* (x,t) when

$$t_j = 1$$

for every

$$j \in C$$

such that

$$x_j = 0.$$

Equivalently, applying r_C introduces no 1-coordinate that conflicts with the target state.

For damaged and target states

$$x, t \in \{0,1\}^m,$$

define the target-relative repair depth

$$\delta_{\mathcal{Q}}(x,t) = \min\left\{|J| \mid J \subseteq \mathcal{Q},\ R_J(x) = t\right\},$$

whenever such a repair set exists.

Theorem 4.83 (Unique Minimal Repair under Disjoint Support)**.** *Let*

$$\mathcal{Q} = \{r_{C_1}, \ldots, r_{C_n}\}$$

be a disjoint-support Boolean repair family.

Let

$$x, t \in \{0,1\}^m$$

satisfy

$$x_j \leq t_j$$

for every j*.*

Assume:

1. *every coordinate satisfying*
$$x_j = 0, \qquad t_j = 1$$
lies in the support of some primitive repair;

2. *every repair in*
$$\mathrm{Req}(x, t)$$
is target-compatible with (x, t).

Then
$$\mathrm{Req}(x, t)$$
transforms x *into* t*, and it is the unique inclusion-minimal successful repair set.*
Moreover,
$$\delta_{\mathcal{Q}}(x, t) = |\,\mathrm{Req}(x, t)|.$$

Proof. Let
$$R_{\mathrm{Req}}$$
denote the composition of all repairs in
$$\mathrm{Req}(x, t).$$
Because the repair supports are pairwise disjoint, the order of composition is irrelevant.
Consider a coordinate j.
If
$$x_j = t_j,$$
then either no required repair changes j, or a required repair contains j. In the latter case, target compatibility guarantees that
$$t_j = 1,$$
and since $x_j = t_j$, the coordinate already equals 1. Thus its final value remains t_j.
If
$$x_j = 0 \qquad \text{and} \qquad t_j = 1,$$
then by assumption j belongs to the support of some primitive repair. That repair belongs to
$$\mathrm{Req}(x, t)$$
and therefore sets coordinate j to 1.
Hence
$$R_{\mathrm{Req}}(x) = t.$$
Now let J be any successful repair set. By the preceding proposition,
$$\mathrm{Req}(x, t) \subseteq J.$$
Therefore no proper subset of
$$\mathrm{Req}(x, t)$$
can be successful, and every successful repair set contains it.
Thus
$$\mathrm{Req}(x, t)$$
is the unique inclusion-minimal successful repair set.

Its cardinality is consequently the minimum number of primitive repairs required to reach t, so

$$\delta_{\mathcal{Q}}(x,t) = |\operatorname{Req}(x,t)|.$$

□

The preceding theorem yields an immediate complexity consequence.

Corollary 4.84 (Polynomial-Time Bounded Repair Depth under Disjoint Support)**.** *Bounded Repair Depth for disjoint-support Boolean repair systems can be decided in polynomial time.*

Proof. Given a damaged state

$$x$$

and target state

$$t,$$

scan the primitive repair supports and construct

$$\operatorname{Req}(x,t).$$

During the same scan, verify that every coordinate requiring restoration is covered and that every required repair is target-compatible.

If either condition fails, no successful repair exists.

Otherwise, the unique minimal successful repair set is

$$\operatorname{Req}(x,t).$$

The bounded-depth question therefore reduces to testing

$$|\operatorname{Req}(x,t)| \leq k.$$

All operations require time polynomial in the explicit descriptions of the state vectors and primitive repair supports. □

Thus the complexity landscape contains a sharp structural contrast:

$$\begin{aligned} \text{overlapping repair supports} &\implies \text{combinatorial choice,} \\ \text{disjoint repair supports} &\implies \text{forced repair selection.} \end{aligned}$$

The Set Cover reduction exploits the first regime. An impaired coordinate may be recoverable by several different primitive operations, so a repair system must choose among overlapping alternatives.

Under disjoint support, this ambiguity disappears. Every coordinate requiring repair identifies at most one primitive operation capable of restoring it.

The distinction suggests that repair complexity is governed by the interaction structure among primitive repair effects.

Definition 4.85 (Repair Interaction Graph)**.** For a Boolean repair family

$$\mathcal{Q} = \{r_{C_1}, \ldots, r_{C_n}\},$$

define the *repair interaction graph*

$$G_{\mathcal{Q}} = (\mathcal{Q}, E)$$

by placing an edge

$$\{r_{C_i}, r_{C_j}\} \in E$$

precisely when

$$C_i \cap C_j \neq \varnothing.$$

The disjoint-support case corresponds exactly to

$$E = \varnothing.$$

At the opposite extreme, highly overlapping repair supports may generate a dense interaction graph and a large family of alternative recovery pathways.

The interaction graph therefore provides a combinatorial representation of dependency among primitive repairs.

This suggests a broader structural program: rather than classifying repair problems solely by the number of primitive operations, one may classify them by the geometry of their interaction graph.

Parameters such as connected-component size, degree, and treewidth may then govern the complexity of minimal repair.

The NP-completeness theorem and the disjoint-support tractability theorem represent the first two endpoints of this spectrum:

$$\boxed{\text{repair interaction} \longrightarrow \text{repair-selection complexity.}}$$

5 Regenerative Runtimes

5.1 Biological Runtime States

The repair algebra developed above describes the transformations by which a system may be damaged, modified, and restored. A regenerative biological system, however, is not merely a collection of transformations. It is a system whose state evolves through time while remaining subject to viability and homeostatic constraints.

We therefore lift the repair algebra into a runtime semantics.

Definition 5.1 (Regenerative Runtime)**.** A *regenerative runtime* is a tuple

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi),$$

where:

1. S is the runtime state space;
2. $V \subseteq S$ is the viability region;
3. $$\eta : S \to \Omega$$ is the homeostatic signature map;
4. $\mathcal{D}$ is a collection of damage transformations;
5. $\mathcal{R}$ is a collection of repair transformations;

6. $\mathcal{T}$ is a collection of admissible runtime transformations;

7. $\mathcal{C}$ is the runtime context space;

8. $$\Pi$$

   is a runtime policy specifying which admissible transition may be executed from a given state and runtime context.

The components

$$(S, V, \Omega, \eta, \mathcal{D}, \mathcal{R})$$

are inherited from the underlying repair algebra

$$\mathcal{A} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \circ, \mathrm{id}_S),$$

while

$$(\mathcal{T}, \mathcal{C}, \Pi)$$

adds its execution semantics.

The distinction between repair algebra and regenerative runtime is therefore structural.

The repair algebra determines which transformations exist.

The regenerative runtime determines how transformations are selected and executed through time.

Definition 5.2 (Runtime Configuration)**.** A *runtime configuration* is a pair

$$(x, c) \in S \times \mathcal{C},$$

where x is the current biological state and c is the current runtime context.

The context may encode environmental conditions, available resources, regulatory signals, temporal information, or other variables relevant to transition selection.

When no explicit contextual information is required, the runtime configuration may be identified with its state

$$x \in S.$$

Definition 5.3 (Runtime Transition)**.** A *runtime transition* is an admissible transformation

$$\tau \in \mathcal{T}$$

acting on the current runtime state.

We write

$$x \xrightarrow{\tau} y$$

when

$$y = \tau(x).$$

A runtime transition need not itself be a repair operation. It may represent ordinary state evolution, adaptation, damage, compensation, or repair.

Thus the transition family may contain several distinguished classes:

$$\mathcal{T} = \mathcal{T}_{\text{ord}} \cup \mathcal{D} \cup \mathcal{R} \cup \mathcal{T}_{\text{ad}} \cup \mathcal{T}_{\text{comp}},$$

where the components represent ordinary evolution, damage, repair, adaptation, and compensation respectively.

These classes need not be disjoint. A transformation may play different roles relative to different states or runtime contexts.

Definition 5.4 (Runtime Policy)**.** A *runtime policy* is a partial function

$$\Pi : S \times \mathcal{C} \rightharpoonup \mathcal{T}$$

such that whenever

$$\Pi(x, c) = \tau,$$

the transformation τ is admissible from state x under context c.

The resulting transition is

$$x \xrightarrow{\Pi(x,c)} \Pi(x, c)(x).$$

A deterministic runtime policy selects at most one transition from each configuration.

More generally, one may replace Π by a relation or set-valued map when multiple transitions are simultaneously admissible. The deterministic case is sufficient for the present development and provides a direct executable semantics.

Definition 5.5 (Runtime Execution)**.** Let

$$(x_0, c_0)$$

be an initial runtime configuration.

A *context trajectory* is a finite or infinite sequence

$$c_0, c_1, c_2, \ldots$$

of elements of $\mathcal{C}$, representing the runtime context presented at successive execution steps.

Relative to a context trajectory, a *runtime execution* is a finite or infinite sequence

$$x_0 \xrightarrow{\tau_0} x_1 \xrightarrow{\tau_1} x_2 \xrightarrow{\tau_2} \cdots$$

such that for every time index t for which the execution continues,

$$\tau_t = \Pi(x_t, c_t)$$

and

$$x_{t+1} = \tau_t(x_t).$$

Equivalently,

$$x_{t+1} = \Pi(x_t, c_t)(x_t)$$

whenever the policy is defined.

The context trajectory may be externally supplied or generated by an environmental process. No particular dynamics on $\mathcal{C}$ are assumed unless explicitly specified.

The runtime therefore generates a discrete trajectory

$$(x_t)_{t\geq 0}$$

through the biological state space.

The state at time $t+1$ depends upon the current state, the runtime context, and the transition selected by the policy.

Definition 5.6 (Runtime Reachability)**.** For states

$$x, y \in S,$$

write

$$x \rightsquigarrow_{\mathfrak{R}} y$$

when there exists a finite context trajectory

$$c_0, c_1, \ldots, c_{m-1}$$

and a corresponding finite runtime execution

$$x = x_0 \xrightarrow{\tau_0} x_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{m-1}} x_m = y,$$

where

$$\tau_t = \Pi(x_t, c_t)$$

for every

$$0 \leq t < m.$$

The set of states reachable from x is

$$\mathrm{Reach}_{\mathfrak{R}}(x) = \{y \in S \mid x \rightsquigarrow_{\mathfrak{R}} y\}.$$

Reachability introduces a temporal distinction absent from the underlying repair algebra.

A transformation may exist algebraically without being executable from a particular runtime state under the current policy. Thus algebraic possibility does not imply runtime reachability.

Definition 5.7 (Viable Runtime Execution)**.** A runtime execution

$$x_0 \to x_1 \to x_2 \to \cdots$$

is *viable* over a time interval I when

$$x_t \in V$$

for every

$$t \in I.$$

It is *globally viable* when

$$x_t \in V$$

for every time at which the execution is defined.

A regenerative runtime need not remain viable after every perturbation. Damage may move the runtime outside V, after which repair dynamics may return it to the viable region.

Accordingly, regeneration is naturally expressed as a reachability property.

Definition 5.8 (Runtime Recoverability)**.** A state

$$x \in S$$

is *runtime-recoverable to a homeostatic class* $H \subseteq V$ when there exists

$$y \in H$$

such that

$$x \rightsquigarrow_{\mathfrak{R}} y.$$

For a viable reference state

$$s \in V,$$

define its viable semantic target by

$$H_s = V \cap [s]_\eta.$$

A damaged state x is therefore recoverable relative to s when

$$\mathrm{Reach}_{\mathfrak{R}}(x) \cap H_s \neq \varnothing.$$

This extends algebraic recoverability to execution semantics.

The repair algebra asks whether there exists a repair transformation capable of restoring viability and semantic equivalence.

The regenerative runtime asks whether such restoration is reachable through the transitions actually permitted by the runtime policy.

Proposition 5.9 (Runtime Recovery Implies Algebraic Recovery)**.** *Suppose every repair transition executed by*

$$\mathfrak{R}$$

belongs to the repair monoid

$$\mathcal{R}$$

of the underlying repair algebra.

Let

$$s \in V$$

and

$$d \in \mathcal{D}.$$

If there exists a finite runtime execution consisting entirely of repair transitions

$$d(s) \xrightarrow{r_1} x_1 \xrightarrow{r_2} \cdots \xrightarrow{r_m} y$$

with

$$y \in V \cap [s]_\eta,$$

then

$$(s, d)$$

is recoverable in the underlying repair algebra.

Proof. Because

$$r_1, \ldots, r_m \in \mathcal{R}$$

and $\mathcal{R}$ is closed under composition, the composite transformation

$$r = r_m \circ \cdots \circ r_1$$

belongs to

$$\mathcal{R}.$$

The runtime execution gives

$$r(d(s)) = y.$$

Since

$$y \in V \cap [s]_\eta,$$

the transformation r is a successful repair of d at s.

Therefore,

$$\mathrm{Rep}_{\mathcal{A}}(s, d) \neq \varnothing,$$

and hence

$$(s, d)$$

is recoverable. □

The converse need not hold.

A successful repair transformation may exist in the underlying algebra while the runtime policy never selects a pathway realizing that transformation. Thus

algebraic recoverability

is a statement about available repair structure, whereas

runtime recoverability

is a statement about executable repair behavior.

This distinction permits two fundamentally different forms of regenerative failure:

structural failure:	no successful repair exists in the repair algebra,
policy failure:	a successful repair exists but is not reached by the runtime.

The first is a limitation of available repair capability.

The second is a limitation of transition selection.

A regenerative runtime therefore separates the existence of a recovery mechanism from the execution policy required to realize it.

5.2 Homeostasis

The regenerative runtime introduced above evolves through a sequence of runtime states. We now formalize the conditions under which this evolution preserves, departs from, and returns to a designated homeostatic organization.

Homeostasis is not identified with a single microscopic state. Because the semantic observation map

$$\eta : S \to \Omega$$

may identify multiple internal states with the same homeostatic signature, homeostasis is naturally represented by a region of semantically equivalent viable states.

Definition 5.10 (Homeostatic Region)**.** Let

$$s \in V$$

be a viable reference state.

The *homeostatic region generated by* s is

$$H_s = V \cap [s]_\eta = \{x \in V \mid \eta(x) = \eta(s)\}.$$

Thus a homeostatic region contains all viable states realizing the same selected homeostatic signature.

In particular, homeostatic preservation does not require

$$x = s.$$

It requires only

$$x \in V$$

and

$$\eta(x) = \eta(s).$$

This distinction allows a regenerative runtime to undergo internal reorganization while preserving its higher-level identity or function.

Definition 5.11 (Homeostatic Execution)**.** Let

$$x_0 \xrightarrow{\tau_0} x_1 \xrightarrow{\tau_1} x_2 \xrightarrow{\tau_2} \cdots$$

be a runtime execution.

The execution is *homeostatic relative to* H_s over a time interval I when

$$x_t \in H_s$$

for every

$$t \in I.$$

It is *globally homeostatic relative to* H_s when

$$x_t \in H_s$$

for every time at which the execution is defined.

Global homeostasis is therefore stronger than viability.

A viable execution requires

$$x_t \in V,$$

whereas a homeostatic execution additionally requires

$$\eta(x_t) = \eta(s).$$

Hence

$$\text{homeostatic execution} \quad \Longrightarrow \quad \text{viable execution},$$

but the converse need not hold.

A system may remain viable while moving into a different semantic organization.

Definition 5.12 (Homeostatic Invariance)**.** A subset

$$H \subseteq V$$

is *invariant under a runtime policy* Π when, for every runtime configuration

$$(x, c)$$

with

$$x \in H$$

for which the policy is defined,

$$\Pi(x, c)(x) \in H.$$

Homeostatic invariance expresses persistence rather than recovery. Once the runtime enters an invariant homeostatic region, policy-controlled evolution cannot move it outside that region.

Proposition 5.13 (Persistence of an Invariant Homeostatic Region)**.** *Let*

$$H \subseteq V$$

be invariant under the runtime policy Π*.*

If a runtime execution begins at

$$x_0 \in H,$$

then

$$x_t \in H$$

for every subsequent time t for which the execution is defined.

Proof. The proof proceeds by induction on t.

By assumption,

$$x_0 \in H.$$

Suppose

$$x_t \in H.$$

Because H is invariant under Π,

$$x_{t+1} = \Pi(x_t, c_t)(x_t) \in H.$$

Therefore,

$$x_t \in H$$

for every time along the execution. □

The preceding proposition describes a system that never leaves its homeostatic region. Regenerative systems require a second notion: the ability to return after perturbation.

Definition 5.14 (Homeostatic Excursion)**.** Let

$$H \subseteq V$$

be a homeostatic region.

A *homeostatic excursion from* H is a finite runtime segment

$$x_0 \to x_1 \to \cdots \to x_m$$

such that

$$x_0 \in H,$$

at least one intermediate state satisfies

$$x_j \notin H,$$

and

$$x_m \in H.$$

The excursion is *viability-preserving* when

$$x_j \in V$$

for every

$$0 \leq j \leq m.$$

A homeostatic excursion therefore represents temporary displacement from the selected semantic organization followed by recovery.

The system may either remain viable throughout the excursion or temporarily leave the viability region before regeneration restores it.

Definition 5.15 (Homeostatic Return Time)**.** Let

$$H \subseteq V$$

and let

$$x_0 \to x_1 \to x_2 \to \cdots$$

be a runtime execution with

$$x_0 \notin H.$$

The *homeostatic return time* of the execution is

$$T_H(x_0) = \inf \{t \geq 0 \mid x_t \in H\}.$$

If the execution never reaches H, define

$$T_H(x_0) = \infty.$$

The return time measures recovery in runtime steps rather than physical time.

It is therefore distinct from repair depth. Repair depth minimizes over available primitive repair pathways, whereas homeostatic return time measures the number of transitions actually executed by the runtime policy.

Consequently,

$$\delta_{\mathcal{A}}(s, d)$$

and

$$T_{H_s}(d(s))$$

need not coincide.

The first is an algebraic optimum.

The second is an execution-dependent quantity.

Definition 5.16 (Regenerative Stability)**.** Let

$$H \subseteq V$$

be a homeostatic region and let

$$B \subseteq S$$

be a set of admissible perturbation states.

The region H is *regeneratively stable relative to* B under the runtime policy Π when

$$B \subseteq \{x \in S \mid x \rightsquigarrow_{\mathfrak{R}} H\},$$

where

$$x \rightsquigarrow_{\mathfrak{R}} H$$

means that some finite policy-generated runtime execution beginning at x reaches a state in H.

Thus regenerative stability does not require the runtime to remain inside H.

Instead, it requires states in the specified perturbation region to possess a runtime path back to H.

This separates two forms of homeostatic robustness:

invariant homeostasis	prevents departure from H,
regenerative homeostasis	permits departure but guarantees return from a specified region.

The distinction is fundamental for regenerative systems. A system capable only of invariant preservation can tolerate transitions that remain inside its homeostatic region. A regenerative system can additionally recover from states outside that region.

Definition 5.17 (Basin of Regenerative Recovery)**.** The *basin of regenerative recovery* of a homeostatic region H is

$$\mathcal{B}_{\mathfrak{R}}(H) = \{x \in S \mid x \rightsquigarrow_{\mathfrak{R}} H\}.$$

The basin

$$\mathcal{B}_{\mathfrak{R}}(H)$$

contains precisely those runtime states from which the runtime can return to the selected homeostatic region.

By definition,

$$H \subseteq \mathcal{B}_{\mathfrak{R}}(H),$$

provided that zero-step reachability is permitted.

States outside

$$\mathcal{B}_{\mathfrak{R}}(H)$$

are not recoverable to H under the current runtime policy.

Proposition 5.18 (Forward Closure of the Recovery Basin)**.** *Assume the runtime policy* Π *is deterministic.*

Let

$$H \subseteq V.$$

If

$$x \in \mathcal{B}_{\mathfrak{R}}(H)$$

and the policy selects a transition

$$x \to y$$

lying on a finite policy-generated execution from x *to* H*, then*

$$y \in \mathcal{B}_{\mathfrak{R}}(H).$$

Proof. Since

$$x \in \mathcal{B}_{\mathfrak{R}}(H),$$

there exists a finite policy-generated execution

$$x = x_0 \to x_1 \to \cdots \to x_m$$

with

$$x_m \in H.$$

If

$$y = x_1,$$

then the suffix

$$y = x_1 \to x_2 \to \cdots \to x_m$$

is itself a finite runtime execution reaching H.

Therefore,

$$y \in \mathcal{B}_{\mathfrak{R}}(H).$$

□

For a deterministic policy, every state in the recovery basin therefore lies on a trajectory whose remaining states continue toward the same recoverable region.

This gives the homeostatic dynamics a natural decomposition.

Relative to a chosen homeostatic region H, runtime states may be divided into:

$$\begin{aligned} H &= \text{currently homeostatic states}, \\ \mathcal{B}_{\mathfrak{R}}(H) \setminus H &= \text{nonhomeostatic but recoverable states}, \\ S \setminus \mathcal{B}_{\mathfrak{R}}(H) &= \text{states not recoverable to } H \text{ under the current policy}. \end{aligned}$$

Thus homeostasis is not represented by a binary distinction between "healthy" and "damaged" states.

Instead, the regenerative runtime induces a structured state-space geometry:

$$\boxed{H \subseteq \mathcal{B}_{\mathfrak{R}}(H) \subseteq S.}$$

The homeostatic region identifies the organization currently being preserved.

The recovery basin identifies the states from which that organization can still be regenerated.

The complement identifies states beyond the recovery capability of the current runtime policy.

This geometric view provides the basis for distinguishing adaptive states, repair states, and failure states in the following subsections.

5.3 Adaptive States

The preceding section characterized homeostasis relative to a designated homeostatic region

$$H_s = V \cap [s]_\eta.$$

A regenerative runtime, however, need not respond to every perturbation by returning to the same homeostatic region. Some perturbations may induce a transition to a different viable organization that can itself be maintained under subsequent runtime evolution.

We formalize such transitions as adaptation.

Definition 5.19 (Adaptive State)**.** Let

$$H_s = V \cap [s]_\eta$$

be a designated homeostatic region.

A state

$$x \in V$$

is *adaptive relative to* H_s when

$$x \notin H_s$$

and x belongs to some viable runtime region

$$A \subseteq V$$

that is invariant under the runtime policy.

Thus an adaptive state is viable but does not realize the original homeostatic signature.

In particular,

$$\eta(x) \neq \eta(s).$$

Adaptation therefore differs from semantic repair.

A semantic repair returns the runtime to

$$V \cap [s]_\eta,$$

whereas adaptation permits the runtime to occupy a different viable semantic class.

Definition 5.20 (Adaptive Region)**.** An *adaptive region* relative to a homeostatic region H_s is a nonempty subset

$$A \subseteq V$$

such that

$$A \cap H_s = \varnothing$$

and A is invariant under the runtime policy.

If all states in A share a common semantic signature

$$\omega_A \in \Omega,$$

so that

$$\eta(x) = \omega_A$$

for every

$$x \in A,$$

then A is called a *semantically coherent adaptive region*.

A semantically coherent adaptive region represents an alternative viable organization of the runtime.

The original and adaptive regions may therefore satisfy

$$H_s \subseteq \eta^{-1}(\eta(s)),$$

while

$$A \subseteq \eta^{-1}(\omega_A), \qquad \omega_A \neq \eta(s).$$

Both regions lie within

$$V,$$

but they represent distinct semantic organizations.

Definition 5.21 (Adaptive Transition)**.** Let $H \subseteq V$ be a homeostatic region and let

$$A \subseteq V$$

be an adaptive region.

An *adaptive transition from H to A* is a finite runtime execution

$$x_0 \to x_1 \to \cdots \to x_m$$

such that

$$x_0 \in H, \qquad x_m \in A.$$

The transition is *viability-preserving* when

$$x_i \in V$$

for every

$$0 \leq i \leq m.$$

The distinction between repair and adaptation can now be expressed by their targets.
For a reference homeostatic region H_s,

$$\boxed{\begin{array}{rcll} \text{repair} & : & x \rightsquigarrow_{\mathfrak{R}} H_s, & \\ \text{adaptation} & : & x \rightsquigarrow_{\mathfrak{R}} A, & A \subseteq V, \quad A \cap H_s = \varnothing. \end{array}}$$

Repair restores a previously selected semantic organization.
Adaptation establishes or enters a different viable organization.

Definition 5.22 (Adaptive Reachability)**.** Let

$$H, A \subseteq V$$

be respectively a homeostatic region and an adaptive region.

The region A is *adaptively reachable from* H when there exist states

$$x \in H, \qquad y \in A$$

such that

$$x \rightsquigarrow_{\mathfrak{R}} y.$$

We write

$$H \rightsquigarrow_{\mathfrak{R}} A.$$

Adaptive reachability introduces transitions between viable organizational regimes.
This motivates a higher-level representation of runtime dynamics.

Definition 5.23 (Viable Regime Graph)**.** Let

$$\mathscr{V} = \{H_1, \ldots, H_q\}$$

be a collection of pairwise disjoint invariant viable regions of a regenerative runtime.

The *viable regime graph* is the directed graph

$$G^V_{\mathfrak{R}} = (\mathscr{V}, E_V)$$

with an edge

$$H_i \longrightarrow H_j$$

whenever

$$H_i \rightsquigarrow_{\mathfrak{R}} H_j.$$

The viable regime graph compresses state-level runtime dynamics into transitions among viable organizational regimes.

A single regime may contain many microscopic runtime states, while the graph records whether execution can move the system from one persistent viable organization to another.

Definition 5.24 (Reversible Adaptation)**.** An adaptive transition

$$H \rightsquigarrow_{\mathfrak{R}} A$$

is *reversible* when

$$A \rightsquigarrow_{\mathfrak{R}} H.$$

Otherwise, it is *runtime-irreversible relative to the current policy.*

Runtime irreversibility is policy-relative.

It does not imply that no algebraic transformation capable of returning the system exists. It means that the current runtime execution semantics provide no path from the adaptive region back to the original homeostatic region.

Thus one may have

$$A \not\rightsquigarrow_{\mathfrak{R}} H$$

even though the underlying transformation algebra contains a composite mapping states in A into H.

Definition 5.25 (Adaptive Basin)**.** Let

$$A \subseteq V$$

be an adaptive region.

Its *adaptive basin* is

$$\mathcal{B}_{\mathfrak{R}}(A) = \{x \in S \mid x \rightsquigarrow_{\mathfrak{R}} A\}.$$

The runtime may therefore possess several competing basins:

$$\mathcal{B}_{\mathfrak{R}}(H_1), \ldots, \mathcal{B}_{\mathfrak{R}}(H_q).$$

A perturbed state may lie in the basin of the original homeostatic region, in the basin of an adaptive region, in several basins when execution is nondeterministic, or outside the basin of every known viable regime.

For deterministic policies, the realized trajectory selects a particular long-term runtime behavior.

Definition 5.26 (Adaptive Commitment)**.** Let H be an original homeostatic region and A an adaptive region.

A state

$$x \in \mathcal{B}_{\mathfrak{R}}(A)$$

is *committed to A relative to H* when

$$x \notin \mathcal{B}_{\mathfrak{R}}(H).$$

Thus

$$x \in \mathcal{B}_{\mathfrak{R}}(A) \setminus \mathcal{B}_{\mathfrak{R}}(H).$$

Adaptive commitment marks a qualitative change in regenerative possibility.

Before commitment, restoration of the original homeostatic organization may remain reachable.

After commitment, the original region is no longer reachable under the current runtime policy, although another viable regime remains reachable.

This gives three distinct post-perturbation possibilities:

recovery	:	$x \rightsquigarrow_{\mathfrak{R}} H$,
adaptation	:	$x \rightsquigarrow_{\mathfrak{R}} A$ for an alternative viable region A,
failure	:	x reaches no designated viable regime.

The distinction is semantic rather than merely geometric.

Recovery preserves or restores the selected homeostatic signature.

Adaptation changes the selected viable organization while maintaining viability.

Failure represents the loss of both the original homeostatic organization and any accessible alternative viable organization.

Proposition 5.27 (Adaptive Regions Are Distinct from Homeostatic Recovery)**.** *Let*

$$H_s = V \cap [s]_\eta$$

and let $A \subseteq V$ be a semantically coherent adaptive region with signature

$$\omega_A \neq \eta(s).$$

Then

$$A \cap H_s = \varnothing.$$

Consequently, reaching A does not constitute successful semantic repair of the reference state s.

Proof. Suppose for contradiction that

$$x \in A \cap H_s.$$

Since

$$x \in A,$$

semantic coherence gives

$$\eta(x) = \omega_A.$$

Since

$$x \in H_s,$$

the definition of H_s gives

$$\eta(x) = \eta(s).$$

Hence

$$\omega_A = \eta(s),$$

contradicting the assumption

$$\omega_A \neq \eta(s).$$

Therefore,

$$A \cap H_s = \varnothing.$$

Since successful semantic repair requires return to

$$V \cap [s]_\eta = H_s,$$

arrival in A is adaptive rather than reparative. □

The regenerative runtime therefore supports more than a binary distinction between successful recovery and failure.

Its viable state space may contain multiple persistent semantic regimes, with runtime execution determining whether a perturbation produces restoration, adaptation, commitment to a new regime, or eventual failure.

This extends the geometry developed in the preceding section from

$$H \subseteq \mathcal{B}_{\mathfrak{R}}(H) \subseteq S$$

to a family of competing viable regions and their basins:

$$\boxed{\{H_i, \mathcal{B}_{\mathfrak{R}}(H_i)\}_{i \in I}\,.}$$

The next question is then not merely whether a state is homeostatic or adaptive, but whether the runtime is actively executing transformations directed toward restoration.

This motivates the notion of a repair state.

5.4 Repair States

The preceding sections distinguished homeostatic states from adaptive states. We now identify states associated specifically with the execution of regenerative recovery.

A state may lie outside a designated homeostatic region while remaining recoverable to that region. Recoverability alone, however, does not imply that the runtime is currently executing a recovery pathway.

We therefore distinguish recoverable states from repair states.

Definition 5.28 (Repair State)**.** Let

$$H \subseteq V$$

be a designated homeostatic region.

A state

$$x \in S \setminus H$$

is a *repair state relative to* H when:

1. $$x \in \mathcal{B}_{\mathfrak{R}}(H),$$

   so that H remains runtime-reachable from x; and

2. the transition selected by the runtime policy at x belongs to the repair family:

   $$\Pi(x, c) \in \mathcal{R}$$

   for the current runtime context c.

Thus a repair state is not merely damaged and recoverable. It is a recoverable state from which the runtime is actively executing a repair transformation.

Let

$$\mathrm{RS}_{\mathfrak{R}}(H)$$

denote the set of repair states relative to H.

Then

$$\mathrm{RS}_{\mathfrak{R}}(H) \subseteq \mathcal{B}_{\mathfrak{R}}(H) \setminus H.$$

The inclusion may be strict.

A state may remain recoverable while the runtime executes an ordinary, adaptive, compensatory, or otherwise non-reparative transition.

Definition 5.29 (Repair Transition)**.** A runtime transition

$$x \xrightarrow{r} y$$

is a *repair transition relative to* H when

$$r \in \mathcal{R}$$

and

$$x \in \mathcal{B}_{\mathfrak{R}}(H).$$

It is a *successful terminal repair transition* when additionally

$$y \in H.$$

A repair transition therefore describes a single executed step of a recovery process.

It need not itself restore homeostasis. Multiple repair transitions may be required before the runtime returns to H.

Definition 5.30 (Repair Episode)**.** A *repair episode relative to* H is a finite runtime segment

$$x_0 \xrightarrow{r_0} x_1 \xrightarrow{r_1} \cdots \xrightarrow{r_{m-1}} x_m$$

such that

$$x_0 \notin H, \qquad x_m \in H,$$

and

$$r_i \in \mathcal{R}$$

for every

$$0 \leq i < m.$$

The *length* of the repair episode is

$$m.$$

Repair episodes are the runtime counterparts of repair pathways in the underlying algebra.

The distinction is that an algebraic repair pathway represents an available composition of primitive repair operations, whereas a repair episode records a pathway actually selected and executed by the runtime policy.

Proposition 5.31 (Repair Episodes Induce Successful Algebraic Repairs)**.** *Let*

$$H_s = V \cap [s]_\eta$$

for some viable reference state

$$s \in V.$$

Suppose

$$d(s) = x_0 \xrightarrow{r_0} x_1 \xrightarrow{r_1} \cdots \xrightarrow{r_{m-1}} x_m$$

is a repair episode with

$$x_m \in H_s.$$

Then the composite

$$r = r_{m-1} \circ \cdots \circ r_0$$

is a successful repair of d *at* s*.*

Proof. Because each

$$r_i \in \mathcal{R}$$

and the repair family is closed under composition,

$$r = r_{m-1} \circ \cdots \circ r_0 \in \mathcal{R}.$$

The repair episode gives

$$r(d(s)) = x_m.$$

Since

$$x_m \in H_s = V \cap [s]_\eta,$$

we have

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s).$$

Therefore r is a successful repair of d at s. □

The length of an executed repair episode need not equal the algebraic repair depth.

Definition 5.32 (Runtime Repair Length)**.** Let

$$s \in V, \qquad d \in \mathcal{D}.$$

If the runtime policy generates a repair episode

$$d(s) = x_0 \to x_1 \to \cdots \to x_m \in H_s,$$

define its *runtime repair length* by

$$\lambda_\Pi(s, d) = m.$$

If no repair episode generated by the policy reaches H_s, define

$$\lambda_\Pi(s, d) = \infty.$$

When the runtime episode uses primitive repair operations from the same basis used to define repair depth, algebraic optimality gives a lower bound.

Proposition 5.33 (Repair-Depth Lower Bound)**.** *Suppose the runtime repair episode for* (s, d) *consists entirely of primitive repair operations from*

$$\mathcal{Q}.$$

If

$$\lambda_\Pi(s, d) < \infty,$$

then

$$\delta_\mathcal{A}(s, d) \leq \lambda_\Pi(s, d).$$

Proof. The executed repair episode determines a successful primitive repair word of length

$$\lambda_\Pi(s, d).$$

By definition,

$$\delta_\mathcal{A}(s, d)$$

is the minimum length among all successful primitive repair words.

Therefore,

$$\delta_\mathcal{A}(s, d) \leq \lambda_\Pi(s, d).$$

□

The difference

$$\lambda_\Pi(s, d) - \delta_\mathcal{A}(s, d)$$

therefore measures the excess number of primitive repair steps executed by the runtime relative to an algebraically shortest successful pathway.

Definition 5.34 (Repair Overhead)**.** For a recoverable pair

$$(s, d)$$

with finite runtime repair length, define the *repair overhead* of the policy by

$$\mathrm{OH}_\Pi(s, d) = \lambda_\Pi(s, d) - \delta_{\mathcal{A}}(s, d).$$

By the preceding proposition,

$$\mathrm{OH}_\Pi(s, d) \geq 0.$$

A policy has zero repair overhead at (s, d) precisely when its executed primitive repair episode has minimum possible length:

$$\lambda_\Pi(s, d) = \delta_{\mathcal{A}}(s, d).$$

Definition 5.35 (Repair-Optimal Policy)**.** A runtime policy Π is *repair-optimal at* (s, d) when

$$\lambda_\Pi(s, d) = \delta_{\mathcal{A}}(s, d).$$

It is *globally repair-optimal* on a class

$$\mathcal{X} \subseteq V \times \mathcal{D}$$

when it is repair-optimal for every recoverable pair

$$(s, d) \in \mathcal{X}.$$

Repair optimality therefore connects the algebraic optimization problem developed earlier with runtime execution.

The repair algebra determines

$$\delta_{\mathcal{A}}(s, d),$$

the best achievable primitive repair depth.

The runtime policy determines

$$\lambda_\Pi(s, d),$$

the repair length actually realized.

Thus

$$\boxed{\delta_{\mathcal{A}}(s, d) \leq \lambda_\Pi(s, d).}$$

The gap between them measures execution inefficiency relative to the available repair structure.

This distinction becomes especially important in light of the complexity results established above. Computing a shortest repair pathway may be NP-hard even when verifying an individual pathway is efficient.

A runtime policy may therefore face a tradeoff between the computational cost of selecting an optimal repair and the execution cost of following a nonminimal repair pathway.

Definition 5.36 (Repair Progress)**.** Let

$$x \in \mathcal{B}_{\mathfrak{R}}(H).$$

Define the *remaining repair distance* to H by

$$D_H(x) = \inf \left\{ m \geq 0 \mid x \rightsquigarrow_{\mathfrak{R}}^{m} H \right\},$$

where

$$x \rightsquigarrow_{\mathfrak{R}}^{m} H$$

means that H can be reached from x in exactly m runtime transitions.

A transition

$$x \to y$$

makes *strict repair progress* when

$$D_H(y) < D_H(x).$$

The quantity

$$D_H(x)$$

is a runtime reachability distance. It depends on the admissible transition structure and policy semantics rather than solely on the underlying repair monoid.

Proposition 5.37 (Finite Recovery under Strict Repair Progress)**.** *Let*

$$x_0$$

be a runtime state with

$$D_H(x_0) < \infty.$$

Suppose every transition executed during a repair episode satisfies

$$D_H(x_{t+1}) < D_H(x_t)$$

whenever

$$x_t \notin H.$$

Then the runtime reaches H after finitely many transitions.

Proof. The values

$$D_H(x_t)$$

are nonnegative integers.

By assumption, each transition outside H strictly decreases this integer.

Hence an infinite strictly decreasing sequence

$$D_H(x_0) > D_H(x_1) > D_H(x_2) > \cdots$$

is impossible.

Therefore, after finitely many transitions,

$$D_H(x_t) = 0.$$

By definition of the runtime distance,

$$D_H(x_t) = 0$$

implies

$$x_t \in H.$$

Thus the repair episode reaches the homeostatic region in finite time. □

This gives a simple termination principle for regenerative execution:

strict descent of a well-founded repair measure $\Longrightarrow$ finite recovery.

Repair states can therefore be characterized not only by the type of transformation currently being executed, but also by whether execution makes measurable progress toward the target homeostatic region.

The runtime state space now contains several mathematically distinct classes:

H	homeostatic states,
A	adaptive states,
$\mathrm{RS}_{\mathfrak{R}}(H)$	actively repairing states,
$\mathcal{B}_{\mathfrak{R}}(H) \setminus H$	recoverable nonhomeostatic states.

The remaining question is what occurs when recovery is no longer reachable, when execution ceases to make progress, or when the runtime leaves every accessible viable regime.

These cases define runtime failure.

5.5 Failure States

The preceding sections distinguished homeostatic, adaptive, and repair states. We now formalize runtime failure.

Failure is not identified with a single subset of the state space. A runtime may fail because viability has been lost, because a designated homeostatic region is no longer reachable, because an available algebraic repair is not realized by the runtime policy, or because an executing repair process fails to converge toward recovery.

We therefore distinguish several forms of failure.

Definition 5.38 (Viability Failure)**.** A runtime state

$$x \in S$$

is in *viability failure* when

$$x \notin V.$$

The viability-failure region is

$$F_V = S \setminus V.$$

Viability failure is an instantaneous state property. It does not by itself imply irrecoverability.

Indeed, a state

$$x \notin V$$

may nevertheless satisfy

$$x \in \mathcal{B}_{\mathfrak{R}}(H)$$

for some homeostatic region

$$H \subseteq V.$$

Such a state is nonviable but regeneratively recoverable.

This motivates a stronger notion.

Definition 5.39 (Recovery Failure)**.** Let

$$H \subseteq V$$

be a designated homeostatic region.

A state

$$x \in S$$

is in *recovery failure relative to* H when

$$x \notin \mathcal{B}_{\mathfrak{R}}(H).$$

Equivalently,

$$\mathrm{Reach}_{\mathfrak{R}}(x) \cap H = \varnothing.$$

The recovery-failure region relative to H is

$$F_H = S \setminus \mathcal{B}_{\mathfrak{R}}(H).$$

Recovery failure is therefore stronger than displacement from homeostasis.

A state may satisfy

$$x \notin H$$

while still belonging to

$$\mathcal{B}_{\mathfrak{R}}(H).$$

Such a state is displaced but recoverable.

By contrast,

$$x \in F_H$$

means that the current runtime semantics provide no finite execution returning the system to H.

Viability failure and recovery failure are logically distinct.

One may have

$$x \notin V \qquad \text{and} \qquad x \in \mathcal{B}_{\mathfrak{R}}(H),$$

representing temporary nonviability followed by possible regeneration.

Conversely, one may have

$$x \in V \qquad \text{and} \qquad x \notin \mathcal{B}_{\mathfrak{R}}(H),$$

representing a viable state from which the original homeostatic organization cannot be recovered under the current runtime policy.

The second case may arise, for example, after adaptive commitment to another viable regime.

Definition 5.40 (Global Regenerative Failure)**.** Let

$$\mathscr{V} = \{H_i\}_{i \in I}$$

be the designated family of viable runtime regimes.

A state

$$x \in S$$

is in *global regenerative failure* when

$$x \notin \bigcup_{i \in I} \mathcal{B}_{\mathfrak{R}}(H_i).$$

Define

$$F_{\mathrm{reg}} = S \setminus \bigcup_{i \in I} \mathcal{B}_{\mathfrak{R}}(H_i).$$

A state in

$$F_{\text{reg}}$$

cannot reach any designated viable regime under the current runtime semantics.

This distinguishes failure relative to one homeostatic target from failure of the regenerative system as a whole.

A state may satisfy

$$x \notin \mathcal{B}_{\mathfrak{R}}(H_s)$$

while still satisfying

$$x \in \mathcal{B}_{\mathfrak{R}}(A)$$

for some alternative adaptive region A.

Such a state has lost recoverability to the original homeostatic organization without undergoing global regenerative failure.

Proposition 5.41 (Failure Decomposition)**.** *Let*

$$\mathscr{V} = \{H_i\}_{i \in I}$$

be the designated viable regimes of a regenerative runtime.

Then

$$F_{\text{reg}} = \bigcap_{i \in I} (S \setminus \mathcal{B}_{\mathfrak{R}}(H_i)) .$$

Proof. By definition,

$$F_{\text{reg}} = S \setminus \bigcup_{i \in I} \mathcal{B}_{\mathfrak{R}}(H_i).$$

Applying De Morgan's law gives

$$S \setminus \bigcup_{i \in I} \mathcal{B}_{\mathfrak{R}}(H_i) = \bigcap_{i \in I} (S \setminus \mathcal{B}_{\mathfrak{R}}(H_i)) .$$

Therefore,

$$F_{\text{reg}} = \bigcap_{i \in I} (S \setminus \mathcal{B}_{\mathfrak{R}}(H_i)) .$$

□

The preceding notions describe failure relative to runtime reachability. A different form of failure occurs when the underlying repair algebra contains a successful repair but the runtime policy fails to realize it.

Definition 5.42 (Policy-Induced Repair Failure)**.** Let

$$s \in V, \qquad d \in \mathcal{D},$$

and suppose

$$(s, d)$$

is recoverable in the underlying repair algebra:

$$\text{Rep}_{\mathcal{A}}(s, d) \neq \varnothing.$$

The pair (s, d) exhibits *policy-induced repair failure* when

$$d(s) \notin \mathcal{B}_{\mathfrak{R}}(H_s),$$

where

$$H_s = V \cap [s]_\eta.$$

Policy-induced repair failure means that recovery is structurally available but operationally inaccessible.

Thus

$$\boxed{\text{algebraic recoverability} \not\Rightarrow \text{runtime recoverability.}}$$

This is not a failure of repair capacity.

It is a failure of repair execution.

Proposition 5.43 (Policy Failure Criterion)**.** *Let*

$$s \in V, \qquad d \in \mathcal{D},$$

and assume

$$\mathrm{Rep}_{\mathcal{A}}(s, d) \neq \varnothing.$$

If every policy-generated execution beginning at

$$d(s)$$

avoids

$$H_s,$$

then (s, d) *exhibits policy-induced repair failure.*

Proof. If every policy-generated execution beginning at

$$d(s)$$

avoids H_s, then no state in H_s is runtime-reachable from $d(s)$.

Hence

$$d(s) \notin \mathcal{B}_{\mathfrak{R}}(H_s).$$

Since algebraic recoverability was assumed, the definition of policy-induced repair failure applies. □

A runtime may also begin a repair process without successfully completing it.

Definition 5.44 (Repair Stagnation)**.** Let

$$H \subseteq V$$

and let

$$x_0 \to x_1 \to x_2 \to \cdots$$

be a runtime execution beginning in the recovery basin

$$x_0 \in \mathcal{B}_{\mathfrak{R}}(H).$$

The execution exhibits *repair stagnation* when there exists a time

$$t_0$$

such that

$$x_t \notin H$$

for all

$$t \geq t_0$$

and the remaining repair distance fails to decrease infinitely often.

In particular, a persistent region in which

$$D_H(x_{t+1}) \geq D_H(x_t)$$

for all sufficiently large t constitutes repair stagnation.

Repair stagnation captures failure of progress rather than immediate loss of recoverability.

The runtime may remain inside

$$\mathcal{B}_{\mathfrak{R}}(H)$$

while repeatedly selecting transitions that do not bring it closer to H.

A stronger failure occurs when execution leaves the recovery basin entirely.

Definition 5.45 (Recovery Escape)**.** Let

$$H \subseteq V.$$

A runtime transition

$$x \to y$$

is a *recovery escape relative to* H when

$$x \in \mathcal{B}_{\mathfrak{R}}(H)$$

but

$$y \notin \mathcal{B}_{\mathfrak{R}}(H).$$

A recovery escape crosses the boundary between states from which restoration remains possible and states from which restoration is no longer reachable under the runtime semantics.

This boundary is therefore dynamically significant.

Define the *recovery boundary* by

$$\partial_{\mathfrak{R}} H = \{x \in \mathcal{B}_{\mathfrak{R}}(H) \mid \exists\, x \to y \text{ with } y \notin \mathcal{B}_{\mathfrak{R}}(H)\}\,.$$

States in

$$\partial_{\mathfrak{R}} H$$

are recoverable states from which at least one admissible transition can destroy recoverability.

The existence of such states shows that recoverability may itself be a runtime resource that must be preserved.

Definition 5.46 (Recovery-Preserving Policy)**.** A runtime policy Π is *recovery-preserving relative to* H when

$$x \in \mathcal{B}_{\mathfrak{R}}(H)$$

implies

$$\Pi(x, c)(x) \in \mathcal{B}_{\mathfrak{R}}(H)$$

whenever the policy is defined.

Thus the recovery basin is invariant under a recovery-preserving policy.

Proposition 5.47 (Preservation of Recoverability)**.** *If* Π *is recovery-preserving relative to* H*, then every policy-generated execution beginning in*

$$\mathcal{B}_{\mathfrak{R}}(H)$$

remains in

$$\mathcal{B}_{\mathfrak{R}}(H)$$

for as long as the execution is defined.

Proof. The result follows by induction.

Let

$$x_0 \in \mathcal{B}_{\mathfrak{R}}(H).$$

Suppose

$$x_t \in \mathcal{B}_{\mathfrak{R}}(H).$$

Because Π is recovery-preserving,

$$x_{t+1} = \Pi(x_t, c_t)(x_t) \in \mathcal{B}_{\mathfrak{R}}(H).$$

Hence every subsequent state remains inside the recovery basin. □

Recovery preservation does not itself guarantee recovery.

A policy may remain indefinitely inside the recovery basin without reaching the target region.

Combining recovery preservation with strict repair progress yields a stronger result.

Theorem 5.48 (Sufficient Conditions for Finite Regeneration)**.** *Let*

$$H \subseteq V$$

be a homeostatic region, and let

$$x_0 \in \mathcal{B}_{\mathfrak{R}}(H).$$

Suppose that throughout the policy-generated execution beginning at x_0*:*

1. *the policy is recovery-preserving relative to* H*; and*
2. *whenever*

$$x_t \notin H,$$

the selected transition satisfies

$$D_H(x_{t+1}) < D_H(x_t).$$

Then the runtime reaches

$$H$$

after finitely many transitions.

Proof. Recovery preservation ensures that every state along the execution remains in

$$\mathcal{B}_{\mathfrak{R}}(H),$$

so

$$D_H(x_t) < \infty$$

throughout the execution.

Whenever

$$x_t \notin H,$$

the second assumption gives

$$D_H(x_{t+1}) < D_H(x_t).$$

Thus

$$D_H(x_t)$$

forms a strictly decreasing sequence of nonnegative integers until it reaches zero.

Such a sequence must terminate after finitely many steps.

Therefore there exists

$$T < \infty$$

such that

$$D_H(x_T) = 0.$$

By definition,

$$x_T \in H.$$

Hence the runtime regenerates to the homeostatic region in finite time. □

The theorem separates two requirements for successful regenerative execution:

preserve the possibility of recovery
+
make well-founded progress toward recovery
⇓
finite regeneration.

This provides a runtime interpretation of failure that is richer than nonviability alone.

Relative to a target homeostatic region H, a state or execution may exhibit:

1. *viability failure*:
$$x \notin V;$$
2. *recovery failure*:
$$x \notin \mathcal{B}_{\mathfrak{R}}(H);$$
3. *policy-induced repair failure*: an algebraic repair exists but is not runtime-reachable;
4. *repair stagnation*: recoverability persists but execution fails to make sustained progress;
5. *recovery escape*: execution crosses from the recovery basin into its complement;
6. *global regenerative failure*: no designated viable regime remains reachable.

The runtime semantics therefore distinguish loss of current viability from loss of future regenerative possibility.

This distinction is essential: a runtime may temporarily cease to be viable while retaining a path to recovery, whereas another runtime may remain currently viable while having irreversibly lost access to its original homeostatic organization.

The final subsection develops the pathways by which recovery is actually realized.

5.6 Recovery Paths

The preceding sections classified runtime states according to their relation to homeostasis, adaptation, repair, and failure. We now study the finite trajectories by which a regenerative runtime returns from a perturbed state to a designated homeostatic region.

These trajectories are called recovery paths.

Definition 5.49 (Recovery Path)**.** Let

$$H \subseteq V$$

be a designated homeostatic region and let

$$x \in S.$$

A *recovery path from x to H* is a finite runtime execution

$$\gamma : \qquad x = x_0 \xrightarrow{\tau_0} x_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{m-1}} x_m$$

such that

$$x_m \in H.$$

The *length* of the recovery path is

$$|\gamma| = m.$$

A recovery path may contain transitions of several kinds.

In particular, it need not consist entirely of repair transformations. Ordinary evolution, compensation, adaptation, or other admissible runtime transitions may occur before homeostasis is restored.

Thus recovery paths generalize repair episodes.

Every repair episode terminating in H is a recovery path to H, but a recovery path need not be a repair episode.

Definition 5.50 (Recovery Path Set)**.** For

$$x \in S$$

and

$$H \subseteq V,$$

define

$$\mathrm{Path}_{\mathfrak{R}}(x, H)$$

to be the set of all finite recovery paths from x to H.

Then

$$x \in \mathcal{B}_{\mathfrak{R}}(H)$$

if and only if

$$\mathrm{Path}_{\mathfrak{R}}(x, H) \neq \varnothing.$$

The recovery basin may therefore be characterized entirely in terms of recovery paths:

$$\mathcal{B}_{\mathfrak{R}}(H) = \{x \in S \mid \mathrm{Path}_{\mathfrak{R}}(x, H) \neq \varnothing\}.$$

Definition 5.51 (Shortest Recovery Length)**.** For

$$x \in \mathcal{B}_{\mathfrak{R}}(H),$$

define the *shortest recovery length* by

$$L_H(x) = \min\{|\gamma| \mid \gamma \in \mathrm{Path}_{\mathfrak{R}}(x, H)\}.$$

If

$$x \notin \mathcal{B}_{\mathfrak{R}}(H),$$

define

$$L_H(x) = \infty.$$

For the discrete runtime semantics considered here,

$$L_H(x) = D_H(x),$$

where D_H is the remaining repair distance introduced above.

We retain the recovery-path notation because it emphasizes the trajectory realizing the distance rather than only its numerical value.

Definition 5.52 (Minimal Recovery Path)**.** A recovery path

$$\gamma \in \mathrm{Path}_{\mathfrak{R}}(x, H)$$

is *minimal* when

$$|\gamma| = L_H(x).$$

Minimality here is runtime-relative.

A minimal recovery path is shortest among the transitions admissible under the runtime semantics. It need not coincide with a shortest repair word in the underlying repair algebra.

This produces three distinct notions:

$\delta_{\mathcal{A}}(s, d)$	$=$	minimum primitive algebraic repair depth,
$L_{H_s}(d(s))$	$=$	minimum admissible runtime recovery length,
$T_{H_s}(d(s))$	$=$	recovery time of the realized runtime execution.

The three quantities measure different levels of the regenerative system.

The first concerns algebraic capability.

The second concerns runtime admissibility.

The third concerns actual execution.

Proposition 5.53 (Runtime Constraint Inequality)**.** *Let*

$$s \in V, \qquad d \in \mathcal{D},$$

and suppose every admissible runtime recovery path from $d(s)$ to H_s consists of primitive repair operations from the basis

$$\mathcal{Q}.$$

Then

$$\delta_{\mathcal{A}}(s, d) \leq L_{H_s}(d(s)).$$

Proof. Every runtime recovery path under the stated assumption determines a successful primitive repair word in the underlying repair algebra.

Since

$$\delta_{\mathcal{A}}(s, d)$$

is the minimum length among all successful primitive repair words, it cannot exceed the length of a shortest runtime-admissible one.

Therefore,

$$\delta_{\mathcal{A}}(s, d) \leq L_{H_s}(d(s)).$$

□

When the policy realizes a finite recovery trajectory, one also has

$$L_{H_s}(d(s)) \leq T_{H_s}(d(s)).$$

Hence, under the primitive-repair hypothesis,

$$\boxed{\delta_{\mathcal{A}}(s, d) \leq L_{H_s}(d(s)) \leq T_{H_s}(d(s)).}$$

This chain separates three possible sources of regenerative inefficiency:

$$\text{algebraic limitation}, \qquad \text{runtime restriction}, \qquad \text{policy execution}.$$

Definition 5.54 (Runtime Constraint Gap)**.** For a recoverable pair

$$(s, d),$$

define the *runtime constraint gap* by

$$G_{\mathrm{rt}}(s, d) = L_{H_s}(d(s)) - \delta_{\mathcal{A}}(s, d),$$

whenever both quantities are finite.

The runtime constraint gap measures the additional path length imposed by runtime admissibility relative to the unrestricted primitive repair algebra.

Definition 5.55 (Policy Realization Gap)**.** For a policy-generated execution that reaches H_s, define the *policy realization gap* by

$$G_{\Pi}(s, d) = T_{H_s}(d(s)) - L_{H_s}(d(s)).$$

Thus the total excess recovery length decomposes as

$$T_{H_s}(d(s)) - \delta_{\mathcal{A}}(s, d) = G_{\mathrm{rt}}(s, d) + G_{\Pi}(s, d).$$

The first term measures the cost of runtime constraints.
The second measures the cost of policy selection within those constraints.
This decomposition refines the repair-overhead notion introduced above.

Definition 5.56 (Viability-Preserving Recovery Path)**.** A recovery path

$$\gamma : x_0 \to x_1 \to \cdots \to x_m$$

is *viability-preserving* when

$$x_i \in V$$

for every

$$0 \le i \le m.$$

A recovery path that is not viability-preserving may temporarily pass through states outside V before returning to the homeostatic region.

This motivates a measure of excursion severity.

Definition 5.57 (Nonviable Exposure)**.** Let

$$\gamma : x_0 \to x_1 \to \cdots \to x_m$$

be a recovery path.

Define its *nonviable exposure* by

$$E_V(\gamma) = |\{i \in \{0, \dots, m\} \mid x_i \notin V\}|.$$

Thus

$$E_V(\gamma) = 0$$

if and only if the path is viability-preserving.

Path length and nonviable exposure represent different recovery objectives.

A shortest recovery path need not minimize exposure outside the viability region.

Accordingly, regenerative path selection may be inherently multi-objective.

Definition 5.58 (Recovery Cost Functional)**.** Let

$$c : S \times \mathcal{T} \times S \to \mathbb{R}_{\ge 0}$$

assign a nonnegative cost to each admissible runtime transition.

For a recovery path

$$\gamma : x_0 \xrightarrow{\tau_0} x_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{m-1}} x_m,$$

define

$$C(\gamma) = \sum_{i=0}^{m-1} c(x_i, \tau_i, x_{i+1}).$$

The cost functional may encode transition count, energetic expenditure, resource consumption, time, risk, or another runtime-relevant quantity.

Minimum repair depth is recovered as a special case when every primitive repair transition has unit cost.

Definition 5.59 (Optimal Recovery Path)**.** A recovery path

$$\gamma^\star \in \mathrm{Path}_{\mathfrak{R}}(x, H)$$

is *cost-optimal* when

$$C(\gamma^\star) = \min \{C(\gamma) \mid \gamma \in \mathrm{Path}_{\mathfrak{R}}(x, H)\}.$$

Shortest recovery and minimum-cost recovery therefore need not coincide.

The runtime may prefer a longer pathway when it reduces some other relevant cost.

Definition 5.60 (Recovery Path Dominance)**.** Let

$$\gamma_1, \gamma_2 \in \mathrm{Path}_{\mathfrak{R}}(x, H).$$

The path γ_1 *dominates* γ_2 with respect to length, cost, and nonviable exposure when

$$|\gamma_1| \leq |\gamma_2|,$$

$$C(\gamma_1) \leq C(\gamma_2),$$

and

$$E_V(\gamma_1) \leq E_V(\gamma_2),$$

with at least one inequality strict.

A recovery path is *Pareto-optimal* when it is not dominated by another recovery path.

Thus a regenerative runtime may possess a family of incomparable optimal recovery strategies rather than a unique preferred pathway.

Definition 5.61 (Recovery Path Graph)**.** Fix a homeostatic region

$$H \subseteq V.$$

The *recovery path graph* is the directed graph

$$G_{\mathrm{rec}}(H) = (V_{\mathrm{rec}}, E_{\mathrm{rec}})$$

whose vertex set is

$$V_{\mathrm{rec}} = \mathcal{B}_{\mathfrak{R}}(H),$$

and whose directed edges are the admissible runtime transitions

$$x \longrightarrow y$$

between states in the recovery basin.

The recovery path graph exposes the internal geometry of regenerative possibility.

Homeostatic states form target vertices.

Repair states identify vertices at which repair transformations are currently selected.

Adaptive states may form alternative viable regions.

Recovery-boundary states identify locations from which an admissible transition may leave the basin.

Failure states lie outside the graph.

The regenerative problem can therefore be represented as navigation through a structured directed state space.

Proposition 5.62 (Shortest-Path Characterization)**.** *Suppose the recovery path graph*

$$G_{\mathrm{rec}}(H)$$

is finite.

Then

$$L_H(x)$$

is the directed graph distance from

$$x$$

to the vertex set

$$H.$$

Proof. By definition, a directed path in

$$G_{\mathrm{rec}}(H)$$

from x to a vertex of H is exactly a finite admissible runtime execution from x to H.

The minimum number of edges among such paths is therefore exactly

$$L_H(x).$$

□

Similarly, when transition costs are nonnegative, cost-optimal recovery is a weighted shortest-path problem on the recovery graph whenever that graph is given explicitly.

This observation does not contradict the NP-hardness of Bounded Repair Depth established earlier.

The complexity depends critically on representation.

An explicitly enumerated finite recovery graph permits ordinary graph-search methods, while a compactly represented repair system may implicitly encode an exponentially large state or pathway space.

Thus the distinction between explicit and compact representations remains fundamental at the runtime level.

Theorem 5.63 (Recovery Path Decomposition)**.** *Let*

$$\gamma : x_0 \xrightarrow{\tau_0} x_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{m-1}} x_m$$

be a recovery path to

$$H.$$

Partition its transitions according to runtime type:

$$I_{\mathrm{ord}}, \quad I_{\mathrm{dam}}, \quad I_{\mathrm{rep}}, \quad I_{\mathrm{ad}}, \quad I_{\mathrm{comp}}.$$

Then

$$m = |I_{\mathrm{ord}}| + |I_{\mathrm{dam}}| + |I_{\mathrm{rep}}| + |I_{\mathrm{ad}}| + |I_{\mathrm{comp}}|$$

provided each executed transition is assigned to exactly one runtime class.

Proof. Under the stated assumption, the five index sets form a partition of

$$\{0, \ldots, m-1\}.$$

The cardinality of their disjoint union is therefore

$$m.$$

Hence

$$m = |I_{\mathrm{ord}}| + |I_{\mathrm{dam}}| + |I_{\mathrm{rep}}| + |I_{\mathrm{ad}}| + |I_{\mathrm{comp}}|.$$

□

This elementary decomposition is useful because recovery need not be purely reparative.

A runtime may compensate before repairing, adapt while preserving viability, undergo additional perturbation, or pass through ordinary transitions before the target homeostatic region is restored.

Recovery is therefore a property of the entire executed trajectory rather than merely of individual repair operations.

The regenerative runtime now admits three complementary levels of analysis:

state level	:	homeostatic, adaptive, repair, and failure states,
path level	:	recovery trajectories and their costs,
algebraic level	:	repair transformations, composition, and minimal depth.

The purpose of the runtime semantics is to connect these levels.

A regenerative system does not merely possess repair transformations.

It occupies states, selects transformations, executes trajectories, preserves or changes viable organization, and may either return to homeostasis or lose the possibility of recovery.

This completes the state-and-path semantics of regenerative runtimes and provides the basis for studying the transformations that govern runtime evolution.

6 Runtime Transformations

6.1 State Transitions

A regenerative runtime evolves through transitions between biological runtime states. These transitions provide the operational layer connecting the algebraic transformations of a repair algebra with the temporal behavior of an executing system.

Let

$$S$$

be the runtime state space.

A runtime transition is represented by the relation induced by the admissible transformation class $\mathcal{T}$:

$$\longrightarrow_{\mathcal{T}} \subseteq S \times S.$$

When no ambiguity can arise, we abbreviate

$$s \longrightarrow_{\mathcal{T}} t$$

by

$$s \longrightarrow t.$$

The transition relation may represent ordinary biological evolution, perturbation, damage, repair, compensation, adaptation, or other admissible changes of state. The interpretation of a transition depends on the regions of state space involved and on the transformation responsible for the transition.

Definition 6.1 (Runtime Transition Relation)**.** Let

$$\mathcal{T}$$

be the collection of admissible runtime transformations of a regenerative runtime. The *runtime transition relation* induced by $\mathcal{T}$ is

$$\longrightarrow_{\mathcal{T}} \subseteq S \times S,$$

defined by

$$s \longrightarrow_{\mathcal{T}} t$$

if and only if there exists

$$\tau \in \mathcal{T}$$

such that

$$\tau(s) = t.$$

For a regenerative runtime

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi),$$

the transition structure $\mathcal{T}$ determines which changes of state may occur during execution, while the policy Π determines which admissible transitions are selected or permitted under the current runtime conditions.

A transition may be induced by an explicit transformation.

Let

$$\tau : S \rightharpoonup S$$

be a partial state transformation.

Whenever τ is defined at s, it induces the transition

$$s \longrightarrow \tau(s).$$

Thus the transformation specifies the state update, while the transition relation records its operational realization.

Definition 6.2 (Transformation-Induced Transition)**.** Let

$$\tau : S \rightharpoonup S.$$

A transition

$$s \longrightarrow t$$

is *induced by* τ when

$$t = \tau(s).$$

We write

$$s \xrightarrow{\tau} t$$

when the inducing transformation is relevant to the analysis.

This notation permits the runtime to distinguish transition classes.

For example, a damage transformation

$$d \in \mathcal{D}$$

induces

$$s \xrightarrow{d} d(s),$$

while a repair transformation

$$r \in \mathcal{R}$$

induces

$$x \xrightarrow{r} r(x).$$

More generally, admissible runtime transformations may be partitioned into classes

$$\mathcal{T} = \mathcal{T}_{\text{ord}} \cup \mathcal{T}_{\text{dam}} \cup \mathcal{T}_{\text{rep}} \cup \mathcal{T}_{\text{comp}} \cup \mathcal{T}_{\text{adapt}},$$

representing ordinary, damaging, reparative, compensatory, and adaptive transformations.

These classes need not be disjoint. A single transformation may have more than one interpretation relative to different semantic observables or regions of state space.

The runtime therefore classifies transitions not merely by physical change, but by their relationship to viability and semantic organization.

Let

$$V \subseteq S$$

be the viable region.

Definition 6.3 (Viability-Preserving Transition)**.** A transition

$$s \longrightarrow t$$

is *viability-preserving* when

$$s \in V \quad \Longrightarrow \quad t \in V.$$

For transitions considered only from viable source states, this reduces to

$$s \in V \quad \text{and} \quad t \in V.$$

Viability preservation is weaker than semantic preservation.

A transition may satisfy

$$s, t \in V$$

while

$$\eta(t) \neq \eta(s).$$

Such a transition preserves continued operation while changing the selected semantic organization.

Conversely, a transition is semantics-preserving when

$$\eta(t) = \eta(s).$$

Definition 6.4 (Semantics-Preserving Runtime Transition)**.** A runtime transition

$$s \longrightarrow t$$

is *semantics-preserving* relative to

$$\eta : S \to \Omega$$

when

$$\eta(t) = \eta(s).$$

A transition may therefore be classified independently according to viability and semantic preservation.

For a transition

$$s \longrightarrow t,$$

four basic possibilities arise:

	$t \in V$	$t \notin V$
$\eta(t) = \eta(s)$	viable semantic preservation	semantic preservation without viability
$\eta(t) \neq \eta(s)$	viable semantic change	nonviable semantic change

This separation is important for regenerative systems because damage, compensation, and regeneration need not occupy the same category.

A damaging transition may move the system outside its homeostatic region while leaving it viable. A compensatory transition may preserve viability without restoring the original semantic condition. A regenerative transition sequence may eventually restore the designated semantics without reproducing the original state.

Runtime behavior is therefore naturally represented by paths.

Definition 6.5 (Runtime State Trace)**.** Let

$$s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} s_n$$

be a finite runtime execution in the sense of the preceding section.

Its *runtime state trace* is

$$\pi = (s_0, s_1, \ldots, s_n).$$

The length of the trace is

$$|\pi| = n.$$

If the transitions are induced by transformations

$$\tau_1, \ldots, \tau_n,$$

the execution may be written

$$s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} s_n.$$

The execution is viability-preserving when

$$s_i \in V$$

for every

$$0 \leq i \leq n.$$

It is semantics-preserving relative to η when

$$\eta(s_i) = \eta(s_0)$$

for every

$$0 \leq i \leq n.$$

Proposition 6.6 (Local Viability Preservation)**.** *Let*

$$\pi = (s_0, \ldots, s_n)$$

be a runtime execution with

$$s_0 \in V.$$

If every transition

$$s_i \longrightarrow s_{i+1}$$

is viability-preserving, then

$$s_i \in V$$

for every

$$0 \leq i \leq n.$$

Proof. Proceed by induction.

The initial state satisfies

$$s_0 \in V.$$

Suppose

$$s_i \in V.$$

Since

$$s_i \longrightarrow s_{i+1}$$

is viability-preserving,

$$s_{i+1} \in V.$$

Therefore every state of the execution belongs to V. □

The analogous result holds for semantic preservation.

Proposition 6.7 (Local Semantic Preservation)**.** *Let*

$$\pi = (s_0, \ldots, s_n)$$

be a runtime execution.

If every transition satisfies

$$\eta(s_{i+1}) = \eta(s_i),$$

then

$$\eta(s_n) = \eta(s_0).$$

More strongly,

$$\eta(s_i) = \eta(s_0)$$

for every

$$0 \leq i \leq n.$$

Proof. Repeated application of the transition condition gives

$$\eta(s_0) = \eta(s_1) = \cdots = \eta(s_n).$$

□

These propositions characterize maintenance, but regeneration requires a more general transition pattern.

Let

$$H \subseteq V$$

be a designated homeostatic region.

A system may begin in

$$s_0 \in H,$$

undergo a perturbation producing

$$s_k \notin H,$$

and later reach

$$s_n \in H.$$

The corresponding trajectory has the general form

$$\boxed{\text{homeostasis} \longrightarrow \text{perturbation} \longrightarrow \text{displacement} \longrightarrow \text{recovery} \longrightarrow \text{homeostasis.}}$$

The individual transitions need not preserve homeostatic membership. Regeneration is instead a property of the resulting trajectory.

This distinction separates *maintenance* from *recovery*.

Maintenance requires the system to remain within a designated invariant region:

$$s_i \in H \qquad \text{for all } i.$$

Recovery permits departure from that region and requires subsequent return:

$$s_0 \in H, \qquad s_k \notin H, \qquad s_n \in H$$

for some

$$0 < k < n.$$

The transition system therefore provides the temporal structure in which repair-algebra operations become regenerative behavior.

Repair operations describe transformations that may restore a damaged state. Runtime transitions determine whether and how those transformations can actually occur during execution.

This distinction can be expressed as

$$\boxed{\text{repair algebra} = \text{structure of allowable repair transformations,}}$$

while

$$\boxed{\text{regenerative runtime} = \text{dynamics of those transformations through state space.}}$$

The runtime policy adds a further operational constraint.

Let

$$\Pi$$

denote the policy component of the regenerative runtime.

For each runtime configuration

$$(s, c) \in S \times \mathcal{C},$$

the deterministic policy selects an admissible transformation

$$\Pi(s, c) \in \mathcal{T}$$

whenever the policy is defined.

Definition 6.8 (Policy-Admissible Transition)**.** A transition

$$s \xrightarrow{\tau} t$$

is *policy-admissible in context* c when

$$\tau = \Pi(s, c)$$

and

$$t = \tau(s).$$

A finite runtime execution

$$s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{n-1}} s_n$$

relative to a context trajectory

$$c_0, c_1, \ldots, c_{n-1}$$

is therefore policy-admissible when

$$\tau_i = \Pi(s_i, c_i)$$

and

$$s_{i+1} = \tau_i(s_i)$$

for every

$$0 \leq i < n.$$

This distinction separates structural possibility from operational behavior.

A repair transformation may exist algebraically without being selected by the runtime policy. Consequently, the existence of a repair path does not by itself guarantee that the executing system will follow that path.

The runtime transition structure therefore introduces three progressively stronger questions:

> Is a state transformation algebraically available?
>
> Is the corresponding transition operationally admissible?
>
> Will the runtime policy select a trajectory that reaches recovery?

These distinctions become central when regeneration is treated as a property of complete executions rather than isolated repair operations.

The next subsection formalizes regeneration in precisely this sense: as return to a designated homeostatic or semantic region through an admissible runtime trajectory.

6.2 Regeneration

Regeneration is the runtime process by which a system that has been displaced from a designated homeostatic region returns to that region through an admissible sequence of state transitions. Unlike repair, which is represented algebraically by elements of the repair monoid $\mathcal{R}$, regeneration is a property of runtime execution and may involve repair, compensation, adaptation, or ordinary runtime transitions.

Definition 6.9 (Regenerative Execution)**.** Let $\mathfrak{R}$ be a regenerative runtime and let $H \subseteq V$ be a designated homeostatic region. A finite runtime execution

$$\pi : x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

is *regenerative with respect to* H if

$$x_0 \notin H, \qquad x_n \in H.$$

If every intermediate state remains viable,

$$x_i \in V \qquad (0 \le i \le n),$$

then π is a *viability-preserving regenerative execution.*

Thus regeneration concerns restoration of a designated runtime condition rather than restoration of the identical physical or computational state.

Definition 6.10 (Semantic Regeneration)**.** Let $\eta : S \to \Omega$ be the semantic-signature map and let $s \in V$ be a reference state, and define

$$H_s = V \cap [s]_\eta.$$

A regenerative execution terminating at x_n is *semantically regenerative relative to* s if

$$x_n \in V \qquad \text{and} \qquad \eta(x_n) = \eta(s).$$

Equivalently,

$$x_n \in H_s = V \cap [s]_\eta.$$

Semantic regeneration therefore does not require

$$x_n = s.$$

It requires restoration of the semantic invariant represented by η.

Definition 6.11 (Exact Regeneration)**.** A regenerative execution relative to a reference state s is *exact* if

$$x_n = s.$$

Every exact regeneration is semantic regeneration, whereas semantic regeneration need not be exact.

Proposition 6.12 (Exact Regeneration Implies Semantic Regeneration)**.** *If a regenerative execution is exact relative to s, then it is semantically regenerative relative to s.*

Proof. Exact regeneration gives $x_n = s$. Hence

$$\eta(x_n) = \eta(s).$$

Since $s \in H_s \subseteq V$, we also have $x_n \in V$. Therefore $x_n \in H_s$. □

This distinction is fundamental: regeneration preserves or restores what is semantically significant without requiring identity of realization.

Definition 6.13 (Regenerative Reachability)**.** For $x \in S$ and a homeostatic region H, write

$$x \rightsquigarrow_{\text{reg}} H$$

if there exists a finite admissible runtime execution from x to some $y \in H$.

The regenerative basin of H is therefore

$$\mathcal{B}_R(H) = \{x \in S : x \rightsquigarrow_{\text{reg}} H\}.$$

Consequently,

$$x \in \mathcal{B}_R(H) \iff x \rightsquigarrow_{\text{reg}} H.$$

Regeneration is thus equivalent to reachability of the designated homeostatic region.

Proposition 6.14 (Concatenation of Regenerative Reachability)**.** *Suppose*

$$x \rightsquigarrow y \qquad \textit{and} \qquad y \rightsquigarrow_{\text{reg}} H.$$

Then

$$x \rightsquigarrow_{\text{reg}} H.$$

Proof. Let π_1 be a finite admissible execution from x to y, and let π_2 be a finite admissible execution from y to some $z \in H$. Their concatenation is a finite admissible execution from x to $z \in H$. Hence

$$x \rightsquigarrow_{\text{reg}} H.$$

□

Corollary 6.15 (Backward Closure of the Regenerative Basin)**.** *If*

$$y \in \mathcal{B}_R(H)$$

and

$$x \rightsquigarrow y,$$

then

$$x \in \mathcal{B}_R(H).$$

Thus every state capable of reaching a recoverable state is itself recoverable, provided the connecting execution is admissible.

Definition 6.16 (Regenerative Transformation)**.** A runtime transformation τ is *regenerative toward H at x* if

$$x \notin H, \qquad \tau(x) \text{ is defined,}$$

and

$$D_H(\tau(x)) < D_H(x),$$

where D_H denotes the shortest admissible runtime distance to H.

A transformation is *globally regenerative on $U \subseteq \mathcal{B}_R(H) \setminus H$* if it is regenerative toward H at every state in U at which it is selected.

This makes regeneration locally detectable as strict progress toward the target region.

Theorem 6.17 (Finite Regeneration by Strict Descent)**.** *Let*

$$x_0, x_1, x_2, \ldots$$

be a runtime execution with $x_0 \in \mathcal{B}_R(H)$. Suppose that whenever $x_i \notin H$,

$$D_H(x_{i+1}) < D_H(x_i).$$

Then the execution reaches H after finitely many transitions.

Proof. Because $x_0 \in \mathcal{B}_R(H)$,

$$D_H(x_0) \in \mathbb{N}.$$

While the execution remains outside H, the sequence

$$D_H(x_0), D_H(x_1), D_H(x_2), \ldots$$

is strictly decreasing in $\mathbb{N}$. No infinite strictly decreasing sequence of natural numbers exists. Therefore there is some finite n such that

$$D_H(x_n) = 0.$$

By definition of runtime distance,

$$D_H(x_n) = 0 \iff x_n \in H.$$

Hence the execution regenerates in finitely many transitions. □

Corollary 6.18 (Regeneration-Time Bound)**.** *Under the hypotheses of the preceding theorem, if each transition decreases D_H by at least one, then regeneration occurs in at most*

$$D_H(x_0)$$

transitions.

Definition 6.19 (Regenerative Policy)**.** A runtime policy Π is *regenerative on $U \subseteq \mathcal{B}_R(H)$* if every policy-generated execution beginning at $x \in U$ reaches H in finitely many transitions.

It is *viability-preserving regenerative* if, in addition, every state encountered before return lies in V.

This separates two questions:

Can the runtime regenerate?

from

Does its policy actually cause it to regenerate?

The first is structural; the second is operational.

Proposition 6.20 (Structural Recoverability Does Not Imply Policy Regeneration)**.** *There may exist*

$$x \in \mathcal{B}_R(H)$$

such that the execution generated by Π from x never reaches H.

Proof. Membership $x \in \mathcal{B}_R(H)$ asserts the existence of at least one admissible recovery path from x to H. It does not require the policy Π to select transitions belonging to such a path. A policy may instead cycle, stagnate, or select transitions that leave the recovery basin. Therefore structural recoverability does not imply policy-induced regeneration. □

This distinction identifies policy failure as fundamentally different from absence of regenerative capacity.

Definition 6.21 (Regenerative Invariant)**.** Let $I : S \to X$ be an observable. The observable I is a *regenerative invariant relative to* H if the relevant regenerative executions preserve or restore a prescribed value of I.

For a reference state s, restoration means

$$I(x_n) = I(s)$$

at the completion of regeneration, even when

$$x_n \neq s.$$

The semantic signature η is the principal example:

$$\eta(x_n) = \eta(s).$$

Thus regeneration is not characterized by absence of change. It is characterized by the preservation or restoration of specified properties through change.

Theorem 6.22 (Semantic Invariance of Successful Regeneration)**.** *Let $s \in V$, let $d \in \mathcal{D}$, and suppose a runtime execution begins at $d(s)$ and regenerates into*

$$H_s = V \cap [s]_\eta.$$

If x_n is its terminal state, then

$$x_n \in V$$

and

$$\eta(x_n) = \eta(s).$$

Hence the execution restores the semantic invariant of s.

Proof. By regeneration into H_s,

$$x_n \in H_s.$$

Since

$$H_s = V \cap [s]_\eta,$$

it follows immediately that

$$x_n \in V$$

and

$$x_n \in [s]_\eta.$$

Therefore

$$\eta(x_n) = \eta(s).$$

□

The runtime may consequently undergo substantial state transformation while preserving the semantic property that defines successful regeneration:

Regeneration is invariance through transformation.

Repair algebra determines which transformations can restore the required invariants. Regenerative runtime semantics determines whether those transformations can be realized as an admissible execution. The next question is what happens when restoration of the original homeostatic condition is unavailable or unnecessary, but the runtime can preserve viability through an alternative response. This leads to compensation.

6.3 Compensation

Regeneration restores a designated homeostatic condition. A runtime may, however, preserve viability without restoring that condition. Such behavior is modeled as compensation.

Compensation therefore represents an alternative runtime response to damage or displacement: the system remains operational by entering or maintaining a viable state whose semantic or functional organization differs from the original homeostatic target.

Definition 6.23 (Compensatory State)**.** Let $H \subseteq V$ be a designated homeostatic region. A state $x \in S$ is *compensatory relative to* H if

$$x \in V \qquad \text{and} \qquad x \notin H,$$

and there exists an admissible runtime execution that maintains viability from x without requiring immediate return to H.

Thus a compensatory state is viable but non-homeostatic. It represents continued operation under an alternative runtime configuration.

Definition 6.24 (Compensatory Region)**.** A subset

$$C \subseteq V \setminus H$$

is a *compensatory region relative to* H if every state $x \in C$ admits at least one admissible viability-preserving runtime continuation.

If, in addition,

$$x \in C \quad \Longrightarrow \quad \Pi(x, c) \text{ is defined and } \Pi(x, c)(x) \in C$$

for every relevant runtime context c, then C is *policy-invariant*.

A policy-invariant compensatory region can sustain runtime operation without returning to the original homeostatic region.

Definition 6.25 (Compensatory Transformation)**.** Let $H \subseteq V$. A runtime transformation $\tau \in \mathcal{T}_{\text{comp}}$ is *compensatory at* x *relative to* H if

$$x \notin H, \qquad \tau(x) \text{ is defined,}$$

and

$$\tau(x) \in V.$$

If

$$\tau(x) \notin H,$$

then the transformation preserves viability without completing regeneration.

The defining property of compensation is therefore not restoration of the original homeostatic state but preservation of admissible operation.

Definition 6.26 (Compensatory Execution)**.** A finite runtime execution

$$\pi : x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

is *compensatory relative to* H if

$$x_i \in V \qquad (0 \le i \le n),$$

while

$$x_i \notin H$$

for at least one state following the displacement from H, and the execution contains at least one transition from $\mathcal{T}_{\text{comp}}$.

If

$$x_n \in H,$$

the compensatory execution is *transient*: compensation supports the runtime until regeneration is completed.

If

$$x_n \notin H,$$

the execution terminates in a viable non-homeostatic state.

This distinction separates compensation as a temporary recovery mechanism from compensation as a persistent alternative operating regime.

Definition 6.27 (Transient Compensation)**.** A compensatory execution is *transiently compensatory* if there exists an index $j < n$ such that

$$x_j \in V \setminus H$$

and

$$x_n \in H.$$

Transient compensation therefore preserves viability during an excursion that ultimately terminates in regeneration.

Proposition 6.28 (Compensation Can Precede Regeneration)**.** *Let*

$$x_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_k} x_k$$

be a viability-preserving compensatory execution and suppose

$$x_k \in \mathcal{B}_R(H).$$

Then there exists an admissible execution beginning at x_0 that first compensates and subsequently regenerates into H.

Proof. Since $x_k \in \mathcal{B}_R(H)$, there exists a finite admissible recovery path

$$x_k \rightsquigarrow_{\text{reg}} H.$$

Concatenating this recovery path with the compensatory execution from x_0 to x_k yields an admissible execution that first preserves viability through compensation and subsequently reaches H. □

Compensation and regeneration are therefore not mutually exclusive runtime behaviors. Compensation may form an intermediate phase of a larger regenerative execution.

Definition 6.29 (Persistent Compensation)**.** Let $C \subseteq V \setminus H$ be a compensatory region. An infinite runtime execution

$$x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} x_2 \xrightarrow{\tau_3} \cdots$$

is *persistently compensatory in C* if there exists $N \in \mathbb{N}$ such that

$$x_i \in C \qquad \text{for all } i \geq N.$$

Persistent compensation represents indefinite viable operation in an alternative runtime regime.

Proposition 6.30 (Invariant Compensatory Region Supports Persistent Operation)**.** *Let $C \subseteq V \setminus H$ be policy-invariant. If $x_0 \in C$, then every policy-generated execution that remains defined stays in C and therefore remains viable.*

Proof. Since C is policy-invariant,

$$x_i \in C \quad \Longrightarrow \quad x_{i+1} \in C.$$

By induction,

$$x_i \in C$$

for every state of the policy-generated execution. Since $C \subseteq V$, every such state is viable. □

This establishes a formal distinction between viability and homeostasis:

$$C \subseteq V \qquad \text{but} \qquad C \cap H = \varnothing.$$

A system can therefore remain viable indefinitely without returning to its original homeostatic regime.

Definition 6.31 (Semantic Compensation)**.** Let $I : S \to X$ be a runtime observable. A compensatory execution

$$x_0 \to \cdots \to x_n$$

is *semantically compensatory with respect to I* if

$$I(x_n) = I(x_0)$$

or, more generally, if

$$I(x_n) \in A_I,$$

where $A_I \subseteq X$ is a designated admissible set of semantic values.

Semantic compensation therefore permits the system to change its detailed runtime realization while preserving a specified functional invariant.

Definition 6.32 (Compensatory Invariant)**.** An observable

$$I : S \to X$$

is a *compensatory invariant on C* if every admissible compensatory transition

$$x \xrightarrow{\tau} y, \qquad x, y \in C,$$

satisfies

$$I(y) = I(x).$$

The observable I identifies what remains unchanged while the runtime operates outside its original homeostatic region.

Proposition 6.33 (Composition Preserves a Compensatory Invariant)**.** *Let*

$$x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

be a compensatory execution contained in C. If I is a compensatory invariant on C, then

$$I(x_n) = I(x_0).$$

Proof. For every i,

$$I(x_i) = I(x_{i-1}).$$

Repeated application gives

$$I(x_n) = I(x_{n-1}) = \cdots = I(x_0).$$

□

Thus compensation, like regeneration, can be characterized through invariance under transformation. The difference lies in which invariant is restored or preserved.

Definition 6.34 (Homeostatic and Functional Semantics)**.** Let

$$\eta_H : S \to \Sigma_H$$

represent homeostatic semantics and let

$$\eta_F : S \to \Sigma_F$$

represent a functional semantic observable.

A compensatory state x relative to a reference state s may satisfy

$$\eta_H(x) \neq \eta_H(s)$$

while simultaneously satisfying

$$\eta_F(x) = \eta_F(s).$$

This provides a precise representation of functional preservation without restoration of the original homeostatic organization.

Proposition 6.35 (Compensation Need Not Be Regeneration)**.** *There exist compensatory executions that are not regenerative executions.*

Proof. Let $C \subseteq V \setminus H$ be a nonempty policy-invariant compensatory region, and let $x_0 \in C$. By policy invariance, the runtime may remain in C while preserving viability. Since

$$C \cap H = \varnothing,$$

such an execution does not reach H and therefore is not regenerative. □

Definition 6.36 (Compensatory Capacity)**.** For a state $x \in S$, define its *compensatory capacity relative to* H by

$$\mathrm{Comp}_H(x) = \{y \in V \setminus H : x \rightsquigarrow y \text{ through a viability-preserving compensatory execution}\}.$$

The runtime possesses compensatory capacity at x if

$$\mathrm{Comp}_H(x) \neq \varnothing.$$

Compensatory capacity and regenerative capacity measure different properties. A state may possess both, either, or neither.

In particular, the runtime admits the following structural possibilities:

	Regenerative Capacity	Compensatory Capacity
I	yes	yes
II	yes	no
III	no	yes
IV	no	no

Case I permits both restoration and alternative viable operation. Case II permits restoration but no compensatory alternative. Case III permits continued viable operation without restoration. Case IV represents absence of both regenerative and compensatory capacity.

The distinction between regeneration and compensation is therefore not merely terminological. It separates two mathematically different responses to perturbation:

Regeneration :	restore the designated invariant,
Compensation :	preserve viability or function under an alternative regime.

A regenerative runtime may employ both mechanisms within a single execution. Regeneration characterizes return to a designated homeostatic semantic region; compensation characterizes viable operation when such return is delayed, unnecessary, or unavailable.

The remaining question is how repair transformations themselves combine within runtime execution. This requires relating the composition law of the repair algebra to the operational composition of runtime transitions.

6.4 Repair Composition

The repair algebra describes how repair transformations compose as mathematical operations. The regenerative runtime gives those same transformations an operational interpretation as transitions between runtime states. Repair composition connects these two levels.

Write

$$\mathcal{R}^*$$

for the set of finite sequences of repair transformations drawn from the full repair monoid $\mathcal{R}$. These are general repair words. They are distinct from the primitive repair words

$$\mathcal{P}_{\mathcal{R}}^*,$$

whose entries are drawn from the primitive repair basis $\mathcal{P}_{\mathcal{R}}$ and whose lengths are used in the definition of repair depth.

The central requirement is that algebraic composition and runtime execution agree: composing repairs algebraically must produce the same state transformation as executing those repairs sequentially.

Definition 6.37 (Runtime Realization of a Repair)**.** Let $\mathfrak{R}$ be a regenerative runtime whose underlying repair algebra contains the repair monoid $\mathcal{R}$. A repair $r \in \mathcal{R}$ is *runtime-realizable at* $x \in S$ if the runtime admits the transition

$$x \xrightarrow{r} r(x).$$

A repair word

$$w = (r_1, \ldots, r_n) \in \mathcal{R}^*$$

is *runtime-realizable at x* if there exists an admissible runtime execution

$$x = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} x_n$$

such that

$$x_i = r_i(x_{i-1}) \qquad (1 \le i \le n).$$

Recall that the algebraic evaluation of the repair word is

$$[\![w]\!] = r_n \circ \cdots \circ r_1.$$

The runtime interpretation therefore executes the same repairs in the temporal order

$$r_1, r_2, \ldots, r_n.$$

Theorem 6.38 (Repair-Word Runtime Correspondence)**.** *Let*

$$w = (r_1, \ldots, r_n)$$

be a repair word runtime-realizable at x. If

$$x = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} x_n,$$

then

$$x_n = [\![w]\!](x).$$

Proof. By runtime realization,

$$x_1 = r_1(x),$$

$$x_2 = r_2(x_1) = r_2(r_1(x)),$$

and, inductively,

$$x_n = r_n(r_{n-1}(\cdots r_1(x) \cdots)).$$

Hence

$$x_n = (r_n \circ \cdots \circ r_1)(x) = [\![w]\!](x).$$

□

Thus the repair algebra is not merely descriptive of runtime behavior. Its composition law determines the state transformation produced by sequential repair execution.

Definition 6.39 (Operational Repair Composition)**.** Let $r_1, r_2 \in \mathcal{R}$. Their *operational repair composition at x* is the two-step runtime execution

$$x \xrightarrow{r_1} r_1(x) \xrightarrow{r_2} r_2(r_1(x)),$$

whenever both transitions are admissible.

Its algebraic realization is

$$(r_2 \circ r_1)(x).$$

Proposition 6.40 (Closure of Repair Composition)**.** *If*

$$r_1, r_2 \in \mathcal{R},$$

then

$$r_2 \circ r_1 \in \mathcal{R}.$$

Consequently, every finite repair-only runtime execution determines a single repair transformation in $\mathcal{R}$*.*

Proof. The repair transformations form a monoid under composition. Hence $\mathcal{R}$ is closed under composition.

For a finite repair execution

$$x \xrightarrow{r_1} \cdots \xrightarrow{r_n} x_n,$$

closure gives

$$r_n \circ \cdots \circ r_1 \in \mathcal{R}.$$

By the Repair-Word Runtime Correspondence,

$$x_n = (r_n \circ \cdots \circ r_1)(x).$$

□

This permits an entire repair episode to be treated either as a sequence of runtime transitions or as one composite algebraic repair.

Definition 6.41 (Composite Repair)**.** For a repair word

$$w = (r_1, \ldots, r_n),$$

the transformation

$$R_w = [\![w]\!] = r_n \circ \cdots \circ r_1$$

is the *composite repair induced by* w*.*

Different repair words may induce the same composite repair.

Definition 6.42 (Operational Equivalence of Repair Words)**.** Let

$$u, v \in \mathcal{R}^*.$$

The words u and v are *operationally equivalent on* $U \subseteq S$, written

$$u \equiv_U v,$$

if

$$[\![u]\!](x) = [\![v]\!](x) \qquad \text{for every } x \in U.$$

Operational equivalence identifies syntactically different repair programs that induce the same effective transformation on the relevant runtime region.

Proposition 6.43 (Operational Equivalence Preserves Terminal State)**.** *Let* $u, v \in \mathcal{R}^*$ *be runtime-realizable at* $x \in U$*. If*

$$u \equiv_U v,$$

then executions of u *and* v *from* x *terminate in the same state.*

Proof. By operational equivalence,

$$[\![u]\!](x) = [\![v]\!](x).$$

By the Repair-Word Runtime Correspondence, these values are precisely the terminal states of the corresponding runtime executions. □

Definition 6.44 (Semantically Equivalent Repair Executions)**.** Let

$$u, v \in \mathcal{R}^*$$

be runtime-realizable at x. The corresponding repair executions are *semantically equivalent* if

$$\eta([\![u]\!](x)) = \eta([\![v]\!](x)).$$

Operational equivalence implies semantic equivalence, but semantic equivalence need not imply operational equivalence.

Proposition 6.45 (Operational Equivalence Implies Semantic Equivalence)**.** *If*

$$u \equiv_U v$$

and $x \in U$*, then*

$$\eta([\![u]\!](x)) = \eta([\![v]\!](x)).$$

Proof. Operational equivalence gives

$$[\![u]\!](x) = [\![v]\!](x).$$

Applying η to both sides yields the result. □

This separates equality of realization from equality of semantic effect.

Definition 6.46 (Successful Composite Repair)**.** Let $s \in V$, let $d \in \mathcal{D}$, and let $w \in \mathcal{R}^*$. The repair word w is *successfully regenerative for* (s, d) if

$$[\![w]\!](d(s)) \in H_s = V \cap [s]_\eta.$$

Equivalently,

$$[\![w]\!](d(s)) \in V$$

and

$$\eta([\![w]\!](d(s))) = \eta(s).$$

Theorem 6.47 (Successful Composite Repair Induces Regeneration)**.** *Let* $s \in V$*,* $d \in \mathcal{D}$*, and*

$$w = (r_1, \ldots, r_n) \in \mathcal{R}^*$$

be runtime-realizable at $d(s)$*. If*

$$[\![w]\!](d(s)) \in H_s,$$

then the corresponding repair-only runtime execution is regenerative with respect to H_s*.*

Proof. Let

$$d(s) = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} x_n$$

be the runtime realization of w. By the Repair-Word Runtime Correspondence,

$$x_n = [\![w]\!](d(s)).$$

By hypothesis,

$$x_n \in H_s.$$

Therefore the execution reaches the designated homeostatic region and is regenerative with respect to H_s. □

The theorem provides the direct bridge between successful repair in the algebra and successful regeneration in the runtime.

Definition 6.48 (Intermediate Viability of Repair Composition)**.** A repair word

$$w = (r_1, \ldots, r_n)$$

is *intermediately viable at* x if its runtime realization

$$x = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} x_n$$

satisfies

$$x_i \in V \qquad (0 \le i \le n).$$

A composite repair may have a viable terminal state without being intermediately viable. Thus endpoint correctness and execution safety are distinct properties.

Proposition 6.49 (Intermediate Viability Implies Viability-Preserving Regeneration)**.** *Let w be successfully regenerative for (s, d) and suppose w is intermediately viable at $d(s)$. Then its runtime realization is a viability-preserving regenerative execution into H_s.*

Proof. Successful regeneration gives

$$[\![w]\!](d(s)) \in H_s.$$

Intermediate viability gives

$$x_i \in V$$

for every state of the runtime realization. Therefore the execution both remains viable and terminates in H_s. □

This distinction becomes important when repair operations have intermediate effects that are not visible from their final composite transformation.

Definition 6.50 (Commuting Runtime Repairs)**.** Two repairs $r, q \in \mathcal{R}$ *commute on* $U \subseteq S$ if

$$r(q(x)) = q(r(x)) \qquad \text{for every } x \in U.$$

Proposition 6.51 (Order Invariance of Commuting Repairs)**.** *Suppose r and q commute on U, and suppose both repair orders are runtime-realizable at $x \in U$ with all relevant intermediate states remaining in U. Then*

$$x \xrightarrow{r} r(x) \xrightarrow{q} q(r(x))$$

and

$$x \xrightarrow{q} q(x) \xrightarrow{r} r(q(x))$$

have the same terminal state.

Proof. Since r and q commute on U,

$$q(r(x)) = r(q(x)).$$

These are precisely the terminal states of the two runtime executions. □

Thus commutativity induces an operational order invariance.

Definition 6.52 (Idempotent Runtime Repair)**.** A repair $r \in \mathcal{R}$ is *idempotent on* $U \subseteq S$ if

$$r(r(x)) = r(x) \qquad \text{for every } x \in U.$$

Proposition 6.53 (Multiplicity Invariance of Idempotent Repair)**.** *Let r be idempotent on U, and suppose the repeated repair execution remains in U. Then for every integer $n \geq 1$,*

$$r^n(x) = r(x).$$

Proof. The claim follows by induction. For $n = 1$ the statement is immediate. If

$$r^n(x) = r(x),$$

then

$$r^{n+1}(x) = r(r^n(x)) = r(r(x)) = r(x)$$

by idempotence. □

Hence repeated execution of an idempotent repair does not alter its effective result after the first application.

Theorem 6.54 (Canonical Runtime Realization under Commutativity and Idempotence)**.** *Let*

$$\mathcal{Q} = \{q_1, \ldots, q_m\} \subseteq \mathcal{R}$$

be a finite family of repairs that commute pairwise and are idempotent on a runtime-stable region $U \subseteq S$. Suppose all relevant repair executions remain in U.

Then every repair word

$$w \in \mathcal{Q}^*$$

is operationally equivalent on U to a word containing each primitive repair occurring in w exactly once.

Consequently, the effective runtime behavior of w depends only on the subset of primitive repairs appearing in w, not on their order or multiplicity.

Proof. By idempotence, repeated occurrences of any primitive repair may be removed without changing the evaluated transformation on U. By pairwise commutativity, the remaining primitive repairs may be reordered without changing the evaluated transformation. Hence w is operationally equivalent on U to a canonical word containing each primitive repair occurring in w exactly once. □

This transfers the canonicalization results of the repair algebra directly into runtime execution.

If

$$J \subseteq \mathcal{Q}$$

denotes the subset of primitive repairs appearing in a word, the effective repair may therefore be written

$$R_J = \prod_{q \in J} q,$$

where the product is independent of ordering on the stable region U.

The runtime search space consequently descends from arbitrary repair words

$$\mathcal{Q}^*$$

to finite repair subsets

$$2^{\mathcal{Q}}$$

whenever the commutativity, idempotence, and stability hypotheses hold.

Definition 6.55 (Repair-Composition Trace)**.** Let

$$w = (r_1, \ldots, r_n)$$

be runtime-realizable at x. Its *repair-composition trace* is the finite sequence

$$\mathrm{Tr}_w(x) = (x_0, x_1, \ldots, x_n),$$

where

$$x_0 = x$$

and

$$x_i = r_i(x_{i-1}) \qquad (1 \le i \le n).$$

The composite repair determines the endpoint

$$x_n = [\![w]\!](x),$$

whereas the trace records the intermediate operational behavior.

This yields an important separation:

$$\boxed{\text{same composite repair } \not\Rightarrow \text{ same runtime trace.}}$$

Two repair programs may therefore have identical terminal semantics while differing in intermediate viability, cost, duration, resource use, or safety.

Definition 6.56 (Trace-Safe Repair Composition)**.** Let $U_{\mathrm{safe}} \subseteq V$ be a designated safe runtime region. A runtime-realizable repair word w is *trace-safe at* x if

$$\mathrm{Tr}_w(x) \subseteq U_{\mathrm{safe}}.$$

Proposition 6.57 (Trace Safety Implies Intermediate Viability)**.** *If*

$$U_{\mathrm{safe}} \subseteq V$$

and w is trace-safe at x, then w is intermediately viable at x.

Proof. Every state in the repair-composition trace belongs to U_{safe}. Since

$$U_{\mathrm{safe}} \subseteq V,$$

every state in the trace is viable. □

Repair composition therefore exposes two different levels of correctness:

algebraic correctness :	the composite repair has the required endpoint,
operational correctness :	the runtime reaches that endpoint through an admissible trace.

The repair algebra determines the compositional structure of repair. The regenerative runtime realizes that structure as execution. Agreement between these two levels provides the foundation for runtime verification.

The next section develops this verification layer by identifying runtime invariants, safety properties, and conditions under which repair execution is correct.

7 Verification

7.1 Runtime Invariants

A regenerative runtime is characterized not by the absence of change, but by the preservation or recovery of selected properties through change. Damage, repair, adaptation, compensation, and ordinary runtime evolution may all alter the underlying state while leaving some higher-level property unchanged.

Verification therefore requires an explicit account of the properties that must survive runtime execution.

Definition 7.1 (Runtime Invariant)**.** Let

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi)$$

be a regenerative runtime, and let

$$I : S \to \Lambda$$

be an observable on runtime states.

Let

$$X \subseteq S$$

be a designated runtime region and

$$\mathcal{T}_I \subseteq \mathcal{T}$$

a class of admissible transformations.

The observable I is a *runtime invariant on X under $\mathcal{T}_I$* when

$$I(\tau(x)) = I(x)$$

for every

$$x \in X$$

and every

$$\tau \in \mathcal{T}_I$$

for which $\tau(x)$ is defined and belongs to X.

Thus invariance is always relative to a specified observable, state region, and class of transformations. A quantity need not remain invariant under every possible runtime transition in order to constitute a meaningful runtime invariant.

The semantic observation map

$$\eta : S \to \Omega$$

provides the fundamental invariant of the regenerative runtime.

Definition 7.2 (Semantic Invariance)**.** A runtime transformation

$$\tau : S \rightharpoonup S$$

is *semantically invariant at* x when

$$\eta(\tau(x)) = \eta(x).$$

It is semantically invariant on

$$X \subseteq S$$

when this equality holds for every $x \in X$ at which τ is defined.

Semantic invariance permits the internal state of the system to change while its selected organizational meaning remains fixed.

In particular,

$$\tau(x) \neq x$$

does not imply semantic change. It may instead be the case that

$$\tau(x) \sim_\eta x.$$

This distinction is central to regenerative behavior.

Proposition 7.3 (Closure of Semantic Invariance)**.** *Let*

$$\tau_1, \tau_2 : S \rightharpoonup S$$

be semantically invariant on a region $X \subseteq S$*. Suppose that*

$$x \in X, \qquad \tau_1(x) \in X, \qquad \tau_2(\tau_1(x)) \in X.$$

Then

$$\tau_2 \circ \tau_1$$

is semantically invariant at x*.*

Proof. Semantic invariance of τ_1 gives

$$\eta(\tau_1(x)) = \eta(x).$$

Semantic invariance of τ_2 gives

$$\eta(\tau_2(\tau_1(x))) = \eta(\tau_1(x)).$$

Therefore

$$\eta(\tau_2(\tau_1(x))) = \eta(x).$$

□

Hence semantic invariance is compositional whenever the relevant execution remains within the region on which the transformations preserve the observable.

Viability supplies a second fundamental runtime property.

Definition 7.4 (Viability Invariant)**.** A runtime transformation

$$\tau : S \rightharpoonup S$$

is *viability-preserving on* $X \subseteq V$ when

$$\tau(x) \in V$$

for every $x \in X$ at which τ is defined.

Unlike semantic invariance, viability preservation does not require

$$\eta(\tau(x)) = \eta(x).$$

A system may therefore remain operational while undergoing a semantic change.

This yields four elementary transition classes:

	viability preserved	semantic signature preserved
homeostatic preservation	yes	yes
viable semantic change	yes	no
semantic preservation outside viability	no	yes
runtime failure/change	no	no

The first class corresponds to transformations that preserve both continued operation and the selected organizational semantics.

Definition 7.5 (Homeostatic Invariant)**.** For a reference state $s \in V$, recall the homeostatic region

$$H_s = V \cap [s]_\eta.$$

A runtime transformation τ *preserves the homeostatic invariant relative to* s on H_s when

$$\tau(H_s) \subseteq H_s.$$

Proposition 7.6 (Homeostatic Preservation)**.** *Suppose τ is viability-preserving and semantically invariant on H_s. Then*

$$\tau(H_s) \subseteq H_s.$$

Proof. Let

$$x \in H_s.$$

Then

$$x \in V \qquad \text{and} \qquad \eta(x) = \eta(s).$$

Viability preservation gives

$$\tau(x) \in V,$$

while semantic invariance gives

$$\eta(\tau(x)) = \eta(x) = \eta(s).$$

Hence

$$\tau(x) \in V \cap [s]_\eta = H_s.$$

□

This proposition identifies homeostasis as the simultaneous preservation of viability and semantic organization.

The same idea extends from individual transformations to complete runtime executions.

Definition 7.7 (Invariant-Preserving Execution)**.** Let

$$\pi : x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

be a finite runtime execution.

For an observable

$$I : S \to \Lambda,$$

the execution π is *I-invariant* when

$$I(x_i) = I(x_0)$$

for every

$$0 \le i \le n.$$

It is *semantically invariant* when

$$\eta(x_i) = \eta(x_0)$$

for every i.

Proposition 7.8 (Local Preservation Implies Execution Invariance)**.** *Suppose every transition of*

$$\pi : x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

satisfies

$$I(x_i) = I(x_{i-1}) \qquad (1 \le i \le n).$$

Then π is I-invariant.

Proof. Repeated application of the transition equalities gives

$$I(x_n) = I(x_{n-1}) = \cdots = I(x_1) = I(x_0).$$

The same argument applies to every prefix of the execution, so

$$I(x_i) = I(x_0)$$

for every i. □

Regeneration differs from ordinary invariant preservation because the required invariant may temporarily be lost and subsequently recovered.

Definition 7.9 (Recovered Invariant)**.** Let

$$I : S \to \Lambda$$

and let

$$\pi : x_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} x_n$$

be a runtime execution.

Suppose a reference value

$$\lambda \in \Lambda$$

is designated.

The invariant I is *recovered by* π when

$$I(x_n) = \lambda,$$

even if

$$I(x_i) \neq \lambda$$

for some intermediate state x_i.

This separates two fundamentally different runtime phenomena:

maintenance :	an invariant remains true throughout execution,
regeneration :	an invariant that has been lost is restored by execution.

For repair following damage, the relevant reference value is the semantic signature of the original viable state.

Theorem 7.10 (Semantic Recovery under Successful Regeneration)**.** *Let*

$$s \in V, \qquad d \in \mathcal{D},$$

and let

$$\pi : d(s) = x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

be regenerative with respect to

$$H_s = V \cap [s]_\eta.$$

Then

$$x_n \in V$$

and

$$\eta(x_n) = \eta(s).$$

Thus the semantic invariant associated with s is recovered at the terminal state.

Proof. Since π is regenerative with respect to H_s,

$$x_n \in H_s.$$

By definition,

$$H_s = V \cap [s]_\eta.$$

Therefore

$$x_n \in V$$

and

$$x_n \in [s]_\eta.$$

The latter condition is equivalent to

$$\eta(x_n) = \eta(s).$$

□

Importantly, this theorem does not require

$$x_n = s.$$

Consequently, regeneration does not require invariance of microscopic or internal state. What is restored is the selected semantic organization.

$$\boxed{x_n \neq s \qquad \text{while} \qquad \eta(x_n) = \eta(s)}$$

is therefore a canonical regenerative outcome.

Runtime invariants may also be considered under changes of representation. The repair-algebra morphisms developed earlier provide the corresponding structural mechanism.

Proposition 7.11 (Representational Preservation of Semantic Invariants)**.** *Let*

$$\mathfrak{F} : \mathcal{A} \to \mathcal{B}$$

be a repair-algebra morphism with state map

$$f : S_A \to S_B$$

and signature map

$$g : \Omega_A \to \Omega_B.$$

If

$$\eta_A(x) = \eta_A(y),$$

then

$$\eta_B(f(x)) = \eta_B(f(y)).$$

Proof. Semantic compatibility gives

$$\eta_B(f(x)) = g(\eta_A(x))$$

and

$$\eta_B(f(y)) = g(\eta_A(y)).$$

Since

$$\eta_A(x) = \eta_A(y),$$

their images under g are equal. Hence

$$\eta_B(f(x)) = \eta_B(f(y)).$$

□

Thus semantic equivalence survives structure-preserving translation between repair algebras. Under the stronger faithfulness conditions developed earlier, semantic equivalence may also be reflected from the target representation back to the source.

The verification problem for a regenerative runtime can therefore be stated in invariant form:

Which properties must survive transformation?
Which properties may temporarily fail but must be recovered?
Which transformations preserve or restore them?

Repair algebras answer these questions at the level of transformations and composition. Regenerative runtimes answer them at the level of execution.

In this sense, regeneration is not invariance of state. It is the preservation or recovery of selected invariants through state transformation.

The next subsection strengthens this verification layer by requiring not only correct invariant behavior at the endpoint of execution, but safety throughout the runtime trajectory.

7.2 Safety Properties

Runtime invariants specify properties that must be preserved or recovered during regenerative execution. Safety strengthens this requirement by constraining the states and transitions through which execution is permitted to proceed.

A repair may therefore be correct at its endpoint while nevertheless being unsafe as an execution. Verification of a regenerative runtime must distinguish terminal correctness from trajectory safety.

Definition 7.12 (Safe Region)**.** Let

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi)$$

be a regenerative runtime.

A subset

$$S_{\text{safe}} \subseteq S$$

is a *safe region* if states in S_{safe} satisfy the designated runtime safety requirements.

When safety requires continued viability, we assume

$$S_{\text{safe}} \subseteq V.$$

The safe region may impose requirements stronger than viability alone. A viable state may remain operational while nevertheless violating a designated safety constraint.

Definition 7.13 (Safe Transition)**.** A runtime transition

$$x \xrightarrow{\tau} y$$

is *safe relative to* S_{safe} if

$$x \in S_{\text{safe}} \qquad \Longrightarrow \qquad y \in S_{\text{safe}}.$$

A runtime transformation τ is *safe on* S_{safe} if

$$\tau(x) \in S_{\text{safe}}$$

for every $x \in S_{\text{safe}}$ at which τ is defined.

Equivalently, when τ is total on the safe region,

$$\tau(S_{\text{safe}}) \subseteq S_{\text{safe}}.$$

Definition 7.14 (Safe Execution)**.** A finite runtime execution

$$\pi : x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

is *safe relative to* S_{safe} if

$$x_i \in S_{\text{safe}} \qquad (0 \leq i \leq n).$$

Safety is therefore a trace property rather than merely an endpoint property.

Theorem 7.15 (Local Safety Implies Execution Safety)**.** *Let*

$$x_0 \in S_{\text{safe}},$$

and suppose every transition selected during an execution is safe on S_{safe}*. Then the entire execution remains in* S_{safe}*.*

Proof. The result follows by induction on execution length.

The initial state satisfies

$$x_0 \in S_{\text{safe}}.$$

Suppose

$$x_i \in S_{\text{safe}}.$$

Since the transition

$$x_i \xrightarrow{\tau_{i+1}} x_{i+1}$$

is safe,

$$x_{i+1} \in S_{\text{safe}}.$$

Hence every state of the execution belongs to S_{safe}. □

This gives a local verification principle: global trajectory safety follows from an initially safe state together with preservation of the safe region by every executed transformation.

Definition 7.16 (Safety Invariant)**.** A predicate

$$P : S \to \{\mathsf{true}, \mathsf{false}\}$$

is a *safety invariant* for a class of runtime executions if

$$P(x_0)$$

implies

$$P(x_i)$$

for every state x_i occurring in every execution in the class.

Every safe region determines a corresponding predicate

$$P_{\text{safe}}(x) \iff x \in S_{\text{safe}}.$$

Thus region-based and predicate-based formulations of runtime safety are equivalent.

Definition 7.17 (Unsafe State)**.** The *unsafe region* associated with S_{safe} is

$$S_{\text{unsafe}} = S \setminus S_{\text{safe}}.$$

An execution exhibits a *safety violation* if there exists some i such that

$$x_i \in S_{\text{unsafe}}.$$

If

$$S_{\text{safe}} \subseteq V,$$

then every nonviable state is necessarily outside the safe region whenever the safe region contains all states permitted by the safety specification.

However, the converse need not hold:

$$x \in V \quad \not\Rightarrow \quad x \in S_{\text{safe}}.$$

Safety can therefore impose stronger constraints than viability.

Definition 7.18 (Safe Repair)**.** Let

$$s \in V, \qquad d \in \mathcal{D},$$

and let

$$w = (r_1, \ldots, r_n) \in \mathcal{R}^*$$

be runtime-realizable at $d(s)$.

The repair word w is a *safe repair relative to* S_{safe} if its repair-composition trace satisfies

$$\text{Tr}_w(d(s)) \subseteq S_{\text{safe}}.$$

A safe repair is therefore verified at every intermediate state rather than only at its terminal state.

Definition 7.19 (Endpoint-Correct Repair)**.** A runtime-realizable repair word w is *endpoint-correct for* (s, d) if

$$[\![w]\!](d(s)) \in H_s.$$

Endpoint correctness alone does not imply safety.

Proposition 7.20 (Endpoint Correctness Does Not Imply Trace Safety)**.** *There may exist an endpoint-correct repair word w such that*

$$\text{Tr}_w(d(s)) \not\subseteq S_{\text{safe}}.$$

Proof. Endpoint correctness constrains only

$$[\![w]\!](d(s)),$$

the terminal state of the repair execution.

It imposes no condition on intermediate states. Hence an execution may pass through some

$$x_i \notin S_{\text{safe}}$$

before eventually terminating in

$$H_s.$$

Therefore endpoint correctness does not imply trace safety. □

This distinction is essential for executable repair systems:

$$\boxed{\text{correct destination } \not\Rightarrow \text{ safe execution.}}$$

Definition 7.21 (Safely Regenerative Execution)**.** A runtime execution is *safely regenerative relative to* (H, S_{safe}) if it is regenerative with respect to H and safe relative to S_{safe}.

Thus

$$x_n \in H$$

and

$$x_i \in S_{\text{safe}} \qquad (0 \le i \le n).$$

Theorem 7.22 (Safe Successful Repair Induces Safe Regeneration)**.** *Let w be a runtime-realizable repair word for (s, d). Suppose*

$$[\![w]\!](d(s)) \in H_s$$

and

$$\mathrm{Tr}_w(d(s)) \subseteq S_{\text{safe}}.$$

Then the runtime realization of w is safely regenerative with respect to

$$(H_s, S_{\text{safe}}).$$

Proof. The first hypothesis implies that the terminal state belongs to H_s, so the execution is regenerative.

The second hypothesis states that every state of the execution belongs to S_{safe}, so the execution is safe.

Hence the execution is safely regenerative. □

Runtime safety may also be expressed as avoidance of forbidden states.

Definition 7.23 (Forbidden Region)**.** Let

$$F \subseteq S$$

be a designated set of forbidden runtime states.

An execution

$$\pi = (x_0, \ldots, x_n)$$

avoids F if

$$x_i \notin F \qquad (0 \le i \le n).$$

Taking

$$S_{\text{safe}} = S \setminus F$$

makes forbidden-state avoidance equivalent to safe-region preservation.

Definition 7.24 (Safety-Preserving Policy)**.** A runtime policy Π is *safety-preserving on* S_{safe} if every policy transition selected from a safe state returns a safe state.

That is, whenever

$$x \in S_{\text{safe}}$$

and

$$\tau = \Pi(x, c)$$

is defined,

$$\tau(x) \in S_{\text{safe}}.$$

Theorem 7.25 (Policy Safety)**.** *Suppose*

$$x_0 \in S_{\text{safe}}$$

and Π *is safety-preserving on* S_{safe}*. Then every policy-generated execution beginning at* x_0*, for as long as it remains defined, stays in* S_{safe}*.*

Proof. At each state $x_i \in S_{\text{safe}}$, safety preservation of Π gives

$$x_{i+1} = \Pi(x_i, c_i)(x_i) \in S_{\text{safe}}.$$

The result follows inductively. □

Safety and regeneration may nevertheless conflict. A state may be structurally recoverable while possessing no recovery path that satisfies a given safety constraint.

Definition 7.26 (Safe Regenerative Basin)**.** For a homeostatic region H and safe region S_{safe}, define

$$\mathcal{B}_R^{\text{safe}}(H) = \left\{ x \in S_{\text{safe}} \;\middle|\; \begin{array}{c} \text{there exists a finite admissible execution} \\ x = x_0 \rightsquigarrow x_n \in H \\ \text{with } x_i \in S_{\text{safe}} \text{ for all } i \end{array} \right\}.$$

By construction,

$$\mathcal{B}_R^{\text{safe}}(H) \subseteq \mathcal{B}_R(H).$$

Proposition 7.27 (Safe Recoverability Is Stronger Than Recoverability)**.** *There may exist*

$$x \in \mathcal{B}_R(H)$$

such that

$$x \notin \mathcal{B}_R^{\text{safe}}(H).$$

Proof. Membership in $\mathcal{B}_R(H)$ requires only the existence of an admissible path to H. Every such path may nevertheless intersect S_{unsafe}.

In that case a recovery path exists, but no safe recovery path exists. Hence

$$x \in \mathcal{B}_R(H)$$

while

$$x \notin \mathcal{B}_R^{\text{safe}}(H).$$

□

This produces a hierarchy of increasingly strong runtime guarantees:

recoverable $\supseteq$ safely recoverable $\supseteq$ safely recoverable under the implemented policy.

The distinction between these levels is operationally important. Existence of a mathematically valid repair does not imply existence of a safe repair, and existence of a safe repair does not imply that the runtime policy will select it.

Definition 7.28 (Safety Certificate)**.** Let

$$\pi = (x_0, \ldots, x_n)$$

be a finite runtime execution.

A *safety certificate* for π relative to a predicate P consists of evidence that

$$P(x_i)$$

holds for every

$$0 \leq i \leq n.$$

For a repair word

$$w = (r_1, \ldots, r_n),$$

such a certificate verifies the intermediate states generated by the execution rather than merely the value

$$[\![w]\!](d(s)).$$

This distinction identifies the computational verification boundary:

endpoint verification :	verify the result of the composite transformation,
trace verification :	verify every executed intermediate state.

A regenerative runtime therefore supports stronger guarantees than successful repair alone. It can require that the system restore the desired semantic invariant while remaining within a formally specified safe region throughout execution.

The final verification problem is then to combine the algebraic, semantic, and operational requirements into a single notion of repair correctness.

7.3 Repair Correctness

Repair correctness combines the algebraic, semantic, and operational requirements developed throughout the preceding sections. A repair is not fully correct merely because it transforms a damaged state into some viable state. Nor is endpoint restoration sufficient when the corresponding runtime execution passes through forbidden or inadmissible states.

A correct repair must therefore be evaluated at several levels: the repair transformation must be algebraically admissible, its terminal state must restore the required semantic organization, and its runtime realization must satisfy the relevant execution constraints.

Definition 7.29 (Algebraically Valid Repair)**.** Let

$$\mathcal{A} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \circ, \mathrm{id}_S)$$

be a repair algebra.

For

$$s \in V, \qquad d \in \mathcal{D},$$

a transformation r is an *algebraically valid repair candidate* when

$$r \in \mathcal{R}.$$

A repair word

$$w = (r_1, \ldots, r_n)$$

is algebraically valid when

$$r_i \in \mathcal{R} \qquad (1 \leq i \leq n).$$

Since $\mathcal{R}$ is closed under composition, every algebraically valid repair word determines a composite repair

$$[\![w]\!] = r_n \circ \cdots \circ r_1 \in \mathcal{R}.$$

Algebraic validity alone does not imply successful repair.

Definition 7.30 (Semantically Correct Repair)**.** Let

$$s \in V, \qquad d \in \mathcal{D}.$$

An algebraically valid repair transformation $r \in \mathcal{R}$ is *semantically correct for* (s, d) when

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s).$$

Equivalently,

$$r(d(s)) \in H_s = V \cap [s]_\eta.$$

Thus semantic correctness coincides with successful repair at the algebraic level.
For a repair word w, semantic correctness requires

$$[\![w]\!](d(s)) \in H_s.$$

Definition 7.31 (Operationally Correct Repair)**.** Let

$$w = (r_1, \ldots, r_n)$$

be a repair word that is runtime-realizable at $d(s)$, with execution

$$d(s) = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} x_n.$$

The repair word w is *operationally correct relative to an admissible region* $U \subseteq S$ if

$$x_i \in U \qquad (0 \leq i \leq n)$$

and

$$x_n \in H_s.$$

When

$$U = S_{\text{safe}},$$

operational correctness requires trace safety in addition to successful regeneration.

Definition 7.32 (Verified Repair)**.** Let

$$s \in V, \qquad d \in \mathcal{D},$$

and let $w \in \mathcal{R}^*$ be runtime-realizable at $d(s)$.

The repair word w is a *verified repair for* (s, d) *relative to* S_{safe} if all of the following hold:

$$\begin{aligned} &\text{(i)} \quad w \in \mathcal{R}^*, \\ &\text{(ii)} \quad [\![w]\!](d(s)) \in V, \\ &\text{(iii)} \quad \eta([\![w]\!](d(s))) = \eta(s), \\ &\text{(iv)} \quad \mathrm{Tr}_w(d(s)) \subseteq S_{\text{safe}}. \end{aligned}$$

These conditions correspond respectively to algebraic validity, viability restoration, semantic restoration, and execution safety.

Since

$$H_s = V \cap [s]_\eta,$$

conditions (ii) and (iii) may be combined as

$$[\![w]\!](d(s)) \in H_s.$$

Hence verified repair admits the compact characterization

$$\boxed{[\![w]\!](d(s)) \in H_s \quad \text{and} \quad \mathrm{Tr}_w(d(s)) \subseteq S_{\text{safe}}.}$$

Theorem 7.33 (Verified Repair Correctness)**.** *Let w be a verified repair for (s, d) relative to S_{safe}. Then its runtime realization is a safely regenerative execution into H_s.*

Proof. By verified-repair condition (ii),

$$[\![w]\!](d(s)) \in V.$$

By condition (iii),

$$\eta([\![w]\!](d(s))) = \eta(s).$$

Therefore

$$[\![w]\!](d(s)) \in V \cap [s]_\eta = H_s.$$

By the Repair-Word Runtime Correspondence, the terminal state of the runtime realization is precisely

$$x_n = [\![w]\!](d(s)).$$

Hence the execution regenerates into H_s.

Condition (iv) gives

$$\mathrm{Tr}_w(d(s)) \subseteq S_{\text{safe}},$$

so every state of the runtime execution is safe.

Therefore the runtime realization is safely regenerative. □

The theorem unifies the algebraic and runtime notions of successful repair.

Definition 7.34 (Exact Verified Repair)**.** A verified repair w for (s, d) is *exact* when

$$[\![w]\!](d(s)) = s.$$

It is *semantic* when

$$[\![w]\!](d(s)) \sim_\eta s$$

without requiring equality of states.

Thus exact verified repair implies semantic verified repair.

Proposition 7.35 (Exact Verified Repair Implies Semantic Verified Repair)**.** *If*

$$[\![w]\!](d(s)) = s,$$

then

$$\eta([\![w]\!](d(s))) = \eta(s).$$

Proof. Apply η to the equality

$$[\![w]\!](d(s)) = s.$$

□

The converse need not hold. Regenerative correctness therefore does not require reconstruction of the original state.

Definition 7.36 (Repair Correctness Predicate)**.** For fixed s, d, and S_{safe}, define

$$\text{Correct}_{s,d}(w)$$

by

$$\text{Correct}_{s,d}(w) \iff ([\![w]\!](d(s)) \in H_s) \wedge (\text{Tr}_w(d(s)) \subseteq S_{\text{safe}})\,.$$

The correctness problem for a proposed repair word is therefore the decision problem

$$\boxed{\text{Given } w, \text{ determine whether } \text{Correct}_{s,d}(w).}$$

This differs fundamentally from the optimization problem developed earlier. Verification asks whether a proposed repair is correct. Optimization asks whether a correct repair exists subject to a bound such as

$$|w| \leq k.$$

The distinction mirrors the complexity boundary established for bounded repair depth: checking a proposed repair may be efficient even when finding an optimal repair is computationally difficult.

Proposition 7.37 (Correctness Is Preserved under Trace-Safe Operational Equivalence)**.** *Let*

$$u, v \in \mathcal{R}^*$$

be runtime-realizable repair words at $d(s)$. Suppose

$$u \equiv_U v$$

for a region U containing the relevant terminal states, and suppose both traces satisfy

$$\text{Tr}_u(d(s)) \subseteq S_{\text{safe}}$$

and

$$\text{Tr}_v(d(s)) \subseteq S_{\text{safe}}.$$

If u is a verified repair for (s, d), then v is also a verified repair for (s, d).

Proof. Since $u \equiv_U v$,

$$[\![u]\!](d(s)) = [\![v]\!](d(s)).$$

Because u is verified,

$$[\![u]\!](d(s)) \in H_s.$$

Hence

$$[\![v]\!](d(s)) \in H_s.$$

By hypothesis,

$$\mathrm{Tr}_v(d(s)) \subseteq S_{\mathrm{safe}}.$$

Therefore v satisfies both endpoint correctness and trace safety and is a verified repair. □

Operational equivalence alone is insufficient because equivalent repair words may have different intermediate traces. Correctness is preserved only when the alternative realization also satisfies the required trace constraints.

Definition 7.38 (Correctness-Preserving Repair Transformation)**.** Let

$$C_{s,d} = \{w \in \mathcal{R}^* : \mathrm{Correct}_{s,d}(w)\}.$$

A transformation

$$\Gamma : \mathcal{R}^* \to \mathcal{R}^*$$

is *correctness-preserving for* (s, d) if

$$w \in C_{s,d} \quad \Longrightarrow \quad \Gamma(w) \in C_{s,d}.$$

Canonicalization, normalization, optimization, and compilation of repair programs should ideally satisfy this property.

This provides a direct connection between the algebraic theory and an eventual executable implementation: transformations of repair programs may change their representation while preserving their verified behavior.

Theorem 7.39 (Correctness of Canonicalization under Safe Realization)**.** *Let*

$$\mathcal{Q} = \{q_1, \ldots, q_m\} \subseteq \mathcal{R}$$

be pairwise commuting and idempotent on a runtime-stable region U. Let

$$w \in \mathcal{Q}^*$$

be a verified repair whose execution remains in U.

Let

$$\mathrm{can}(w)$$

be a canonical repair word containing each primitive repair occurring in w exactly once.

Suppose $\mathrm{can}(w)$ *is runtime-realizable at* $d(s)$ *and*

$$\mathrm{Tr}_{\mathrm{can}(w)}(d(s)) \subseteq S_{\mathrm{safe}} \cap U.$$

Then

$$\mathrm{can}(w)$$

is also a verified repair.

Proof. By commutativity and idempotence on U,

$$[\![\mathrm{can}(w)]\!] = [\![w]\!]$$

on the relevant runtime region.

Since w is verified,

$$[\![w]\!](d(s)) \in H_s.$$

Therefore

$$[\![\mathrm{can}(w)]\!](d(s)) \in H_s.$$

By hypothesis, the canonical realization satisfies

$$\mathrm{Tr}_{\mathrm{can}(w)}(d(s)) \subseteq S_{\mathrm{safe}}.$$

Hence the canonical repair satisfies both endpoint correctness and trace safety and is verified. □

The theorem shows that algebraic simplification may be transferred to runtime execution without sacrificing correctness, provided that the simplified realization remains safe.

Definition 7.40 (Repair Certificate)**.** A *repair certificate* for (s, d) consists of a finite runtime-realizable repair word

$$w = (r_1, \ldots, r_n)$$

together with sufficient evidence to verify

$$[\![w]\!](d(s)) \in H_s$$

and

$$\mathrm{Tr}_w(d(s)) \subseteq S_{\mathrm{safe}}.$$

A repair certificate therefore witnesses both successful semantic recovery and safe operational realization.

Proposition 7.41 (Soundness of Repair Certificates)**.** *Every valid repair certificate determines a verified repair.*

Proof. A valid certificate establishes precisely the two defining conditions

$$[\![w]\!](d(s)) \in H_s$$

and

$$\mathrm{Tr}_w(d(s)) \subseteq S_{\mathrm{safe}}.$$

Hence

$$\mathrm{Correct}_{s,d}(w)$$

holds. □

The resulting verification architecture can be summarized as

repair algebra
↓
repair word and composite transformation
↓
runtime realization
↓
trace verification
↓
semantic restoration
↓
verified regeneration.

The central distinction is therefore between a transformation that merely produces a desirable result and a repair whose complete execution is verified.

A regenerative runtime supports the stronger claim:

the required invariant is restored through an admissible, verified execution.

This completes the verification layer of the regenerative-runtime framework. The remaining sections examine how the resulting algebraic and operational machinery may be interpreted and applied in computational biology, systems biology, formal biological verification, and executable compiler architectures.

8 Potential Applications

8.1 Computational Biology

The repair-algebra and regenerative-runtime framework is intentionally domain-independent. Its states, semantic observables, damage transformations, repair operations, transition systems, and invariants may be instantiated in any domain admitting an appropriate mathematical representation.

Biological systems provide a particularly natural class of potential instantiations because they exhibit persistent state change while maintaining, losing, adapting, and recovering functional organization.

The framework does not assume that biological repair is literally identical to a computational repair program. Rather, it provides a formal language in which hypotheses about biological repair and regeneration can be stated as precise transformation, reachability, invariance, and verification problems.

Definition 8.1 (Biological Runtime Model)**.** A *biological runtime model* is an instantiation

$$\mathfrak{R}_B = (S_B, V_B, \Omega_B, \eta_B, \mathcal{D}_B, \mathcal{R}_B, \mathcal{T}_B, \mathcal{C}_B, \Pi_B)$$

of a regenerative runtime in which the elements of S_B represent selected biological states and the remaining components are assigned biological interpretations.

In particular:

S_B : modeled biological states,
V_B : states satisfying selected viability conditions,
Ω_B : space of biological semantic signatures,
η_B : observable biological organization,
$\mathcal{D}_B$: modeled perturbation or damage transformations,
$\mathcal{R}_B$: modeled repair transformations,
$\mathcal{T}_B$: admissible biological runtime transformations,
$\mathcal{C}_B$: modeled biological runtime contexts,
Π_B : modeled transition-selection policy.

The construction is relative to the biological scale and observables chosen for the model. A state may represent, for example, a molecular configuration, cellular state, tissue configuration, regulatory state, or another level of biological organization.

The framework therefore does not prescribe a unique biological state space.

Definition 8.2 (Biological Semantic Signature)**.** Let

$$\eta_B : S_B \to \Omega_B$$

be the semantic observation map of a biological runtime model.

The value

$$\eta_B(x)$$

is the *biological semantic signature* assigned to state x.

Two biological states are semantically equivalent relative to the model when

$$x \sim_{\eta_B} y \iff \eta_B(x) = \eta_B(y).$$

The semantic signature may encode whichever properties the model regards as organizationally significant. Consequently, biological semantic equivalence need not imply molecular or microscopic identity.

This distinction permits a mathematical expression of a common feature of biological regeneration: restoration of organization or function without reconstruction of the exact preceding microstate.

For a reference state $s \in V_B$, define

$$H_s^B = V_B \cap [s]_{\eta_B}.$$

The region H_s^B contains viable states that reproduce the selected semantic signature of s.

Definition 8.3 (Biological Damage Model)**.** A transformation

$$d \in \mathcal{D}_B$$

represents a *modeled biological damage process* when it maps a biological state to a state representing the selected consequences of a perturbation:

$$s \longmapsto d(s).$$

No assumption is made that every biological perturbation is deterministic or adequately represented by a single transformation. The deterministic framework developed here represents one mathematical level of approximation; stochastic and continuous extensions may be introduced separately.

Definition 8.4 (Biological Repair Model)**.** A transformation

$$r \in \mathcal{R}_B$$

is a *modeled biological repair* of d at s when

$$r(d(s)) \in V_B$$

and

$$\eta_B(r(d(s))) = \eta_B(s).$$

Equivalently,

$$r(d(s)) \in H_s^B.$$

The repaired state need not satisfy

$$r(d(s)) = s.$$

Thus the framework distinguishes exact biological restoration from semantic biological restoration.

Proposition 8.5 (Biological Repair Preserves the Selected Semantic Target)**.** *If r is a modeled biological repair of d at s, then*

$$r(d(s)) \sim_{\eta_B} s.$$

Proof. By the definition of modeled biological repair,

$$\eta_B(r(d(s))) = \eta_B(s).$$

Hence

$$r(d(s)) \sim_{\eta_B} s.$$

□

This elementary result expresses an important modeling principle: what counts as biological repair is determined by the invariant selected by the model.

Different choices of η_B may therefore induce different notions of successful repair on the same underlying biological state space.

Definition 8.6 (Biological Recoverability)**.** A damaged biological state $d(s)$ is *recoverable relative to* $\mathfrak{R}_B$ if

$$\mathrm{Rep}_{\mathfrak{R}_B}(s, d) \neq \varnothing.$$

Equivalently, there exists some modeled repair transformation r such that

$$r(d(s)) \in H_s^B.$$

Recoverability is therefore not an absolute biological property. It is relative to the modeled state space, repair transformations, semantic observables, and viability conditions.

This makes explicit a distinction that is often hidden in informal descriptions of repair:

recoverable	means	recoverable relative to a specified transformation system.

The runtime layer permits biological recovery to be represented as a trajectory rather than merely an endpoint transformation.

Definition 8.7 (Biological Regenerative Trajectory)**.** A finite execution

$$d(s) = x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} x_n$$

is a *biological regenerative trajectory relative to* s if

$$x_n \in H_s^B.$$

It is viability-preserving if

$$x_i \in V_B \qquad (0 \le i \le n).$$

This representation separates the existence of a biological recovery mechanism from the trajectory through which recovery occurs.

Two trajectories may terminate in the same semantic class while differing substantially in their intermediate states:

$$\eta_B(x_n) = \eta_B(y_m) = \eta_B(s)$$

does not imply

$$(x_0, \ldots, x_n) = (y_0, \ldots, y_m).$$

Consequently, the framework permits comparison of alternative regenerative pathways.

Definition 8.8 (Biological Repair Depth)**.** For a biological runtime model with primitive repair basis $\mathcal{Q}_B$, define

$$\delta_B(s, d) = \min\left\{|w| : w \in \mathcal{Q}_B^*,\ [\![w]\!](d(s)) \in H_s^B\right\},$$

with

$$\delta_B(s, d) = \infty$$

when no successful repair word exists.

Repair depth provides a discrete measure of the minimum number of primitive modeled repair operations required to restore the selected biological semantic target.

It should not automatically be interpreted as physical time, energetic cost, or biological complexity. Such quantities require additional observables or cost functions.

Definition 8.9 (Weighted Biological Repair Cost)**.** Let

$$c : \mathcal{Q}_B \to \mathbb{R}_{\ge 0}$$

assign a nonnegative cost to each primitive modeled repair.

For

$$w = (q_1, \ldots, q_n),$$

define

$$C(w) = \sum_{i=1}^{n} c(q_i).$$

The minimum modeled repair cost is

$$C_B^*(s, d) = \inf\left\{C(w) : [\![w]\!](d(s)) \in H_s^B\right\}.$$

This permits repair pathways to be compared using quantities other than operation count.

Depending on the model, a cost may represent computational complexity, resource consumption, duration, energetic expenditure, intervention cost, or another explicitly defined quantity.

The framework also distinguishes regeneration from compensation.

Suppose

$$C_B \subseteq V_B \setminus H_s^B$$

is a compensatory region. A damaged biological system may enter C_B and remain viable even though

$$\eta_B(x) \neq \eta_B(s).$$

Hence a model may distinguish

regeneration :	restoration of the selected biological semantic target,
compensation :	preservation of viability or function under an alternative regime.

This distinction can be important when multiple biological configurations support continued function.

The verification machinery developed above can then be applied directly to a biological runtime model.

Let

$$S_{\text{safe}}^B \subseteq V_B$$

represent states satisfying a selected biological safety specification.

A candidate repair trajectory may then be required to satisfy both

$$x_n \in H_s^B$$

and

$$x_i \in S_{\text{safe}}^B \qquad (0 \leq i \leq n).$$

Thus the formal question is not merely

Can the modeled system recover?

but

Can it recover the selected invariant through an admissible and safe trajectory?

This formulation suggests several computational problems associated with a biological runtime model:

determine whether a damaged state is recoverable,

enumerate successful repair pathways,

compute or approximate minimum repair depth,

identify compensatory regimes,

verify pathway safety,

determine which invariants survive a perturbation,

determine which invariants can be restored after perturbation.

These problems are mathematically precise once the biological state space, transformations, observables, and admissibility conditions have been specified.

The framework therefore supplies computational biology with a possible formalization strategy:

$$\boxed{\text{biological states} \longrightarrow \text{transformations} \longrightarrow \text{invariants} \longrightarrow \text{recovery pathways} \longrightarrow \text{verification.}}$$

The contribution is not a claim that biological systems are literally digital runtimes. Rather, it is that selected biological theories can be expressed as executable transformation systems whose claims about damage, repair, compensation, and regeneration become mathematically testable within the chosen model.

The next subsection considers the same framework at the level of interacting biological subsystems, where repair and regeneration become properties of systems of coupled components rather than isolated state transformations.

8.2 Systems Computation

The repair-algebra framework also suggests a computational interpretation that is broader than the modeling of any particular biological mechanism. Once biological organization is represented through states, transformations, viability conditions, semantic observables, and recovery criteria, the resulting structure defines a class of computations over organized systems.

We call this perspective *systems computation*.

Systems computation treats the evolution of an organized system as computation over a structured state space in which successful execution is constrained not only by transition rules, but by invariants governing viability, semantics, and recoverability.

Let

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi)$$

be a regenerative runtime.

An execution

$$\pi = (s_0, s_1, \ldots, s_n)$$

is then a computation whose intermediate and terminal states may be evaluated relative to the structural predicates of the runtime.

This differs from an ordinary input-output interpretation of computation. The significance of an execution is not determined solely by its terminal state. Intermediate viability, semantic preservation, repair availability, and recoverability may all contribute to whether the computation is considered successful.

Definition 8.10 (System Computation)**.** A *system computation* in a regenerative runtime $\mathfrak{R}$ is a finite or infinite policy-admissible execution

$$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$$

whose states are interpreted relative to the viability region V, the semantic map η, and the transformation structure of the runtime.

The defining feature of systems computation is therefore that execution occurs inside an organized semantic state space.

A conventional computation may be modeled abstractly as

$$x \mapsto f(x).$$

A system computation instead has the form

$$s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} s_n,$$

together with predicates describing what must remain true, what may change, and what must eventually be restored.

For example, a viable system computation may require

$$s_i \in V$$

for every state of the execution.

A semantics-preserving computation may additionally require

$$\eta(s_i) = \eta(s_0)$$

for every i.

A regenerative computation permits temporary semantic displacement but requires eventual restoration:

$$\eta(s_k) \neq \eta(s_0)$$

for some intermediate state s_k, while

$$\eta(s_n) = \eta(s_0).$$

Thus maintenance and regeneration correspond to different computational regimes.

Definition 8.11 (Maintenance Computation)**.** A system computation

$$\pi = (s_0, \ldots, s_n)$$

is a *maintenance computation* relative to a region $H \subseteq S$ if

$$s_i \in H$$

for every

$$0 \leq i \leq n.$$

Definition 8.12 (Regenerative Computation)**.** A system computation

$$\pi = (s_0, \ldots, s_n)$$

is a *regenerative computation* relative to H if the execution contains a displaced state

$$s_k \notin H$$

and subsequently reaches

$$s_n \in H.$$

The computational distinction is fundamental.

Maintenance computes while remaining inside the designated organizational region. Regeneration computes a path back to that region after displacement.

The repair algebra supplies the transformation language for the second problem.

Let

$$d \in \mathcal{D}$$

be a damage transformation and let

$$w = (r_1, \ldots, r_m)$$

be a repair word.

The corresponding system computation is

$$s \xrightarrow{d} d(s) \xrightarrow{r_1} s_1 \xrightarrow{r_2} \cdots \xrightarrow{r_m} s_m.$$

If

$$s_m \in V$$

and

$$\eta(s_m) = \eta(s),$$

then the computation realizes semantic recovery.

The runtime is therefore solving a constrained reachability problem:

find an admissible execution from a displaced state to a viable state in the required semantic class.

This interpretation converts repair from a descriptive biological concept into a computational problem.

Given

$$(s, d),$$

one may ask whether there exists a primitive repair word

$$w \in \mathcal{P}_{\mathcal{R}}^*$$

such that

$$[\![w]\!](d(s)) \in V \cap [s]_\eta.$$

If such a word exists, the damaged state is recoverable, provided that the primitive repair basis generates the available repair monoid.

If several successful repair pathways exist, the runtime may face an optimization problem. It may seek a pathway minimizing repair depth, execution cost, biological risk, elapsed time, resource consumption, or some other objective.

Let

$$c : \mathcal{P}_{\mathcal{R}}^* \to \mathbb{R}_{\geq 0}$$

be a repair-cost function.

A minimum-cost recovery problem has the form

$$\min_{w \in \mathrm{Path}_{\mathcal{A}}(s,d)} c(w).$$

Repair depth is recovered as the special case

$$c(w) = |w|.$$

Systems computation therefore contains both decision and optimization problems.

The decision problem asks

$$\mathrm{Path}_{\mathcal{A}}(s, d) \neq \varnothing?$$

The optimization problem asks which member of this set should be executed.

This distinction is especially important when multiple repair pathways produce semantically equivalent outcomes.

Suppose

$$w_1, w_2 \in \mathrm{Path}_{\mathcal{A}}(s, d)$$

satisfy

$$\eta([\![w_1]\!](d(s))) = \eta([\![w_2]\!](d(s))) = \eta(s).$$

The terminal states need not be identical:

$$[\![w_1]\!](d(s)) \neq [\![w_2]\!](d(s)).$$

The computations are nevertheless equivalent relative to the selected semantic observable.

Definition 8.13 (Semantically Equivalent System Computations)**.** Two terminating system computations with terminal states x and y are *semantically equivalent* relative to η when

$$\eta(x) = \eta(y).$$

This permits a regenerative runtime to admit multiple implementations of the same semantic recovery.

Systems computation also distinguishes endpoint correctness from trajectory correctness.

A computation may terminate in the desired region while passing through an inadmissible intermediate state.

Thus

$$s_n \in V \cap [s_0]_\eta$$

does not imply that

$$s_i \in V$$

for every intermediate state.

A stronger computation requires

$$s_i \in V \qquad \text{for all } i,$$

together with the required terminal condition.

This yields two independent computational obligations:

reach the correct state class	and	remain within admissible regions while reaching it.

The first is a reachability property.

The second is a safety property.

Their combination gives verified regenerative computation.

Definition 8.14 (Verified Regenerative Computation)**.** A finite system computation

$$\pi = (s_0, \ldots, s_n)$$

is a *verified regenerative computation* relative to an initial semantic reference state s if:

1. every transition is operationally admissible;
2. every required safety condition holds throughout the execution;
3. the terminal state is viable,
$$s_n \in V;$$
4. the required semantics are restored,
$$\eta(s_n) = \eta(s).$$

This formulation provides a direct connection between repair algebra, runtime semantics, and executable verification.

At the algebraic level, the theory determines which repair transformations exist and how they compose.

At the runtime level, it determines which trajectories those transformations induce.

At the computational level, it asks whether those trajectories can be found, executed, optimized, and verified.

Systems computation therefore provides a bridge between the mathematical theory of repair and the analysis of complex biological systems.

The central computational object is no longer merely a function from an input to an output. It is an organized trajectory whose correctness is defined by invariants across transformation:

computation = state transformation constrained by semantic and structural invariants.

For regenerative systems, this becomes

regenerative computation = the computation of an admissible path that restores designated invariants after pert

This perspective scales naturally from individual repair operations to interacting collections of biological subsystems. The next subsection develops that systems-level interpretation.

8.3 Systems Biology

The preceding computational-biological interpretation treats repair and regeneration as transformations of a modeled biological state. Systems biology introduces an additional structural feature: biological organization is typically distributed across interacting subsystems.

A regenerative system may therefore preserve or recover global organization through transformations that occur locally, propagate between components, or compensate for failure in one subsystem through changes in another.

The repair-algebra framework provides a natural language for distinguishing local repair from global regeneration.

Definition 8.15 (Composite Biological System)**.** Let

$$\mathcal{S} = \{S_1, \ldots, S_n\}$$

be a finite family of subsystem state spaces.

The associated global state space is

$$S \subseteq \prod_{i=1}^{n} S_i,$$

where the restriction to S may encode compatibility or interaction constraints between subsystem states.

A global state is written

$$x = (x_1, \ldots, x_n).$$

The direct-product representation is intentionally general. It does not require the subsystems to evolve independently. Coupling constraints may make only a proper subset of the Cartesian product biologically admissible.

Definition 8.16 (Subsystem Projection)**.** For each subsystem i, define the projection

$$\pi_i : S \to S_i$$

by

$$\pi_i(x_1, \ldots, x_n) = x_i.$$

Subsystem observables may then be defined by maps

$$\eta_i : S_i \to \Omega_i,$$

while the global system possesses a semantic observation map

$$\eta : S \to \Omega.$$

The global semantic signature need not be determined by any single subsystem.

Definition 8.17 (Global Viability)**.** Let

$$V_i \subseteq S_i$$

be subsystem viability regions.

A global viability region

$$V \subseteq S$$

specifies the states regarded as viable at the system level.

In general,

$$V \neq \prod_{i=1}^{n} V_i.$$

A system may therefore exhibit global viability even when some subsystem is outside its preferred local region, or global failure even when every subsystem considered individually satisfies a local viability condition.

This distinction permits compensatory and interaction-dependent behavior.

Definition 8.18 (Local Transformation)**.** A runtime transformation

$$\tau : S \rightharpoonup S$$

is *local to subsystem* i if, whenever $\tau(x)$ is defined,

$$\pi_j(\tau(x)) = \pi_j(x) \qquad \text{for every } j \neq i.$$

A transformation affecting several components simultaneously is called a *coupled transformation*.

Thus repair transformations may be classified according to the subsystem support on which they act.

Definition 8.19 (Support of a Transformation)**.** For a transformation

$$\tau : S \rightharpoonup S,$$

define its support at x by

$$\operatorname{supp}_x(\tau) = \{i : \pi_i(\tau(x)) \neq \pi_i(x)\}.$$

When this set is independent of x on a region $U \subseteq S$, write

$$\operatorname{supp}_U(\tau)$$

for the corresponding transformation support.

This notion extends the support structure used earlier for primitive repair operations and makes interaction between repairs explicit at the subsystem level.

Definition 8.20 (Local Repair)**.** Let

$$r \in \mathcal{R}.$$

The repair r is *local to subsystem* i when

$$\operatorname{supp}(r) \subseteq \{i\}.$$

It is a *distributed repair* when

$$|\operatorname{supp}(r)| > 1.$$

A local repair may nevertheless produce a global semantic effect if the repaired subsystem participates in the global semantic observable.

Definition 8.21 (Subsystem Semantic Equivalence)**.** For states

$$x, y \in S,$$

define

$$x \sim_i y$$

when

$$\eta_i(\pi_i(x)) = \eta_i(\pi_i(y)).$$

Global semantic equivalence remains

$$x \sim_\eta y \iff \eta(x) = \eta(y).$$

Local and global semantic equivalence need not coincide.

Proposition 8.22 (Local Semantic Restoration Need Not Imply Global Restoration)**.** *There may exist states* $x, y \in S$ *such that*

$$x \sim_i y$$

for a subsystem i*, while*

$$x \not\sim_\eta y.$$

Proof. The local equivalence relation constrains only the observable

$$\eta_i \circ \pi_i.$$

The global semantic map η may depend on additional subsystems or on relations between subsystems. Equality of the local observable therefore does not imply equality of the global semantic signature. □

This identifies a central systems-level distinction:

$$\boxed{\text{local repair } \not\Rightarrow \text{ global regeneration.}}$$

A subsystem may be restored while the organization of the complete system remains damaged. The converse phenomenon is also possible.

Definition 8.23 (Global Compensation)**.** Let $s \in V$ be a reference state and let $x \in V$.

The state x exhibits *global compensation relative to* s if the global system preserves a designated functional observable

$$\eta_F(x) = \eta_F(s)$$

while at least one subsystem differs from its reference semantic state:

$$\eta_i(\pi_i(x)) \neq \eta_i(\pi_i(s))$$

for some i.

Thus global function may survive local semantic change.

Proposition 8.24 (Global Function Can Be Preserved under Local Change)**.** *Suppose the global functional observable*

$$\eta_F : S \to \Omega_F$$

is invariant under a transformation τ*. Then*

$$\eta_F(\tau(x)) = \eta_F(x)$$

even if

$$\pi_i(\tau(x)) \neq \pi_i(x)$$

for one or more subsystems i*.*

Proof. The first equality follows directly from invariance of η_F under τ. The definition of invariance imposes no requirement that the individual subsystem coordinates remain unchanged. □

This provides a formal representation of compensatory redistribution: internal organization may change while a selected global function remains invariant.

Definition 8.25 (Interaction Graph)**.** Let

$$G = (\{1, \ldots, n\}, E)$$

be a graph whose vertices represent subsystems.

An edge

$$(i, j) \in E$$

indicates that the modeled dynamics of subsystem i may directly affect subsystem j, or conversely according to the chosen graph convention.

The graph G is called the *interaction graph* of the systems model.

The interaction graph permits damage and repair propagation to be studied structurally.

Definition 8.26 (Damage Propagation)**.** Let

$$d \in \mathcal{D}$$

be a damage transformation.

Damage is said to *propagate from subsystem* i *to subsystem* j along an execution

$$x_0 \rightsquigarrow x_m$$

if the initial perturbation affects subsystem i and some subsequent state exhibits a damage-associated change in subsystem j.

Likewise, repair effects may propagate across the interaction graph.

Definition 8.27 (Repair Propagation)**.** A repair initiated on subsystem i exhibits *repair propagation to subsystem j* if a runtime execution containing that repair subsequently produces a designated restorative change in subsystem j.

These definitions distinguish the support of an individual transformation from the larger region of the system influenced by its runtime consequences.

Definition 8.28 (System-Level Regeneration)**.** Let

$$H_s = V \cap [s]_\eta$$

be the global homeostatic region associated with a reference state s.

A systems-level execution

$$d(s) = x_0 \rightsquigarrow x_n$$

is *globally regenerative* if

$$x_n \in H_s.$$

Global regeneration therefore requires restoration of the selected global semantic invariant rather than independent restoration of every subsystem.

Proposition 8.29 (Subsystem Identity Is Not Required for Global Regeneration)**.** *A globally regenerative execution may terminate at a state x_n for which*

$$\pi_i(x_n) \neq \pi_i(s)$$

for one or more subsystems i.

Proof. Global regeneration requires

$$x_n \in V$$

and

$$\eta(x_n) = \eta(s).$$

Neither condition requires coordinatewise equality

$$\pi_i(x_n) = \pi_i(s).$$

Hence global semantic restoration may occur without exact restoration of every subsystem state. □

This is the systems-level form of semantic regeneration:

global organization can be restored without microscopic or componentwise restoration.

The repair algebra also permits interaction between local repair operations to be analyzed.

Let

$$r_i, r_j \in \mathcal{R}$$

be repairs associated with different subsystem supports.

If

$$r_i \circ r_j = r_j \circ r_i,$$

their effective repair is independent of execution order on the region where commutativity holds.

If instead

$$r_i \circ r_j \neq r_j \circ r_i,$$

then subsystem interaction induces order-sensitive repair behavior.

Definition 8.30 (Repair Interaction)**.** Two repairs

$$r, q \in \mathcal{R}$$

interact on $U \subseteq S$ if

$$r(q(x)) \neq q(r(x))$$

for at least one

$$x \in U.$$

Repair interaction therefore measures an operational failure of commutativity.

Proposition 8.31 (Disjoint Support Does Not Alone Guarantee Commutativity)**.** *Repairs acting on disjoint subsystem coordinates need not commute if their actions depend on shared global state.*

Proof. Let r modify only subsystem i and q modify only subsystem j, with $i \neq j$. Although their output supports are disjoint, the value assigned by r may depend on the current state of subsystem j, while the value assigned by q may depend on subsystem i.

Applying one transformation can therefore alter information used by the other. Consequently,

$$r(q(x))$$

and

$$q(r(x))$$

may differ. □

Thus independence of repair requires more than disjoint output support; it also requires suitable independence of the transformations' state dependencies.

This observation connects systems structure directly to the algebra of repair.

Definition 8.32 (Subsystem Recovery Profile)**.** For a damaged state x, define its *subsystem recovery profile* by

$$\mathbf{R}(x) = (R_1(x), \ldots, R_n(x)),$$

where

$$R_i(x) = \begin{cases} 1, & \text{if subsystem } i \text{ satisfies its designated recovery condition,} \\ 0, & \text{otherwise.} \end{cases}$$

The profile records local recovery while the global semantic observable records system-level recovery.

Consequently, two states may satisfy

$$\mathbf{R}(x) \neq \mathbf{R}(y)$$

while

$$\eta(x) = \eta(y).$$

The distinction provides a formal way to separate component-level organization from emergent system-level organization.

Definition 8.33 (Systemic Repair Path)**.** A *systemic repair path* is a runtime execution

$$x_0 \xrightarrow{\tau_1} x_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_m} x_m$$

whose transitions may act on different subsystem supports and whose terminal state satisfies a designated global recovery condition.

A systemic repair path can therefore combine local repair, distributed repair, adaptation, compensation, and ordinary runtime transitions.

Its analysis may involve several distinct quantities:

path length,
number of affected subsystems,
repair depth,
interaction cost,
safety constraints,
global semantic restoration.

The resulting optimization problem is not merely to identify a collection of local repairs, but to identify a globally successful admissible trajectory through an interacting system.

The framework therefore suggests a systems-biological interpretation of regeneration:

local perturbations
↓
interacting subsystem dynamics
↓
repair, adaptation, and compensation
↓
global invariant preservation or recovery.

Under this interpretation, the central mathematical question is not whether every component returns to its original state. It is which system-level properties survive the transformations of the interacting system and which can be recovered when they are lost.

Repair algebras provide the compositional structure for those transformations. Regenerative runtimes provide their execution semantics. Systems biology supplies a natural class of coupled state spaces in which local transformations and global invariants may differ substantially.

The next subsection considers how such formal models could potentially be used to structure and verify computational reasoning in medical decision-support systems.

8.4 Medical Decision Support

The formal distinction between damage, viability, compensation, repair, and regeneration also suggests a potential application to medical decision support. In this setting, the repair-algebra framework is not interpreted as an autonomous medical decision-maker. Rather, it provides a mathematical language for representing candidate interventions, recovery pathways, constraints, and desired outcomes.

Let

$$S$$

denote a clinically relevant state space and let

$$V \subseteq S$$

denote the region of states regarded as viable relative to a specified model.

A patient state may be represented abstractly by

$$s \in S,$$

while a perturbation, injury, pathological process, or other modeled disturbance may be represented by a transformation

$$d \in \mathcal{D}.$$

The resulting state is

$$d(s).$$

Potential interventions may then be represented by transformations

$$r \in \mathcal{R}.$$

The interpretation of r is deliberately general. Depending on the model, it could represent a therapeutic intervention, supportive process, rehabilitative action, compensatory mechanism, or other state-changing operation.

The central computational question becomes whether an admissible sequence of such transformations can move the system from its current state into a designated target region.

Definition 8.34 (Clinical Target Region)**.** Let

$$G \subseteq V$$

be a set of viable states satisfying the clinical objectives encoded by a particular model.

We call G a *clinical target region.*

The target region need not consist of a single state.

This is important because successful treatment does not generally require restoration of an exact previous physical configuration. Multiple states may satisfy the same relevant clinical objective.

If

$$\eta : S \to \Omega$$

represents the selected clinical or functional semantics, then a target region relative to a reference state s may take the form

$$G_s = V \cap [s]_\eta.$$

Recovery then requires reaching some

$$s' \in G_s,$$

rather than necessarily requiring

$$s' = s.$$

Thus the framework distinguishes exact restoration from functional or semantic restoration.

A candidate intervention sequence

$$w = (r_1, \ldots, r_n)$$

produces the trajectory

$$d(s) \xrightarrow{r_1} s_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} s_n.$$

The sequence is successful relative to G when

$$s_n \in G.$$

However, endpoint success alone is insufficient for medical decision support. Intermediate states may also be constrained.

Let

$$K \subseteq S$$

denote a designated safe region.

Definition 8.35 (Admissible Intervention Pathway)**.** An intervention pathway

$$\pi = (s_0, \dots, s_n)$$

is *admissible relative to* K when

$$s_i \in K$$

for every

$$0 \le i \le n.$$

A candidate pathway may therefore reach the desired target while still being rejected because one or more intermediate states violate the modeled safety constraints.

This gives the decision problem two distinct components:

target attainment	and	trajectory admissibility.

The first asks whether the intervention reaches the desired state class.

The second asks whether the route taken to that state remains within the permitted region.

This distinction is particularly important when interventions interact.

Suppose

$$r_1, r_2 \in \mathcal{R}.$$

In general,

$$r_2 \circ r_1 \neq r_1 \circ r_2.$$

Consequently, the same interventions performed in different orders may produce different trajectories or terminal states.

Even when

$$r_2 \circ r_1(s) = r_1 \circ r_2(s),$$

the intermediate states may differ.

Therefore equality of terminal outcomes does not imply equality of intervention pathways.

This provides a formal reason to represent treatment plans as trajectories rather than unordered collections of actions.

Medical decision support may also involve several competing objectives.

Let

$$c_j : \mathcal{R}^* \to \mathbb{R}_{\ge 0}$$

represent costs associated with a candidate intervention pathway.

These may encode modeled quantities such as intervention burden, resource use, pathway length, risk, or other application-specific criteria.

A decision-support system may then seek

$$w^* \in \arg\min_{w \in \mathcal{W}_G} c(w),$$

where

$$\mathcal{W}_G$$

is the set of admissible pathways terminating in the target region.

For multiple objectives, one may instead associate a cost vector

$$C(w) = (c_1(w), \dots, c_k(w)).$$

The problem then becomes one of constrained multi-objective optimization rather than simple recovery detection.

The repair-algebra perspective is useful because it separates three questions that are easily conflated:

1. Is recovery structurally possible?
2. Which admissible pathways can realize that recovery?
3. Which pathway is preferable under the selected optimization criteria?

The first is a recoverability problem.

The second is a constrained reachability problem.

The third is an optimization problem.

These distinctions permit different forms of evidence to enter at different levels of the model.

For example, a transformation may be excluded because it is not available, because it violates an admissibility condition, because it cannot reach the target region, or because another valid pathway is preferred under the chosen cost function.

The framework also permits uncertainty to be incorporated by replacing deterministic transformations with families or relations of possible successor states.

For a nondeterministic intervention r, one may write

$$r(s) \subseteq S.$$

A robust intervention criterion could then require

$$r(s) \subseteq K$$

or, more strongly,

$$r(s) \subseteq G,$$

depending on the intended guarantee.

Probabilistic extensions could associate transition probabilities with possible successor states, although such models lie beyond the deterministic runtime developed in the present paper.

The deterministic theory therefore supplies a structural foundation upon which richer decision models may later be constructed.

A further application is explanation.

Because a candidate pathway is represented explicitly as

$$s_0 \xrightarrow{r_1} s_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} s_n,$$

the system can associate each proposed transformation with its modeled effect, the invariant it preserves or restores, and the condition that justifies its inclusion.

A decision-support output may therefore be accompanied by a formal certificate containing:

- the initial modeled state;
- the target region;
- the proposed intervention sequence;
- the predicted intermediate states;

- the viability and safety conditions checked along the pathway;
- the terminal semantic condition;
- the optimization criterion used to compare alternatives.

Such a certificate does not establish that the underlying biological model is medically correct. It establishes that the proposed conclusion follows from the formal model and assumptions supplied to the runtime.

This distinction is essential.

verification of a medical model $\neq$ validation of the medical assumptions encoded by that model.

Formal verification can determine whether an intervention pathway satisfies specified mathematical conditions. It cannot, by itself, establish that the state representation, biological assumptions, transition model, or clinical criteria accurately describe a real patient.

Accordingly, the framework is best interpreted as a possible mathematical substrate for decision support rather than as a replacement for empirical medicine or clinical judgment.

Its contribution is structural: it provides a language in which recovery, compensation, intervention composition, safety, and target attainment can be expressed within a single transition system.

Under this interpretation, medical decision support becomes an instance of verified systems computation:

current state $\longrightarrow$ candidate interventions $\longrightarrow$ admissible trajectories $\longrightarrow$ verified target attainment.

The same formal machinery can therefore support a broader question: whether biological claims about repair and regeneration can themselves be subjected to machine-checkable verification. The next subsection develops this possibility.

8.5 Formal Biological Verification

The runtime framework developed in this paper makes it possible to formulate biological repair and regeneration as explicit verification problems.

Once biological states, transformations, viability conditions, semantic observables, and recovery criteria are represented mathematically, claims about system behavior can be expressed as propositions over runtime executions.

Let

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi)$$

be a regenerative runtime.

A biological verification problem asks whether a specified property

$$P$$

holds for a state, transformation, execution, or class of executions generated by $\mathfrak{R}$.

The property may concern viability, semantic preservation, recoverability, safety, termination, repair correctness, or successful regeneration.

Definition 8.36 (Biological Verification Property)**.** A *biological verification property* is a predicate

$$P : S^* \to \{\mathsf{true}, \mathsf{false}\}$$

defined over finite runtime executions.

An execution

$$\pi = (s_0, \ldots, s_n)$$

satisfies P when

$$P(\pi) = \mathsf{true}.$$

This representation permits familiar verification questions to be stated directly in terms of regenerative behavior.

For example, viability preservation can be expressed as

$$P_V(\pi) \iff \forall i,\ s_i \in V.$$

Semantic preservation relative to the initial state can be expressed as

$$P_\eta(\pi) \iff \forall i,\ \eta(s_i) = \eta(s_0).$$

Semantic recovery can instead be expressed by the terminal condition

$$P_{\mathrm{rec}}(\pi) \iff \eta(s_n) = \eta(s_0).$$

If

$$H \subseteq V$$

is a designated homeostatic region, regeneration may be expressed as

$$P_H(\pi) \iff s_n \in H,$$

subject to the additional requirement that the execution begins from or passes through a displaced state outside H.

These predicates distinguish several verification objectives that may otherwise appear superficially similar.

A system may preserve viability without preserving semantics:

$$P_V(\pi) = \mathsf{true}, \qquad P_\eta(\pi) = \mathsf{false}.$$

It may fail to preserve semantics continuously while nevertheless restoring them:

$$P_\eta(\pi) = \mathsf{false}, \qquad P_{\mathrm{rec}}(\pi) = \mathsf{true}.$$

Thus verification must specify precisely which property is intended.

Definition 8.37 (Verified Biological Execution)**.** A runtime execution

$$\pi = (s_0, \ldots, s_n)$$

is *verified relative to* P when the execution is admissible under the runtime transition structure and

$$P(\pi) = \mathsf{true}.$$

A verified regenerative execution requires stronger conditions.

Definition 8.38 (Verified Regenerative Execution)**.** Let

$$H \subseteq V$$

be a designated homeostatic region.

An execution

$$\pi = (s_0, \dots, s_n)$$

is a *verified regenerative execution* when:

1. every transition is admissible;
2. all designated safety conditions hold throughout the execution;
3. the execution contains a state outside H;
4. the terminal state satisfies
$$s_n \in H.$$

If regeneration is defined relative to the semantics of a reference state s, the terminal condition may be strengthened to

$$s_n \in V \cap [s]_\eta.$$

Thus

$$s_n \in V$$

and

$$\eta(s_n) = \eta(s).$$

This verifies semantic regeneration rather than merely return to a broad homeostatic region.

The verification problem may concern an individual execution or every possible execution generated under a policy.

Let

$$\mathrm{Exec}(\mathfrak{R}, \Pi, s)$$

denote the set of executions beginning at s that are admissible under policy Π.

A universal verification problem asks whether

$$\forall \pi \in \mathrm{Exec}(\mathfrak{R}, \Pi, s), \qquad P(\pi).$$

An existential verification problem instead asks whether

$$\exists \pi \in \mathrm{Exec}(\mathfrak{R}, \Pi, s)$$

such that

$$P(\pi).$$

The distinction is especially important for recovery.

Existential recoverability asks whether at least one successful recovery path exists.

Universal policy recovery asks whether every execution permitted by the policy eventually reaches the designated recovery region.

Definition 8.39 (Existential Regenerative Verification)**.** A state x is *existentially verified as regenerative* relative to H when there exists a policy-admissible execution

$$x = s_0 \longrightarrow \cdots \longrightarrow s_n$$

such that

$$s_n \in H.$$

Definition 8.40 (Universal Regenerative Verification)**.** A state x is *universally verified as regenerative under policy* Π when every maximal policy-admissible execution beginning at x reaches H.

Universal regenerative verification is therefore stronger than structural recoverability.

Proposition 8.41 (Universal Regeneration Implies Existential Regeneration)**.** *Suppose at least one policy-admissible execution begins at x.*

If x is universally verified as regenerative under Π, then x is existentially verified as regenerative.

Proof. By assumption, there exists at least one policy-admissible execution beginning at x.

Universal regenerative verification requires every such maximal execution to reach H.

Therefore at least one execution reaches H, establishing existential regeneration. □

The converse does not generally hold.

A runtime may contain one successful recovery path together with other paths leading to persistent displacement or failure.

Consequently,

$$\boxed{\text{recoverability} \neq \text{guaranteed recovery.}}$$

This distinction provides a direct connection between the algebraic and operational layers of the theory.

Repair algebra establishes whether successful repair transformations exist.

Runtime verification determines whether admissible execution realizes those transformations while satisfying the required invariants and safety conditions.

Verification can also be expressed through certificates.

Definition 8.42 (Biological Verification Certificate)**.** A *biological verification certificate* for an execution

$$\pi = (s_0, \ldots, s_n)$$

is finite evidence sufficient to establish the designated verification conditions for π.

Depending on the property, the certificate may contain:

- the encoded runtime states;
- the transformations applied between states;
- evidence of transition admissibility;
- evaluations of viability predicates;
- evaluations of semantic observables;
- safety checks for intermediate states;

- the terminal recovery condition.

For a repair word

$$w = (r_1, \dots, r_n),$$

a certificate may therefore record the trace

$$d(s) \xrightarrow{r_1} s_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} s_n$$

together with proofs or machine-checkable evidence that

$$s_i \in K$$

for every required safe region K, and

$$s_n \in V \cap [s]_\eta.$$

Proposition 8.43 (Soundness of a Regenerative Certificate)**.** *Suppose a certificate establishes:*

$$s_{i+1} = r_{i+1}(s_i)$$

for every transition in the repair trace,

$$s_i \in K$$

for every intermediate state,
and

$$s_n \in V \cap [s]_\eta.$$

Then the certified execution is trace-safe relative to K and semantically regenerative relative to s.

Proof. The transition equalities establish that the certificate corresponds to the claimed repair execution.

The conditions

$$s_i \in K$$

establish trace safety.

Finally,

$$s_n \in V \cap [s]_\eta$$

implies

$$s_n \in V$$

and

$$\eta(s_n) = \eta(s).$$

Hence the execution terminates in a viable state semantically equivalent to the reference state. □

The executable-encoding architecture introduced earlier provides a possible machine representation for such certificates.

Suppose

$$\mathrm{enc} : S \to C$$

encodes runtime states and each runtime transformation τ has a correct encoded realization

$$\widehat{\tau} : C \rightharpoonup C.$$

A machine may then evaluate a trace

$$\mathrm{enc}(s_0) \xrightarrow{\widehat{\tau}_1} c_1 \xrightarrow{\widehat{\tau}_2} \cdots \xrightarrow{\widehat{\tau}_n} c_n$$

and verify that decoding produces the intended abstract execution.

If

$$\mathrm{dec}(c_i) = s_i,$$

then abstract verification predicates can be evaluated against the decoded states or against verified encoded realizations of those predicates.

The complete verification architecture therefore has the form

biological model $\longrightarrow$ formal state semantics $\longrightarrow$ executable representation $\longrightarrow$ runtime execution $\longrightarrow$ verification

This architecture exposes an important boundary.

Formal verification establishes consequences of the mathematical model. It does not establish that the model itself is an empirically accurate description of a biological system.

If

$$M$$

denotes the formal model and

$$P$$

a verified property, then formal verification establishes a statement of the form

$$M \models P.$$

It does not, without independent empirical evidence, establish that

$$M$$

faithfully represents the biological system to which it is applied.

Accordingly, two different forms of validation must be distinguished:

formal correctness of the model's consequences	and	empirical adequacy of the model itself.

The present framework addresses the first problem.

Its potential significance is that biological claims concerning preservation, repair, compensation, recovery, and regeneration can be stated with enough mathematical precision to become objects of formal verification.

In this sense, the framework converts a qualitative question,

"Does the system regenerate?"

into a collection of explicit verification questions:

Which invariants define successful recovery?

Which perturbations violate those invariants?

Which repair transformations can restore them?

Which recovery trajectories remain admissible?

Does the runtime policy guarantee that recovery occurs?

The resulting perspective can be summarized as

regeneration becomes verifiable when its defining invariants, transformations, and recovery conditions are made

The final application considered in this section extends this architecture from verification of individual runtime executions toward compiler systems capable of translating high-level regenerative specifications into executable and verifiable runtime machinery.

8.6 Future Compiler Architectures

The preceding framework suggests a further computational direction: the construction of compilers that translate high-level specifications of biological organization, repair, and regeneration into executable runtime models.

Such a compiler would differ from a conventional programming-language compiler. Its source language would describe states, viability conditions, semantic invariants, damage transformations, repair operations, and recovery objectives. Its target would be an executable regenerative runtime together with the verification conditions required to establish correctness.

The conceptual compilation pipeline may be represented as

regenerative specification $\longrightarrow$ repair algebra $\longrightarrow$ runtime transition system $\longrightarrow$ executable encoding $\longrightarrow$ verified

Each stage lowers the description into a more operational representation while preserving the semantic structure required by the preceding level.

Definition 8.44 (Regenerative Specification)**.** A *regenerative specification* is a tuple

$$\mathcal{G} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{I}, \mathcal{C}),$$

where:

- S is a specified state space;
- $V \subseteq S$ is the viability region;
- Ω is a semantic observation space;
- $$\eta : S \to \Omega$$ is a semantic observation map;
- $\mathcal{D}$ specifies admissible damage transformations;
- $\mathcal{R}$ specifies candidate repair transformations;
- $\mathcal{I}$ is a collection of required invariants;
- $\mathcal{C}$ is a collection of operational, safety, or recovery constraints.

The compiler's first task is structural.

From the regenerative specification, it must construct or verify a repair algebra

$$\mathcal{A} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \circ, \mathrm{id}_S)$$

whose operations satisfy the algebraic requirements of the source specification.

The second task is operational.

The compiler must determine how the abstract transformations are realized as runtime transitions. This produces a regenerative runtime

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi),$$

where $\mathcal{T}$ provides the transition structure and Π provides the runtime policy.

The third task is representational.

Runtime states and transformations must be lowered into an executable representation space

$$C.$$

This requires an encoding

$$\mathrm{enc} : S \to C$$

and a corresponding decoding operation

$$\mathrm{dec} : C \rightharpoonup S.$$

Correct representation requires

$$\mathrm{dec}(\mathrm{enc}(s)) = s.$$

For every compiled runtime transformation

$$\tau : S \rightharpoonup S,$$

the compiler must generate an encoded realization

$$\widehat{\tau} : C \rightharpoonup C$$

satisfying

$$\mathrm{dec}\,(\widehat{\tau}(\mathrm{enc}(s))) = \tau(s).$$

The compilation process therefore contains a sequence of correctness obligations.

Definition 8.45 (Semantics-Preserving Regenerative Compilation)**.** A compilation

$$\mathcal{G} \rightsquigarrow \mathfrak{R} \rightsquigarrow \widehat{\mathfrak{R}}$$

is *semantics-preserving* when every source-level state, transformation, and designated invariant has a corresponding target-level realization whose execution preserves the semantic relationships specified by $\mathcal{G}$.

At minimum, such a compiler should preserve semantic equivalence.

If

$$s \sim_\eta t$$

at the source level, then their compiled representations should satisfy the corresponding target-level semantic relation.

Likewise, if a source-level repair word

$$w = (r_1, \ldots, r_n)$$

successfully repairs damage d at state s, then its compiled execution should terminate in a representation of a state

$$s' \in V \cap [s]_\eta.$$

Theorem 8.46 (Compiled Repair Correctness)**.** *Suppose a compiler produces correct encoded realizations*

$$\widehat{d}, \widehat{r}_1, \ldots, \widehat{r}_n$$

of

$$d, r_1, \ldots, r_n.$$

Let

$$w = (r_1, \ldots, r_n)$$

be a successful repair pathway for s, so that

$$[\![w]\!](d(s)) \in V \cap [s]_\eta.$$

Then

$$\operatorname{dec}\left(\widehat{r}_n \circ \cdots \circ \widehat{r}_1 \circ \widehat{d}\,(\operatorname{enc}(s))\right) \in V \cap [s]_\eta.$$

Proof. Correctness of the encoded realizations gives

$$\operatorname{dec}\left(\widehat{r}_n \circ \cdots \circ \widehat{r}_1 \circ \widehat{d}\,(\operatorname{enc}(s))\right) = [\![w]\!](d(s)).$$

Since w is a successful repair pathway,

$$[\![w]\!](d(s)) \in V \cap [s]_\eta.$$

The result follows. □

This theorem expresses the central requirement of a regenerative compiler: successful repair at the semantic level must remain successful after compilation into executable machinery.

The same principle applies to complete trajectories.

Suppose a source-level execution is

$$s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} s_n.$$

The compiled execution should produce

$$c_0 \xrightarrow{\widehat{\tau}_1} c_1 \xrightarrow{\widehat{\tau}_2} \cdots \xrightarrow{\widehat{\tau}_n} c_n$$

such that

$$\operatorname{dec}(c_i) = s_i$$

for every i.

This is stronger than terminal correctness. It establishes correspondence throughout execution.

Such correspondence permits source-level safety properties to descend to the executable runtime.

Let

$$K \subseteq S$$

be a safe region.

If the source execution satisfies

$$s_i \in K$$

for every i, then a correct compiled realization satisfies

$$\operatorname{dec}(c_i) \in K$$

for every i.

Thus trace safety becomes a compilation invariant.

The compiler may also generate verification obligations rather than merely executable code.

For a repair pathway, these obligations may include

$$s_i \in V,$$

$$s_i \in K,$$

$$\eta(s_n) = \eta(s_0),$$

and

$$s_n \in H$$

for appropriate viability, safety, semantic, and homeostatic regions.

The resulting compiler therefore produces two related outputs:

executable runtime + proof obligations.

A stronger architecture may additionally generate machine-checkable certificates establishing those obligations.

The compilation pipeline would then take the form

high-level regenerative specification
↓
repair algebra
↓
regenerative runtime
↓
executable encoding
↓
machine execution + verification certificate.

This architecture provides a concrete interpretation of semantic descent.

At each stage, representational details become more explicit while the compiler carries forward the invariants required by the source theory.

A failure of compilation may therefore have mathematical significance.

For example, compilation may fail because:

- a required repair transformation has no executable realization;
- a semantic invariant cannot be represented at the target level;
- a repair pathway violates a safety condition;
- the runtime policy does not guarantee recovery;
- the target representation collapses distinctions required by the source semantics;
- no admissible recovery pathway exists for a specified damage condition.

Compilation is therefore not merely translation. It may also act as a test of whether a regenerative theory has been specified with sufficient operational precision to support execution.

This motivates a stronger interpretation:

execution is a test of operational semantic completeness.

Here, operational semantic completeness does not refer to logical completeness in the classical proof-theoretic sense. It means that the distinctions required to execute and verify the modeled behavior have been made mathematically explicit.

A compiler cannot operate directly on an informal notion such as "regeneration."

It requires a state representation, a definition of admissible change, a criterion for viability, a specification of the relevant invariants, a representation of repair transformations, and a condition determining when recovery has occurred.

The demand for executable realization therefore forces the semantic content of the theory into explicit mathematical structure.

In this sense, future regenerative compilers would extend the role of verified compilation beyond preservation of conventional program semantics.

Their correctness problem would be:

preserve the semantics of repair and regeneration through successive levels of executable representation.

Such architectures remain prospective. The present paper supplies the algebraic and runtime structures from which they could be developed.

The resulting research program connects repair algebras, semantic descent, executable encodings, runtime verification, and compiler correctness within a single computational architecture.

The next section identifies several mathematical extensions required to develop this program beyond the deterministic framework considered here.

9 Future Research

9.1 Type-Theoretic Biological Systems

The repair-algebra and regenerative-runtime frameworks developed in this paper are presently formulated primarily in terms of sets, transformations, relations, predicates, and semantic maps. A natural extension is to express these structures within dependent type theory [8], where biological states and the conditions governing their admissibility can be represented together with machine-checkable evidence.

The central advantage of a type-theoretic formulation is that properties currently expressed externally as predicates may instead become part of the types inhabited by runtime objects.

Let

$$S$$

be a state space and let

$$V : S \to \mathsf{Prop}$$

be a viability predicate.

In a set-theoretic presentation, one writes

$$s \in V$$

to assert that a state is viable.

In dependent type theory, a viable state may instead be represented by the dependent pair

$$\Sigma(s : S),\, V(s).$$

An inhabitant

$$\langle s, p\rangle : \Sigma(s : S),\, V(s)$$

contains both the runtime state s and evidence

$$p : V(s)$$

that the state satisfies the viability condition.

This changes the status of viability from a property checked after execution to a property that may be incorporated into the representation of admissible runtime states.

Definition 9.1 (Typed Viable State)**.** Given a state type S and viability predicate

$$V : S \to \mathsf{Prop},$$

define the type of viable states by

$$S_V := \Sigma(s : S),\, V(s).$$

A transformation that is required to preserve viability may then be assigned the type

$$r : S_V \to S_V.$$

Such a transformation cannot return an unverified state of S. Its output must include evidence that viability continues to hold.

This suggests a type-theoretic interpretation of invariant-preserving runtime transformations.

Let

$$I : S \to \Lambda$$

be an observable and let

$$\lambda \in \Lambda$$

be a designated invariant value.

Define

$$S_{I=\lambda} := \Sigma(s : S),\, I(s) = \lambda.$$

A transformation preserving this invariant may be represented as

$$\tau : S_{I=\lambda} \to S_{I=\lambda}.$$

The preservation obligation is therefore reflected directly in the transformation's type.

More generally, suppose

$$\eta : S \to \Omega$$

is the semantic observation map.

For a reference state

$$s : S,$$

define its semantic fiber by

$$F_\eta(s) := \Sigma(t : S),\, \eta(t) = \eta(s).$$

This type represents states semantically equivalent to s.

If viability is also required, define the homeostatic semantic fiber

$$H_\eta(s) := \Sigma(t : S),\, V(t) \times (\eta(t) = \eta(s)).$$

This is the type-theoretic analogue of

$$V \cap [s]_\eta.$$

A successful repair of damage d at state s may therefore be represented constructively as an inhabitant of

$$H_\eta(s).$$

Definition 9.2 (Typed Successful Repair)**.** Let

$$d : S \to S$$

be a damage transformation.

A *typed successful repair* of d at s consists of a repair transformation r together with evidence

$$p_V : V(r(d(s)))$$

and

$$p_\eta : \eta(r(d(s))) = \eta(s).$$

Equivalently, the repaired state together with its correctness evidence inhabits

$$H_\eta(s).$$

This formulation makes the relationship between repair and proof explicit.

A successful repair does not merely compute a state

$$r(d(s)).$$

It computes a state together with evidence that the defining recovery conditions hold.

Thus a verified repair may be represented schematically as

$$\boxed{\text{damage} \longrightarrow \text{repair computation} \longrightarrow \text{recovered state} + \text{proof of recovery.}}$$

Dependent types also provide a natural representation for repair preconditions.

A repair operation may not be applicable to every state.

Let

$$A_r : S \to \mathsf{Prop}$$

be the admissibility predicate for repair r.

Instead of representing r as an unrestricted function

$$r : S \to S,$$

one may assign it the dependent type

$$r : \Sigma(s : S), A_r(s) \to S.$$

The input therefore contains evidence that the repair is applicable to the current state.

If the repair must additionally preserve viability, its type may be strengthened to

$$r : \Sigma(s : S), A_r(s) \times V(s) \to S_V.$$

The type system then expresses both the precondition and the required postcondition.

Repair composition becomes correspondingly richer.

Suppose

$$r_1$$

produces states satisfying a predicate P, while

$$r_2$$

requires P as a precondition.

Their composability may then be witnessed at the type level.

Instead of merely asking whether

$$r_2 \circ r_1$$

is syntactically defined, the type checker may verify that the postcondition of r_1 establishes the precondition required by r_2.

This suggests a typed repair calculus in which admissible repair pathways are constructed through proof-carrying composition.

Definition 9.3 (Proof-Carrying Repair Path)**.** A *proof-carrying repair path* is a sequence

$$r_1, \ldots, r_n$$

together with evidence that:

1. each repair is admissible at the state on which it is applied;
2. the output conditions of each repair satisfy the input conditions of the next;
3. designated safety or viability predicates hold at required intermediate states;
4. the terminal state satisfies the specified recovery condition.

Such a path contains not only a repair program but a derivation of its admissibility.

The same idea extends to regenerative executions.

Let

$$\mathrm{Step} : S \to S \to \mathsf{Prop}$$

represent the runtime transition relation.

A finite execution may be represented inductively.

For example, one may define a family

$$\mathrm{Trace} : S \to S \to \mathsf{Type}$$

whose inhabitants witness finite admissible transition sequences between states.

Conceptually,

$$\mathrm{Trace}(s, t)$$

is inhabited exactly when there exists a verified runtime path from s to t.

Regenerative reachability can then be represented by the dependent type

$$\text{Recovery}(s) := \Sigma(t : S),\ \text{Trace}(s, t) \times (t \in H),$$

where H is the designated homeostatic region.

An inhabitant of

$$\text{Recovery}(s)$$

contains:

- a terminal state t;
- a verified execution from s to t;
- evidence that t lies in the recovery region.

Recoverability therefore becomes an inhabitation problem.

$$\boxed{s \text{ is recoverable} \quad \Longleftrightarrow \quad \text{Recovery}(s) \text{ is inhabited.}}$$

This gives a constructive interpretation of the existential recovery conditions developed earlier in the paper.

The distinction between existential and universal regeneration may also be expressed type-theoretically.

Existential recovery asks for an inhabitant of

$$\Sigma(\pi : \text{Exec}(s)),\ \text{ReachesH}(\pi).$$

Universal recovery requires a function assigning recovery evidence to every admissible execution:

$$\Pi(\pi : \text{Exec}(s)),\ \text{ReachesH}(\pi).$$

The logical distinction between

$$\exists$$

and

$$\forall$$

therefore becomes a distinction between dependent sum and dependent product types.

This formulation connects regenerative runtime semantics with the propositions-as-types interpretation.

Statements about biological runtime behavior may become types, while proofs of those statements become computational objects inhabiting those types.

The resulting architecture can be summarized as

$$\boxed{\text{biological property} \longrightarrow \text{logical proposition} \longrightarrow \text{dependent type} \longrightarrow \text{proof object} \longrightarrow \text{verified runtime evidence.}}$$

A type-theoretic regenerative runtime could consequently make invalid states or invalid transitions unrepresentable at selected levels of abstraction.

For example, a function of type

$$S_V \to S_V$$

cannot, within the formal system, produce an output without also supplying evidence of viability.

Likewise, a function returning

$$H_\eta(s)$$

cannot claim successful semantic repair without producing evidence of both viability and semantic restoration.

This does not imply that the underlying biological assumptions are correct. As with the verification framework developed earlier, type checking proves properties relative to the formal model.

The empirical interpretation of the predicates

$$V, \eta, A_r,$$

and the transformations themselves remains an independent scientific question.

Nevertheless, dependent type theory provides a powerful formal environment for expressing the internal correctness obligations of regenerative systems.

It also creates a direct route toward machine-checked implementations.

Repair algebras could be represented as typed structures; repair morphisms as structure-preserving functions; repair pathways as inductive objects; regenerative executions as indexed traces; and correctness theorems as machine-checkable proof terms.

A future implementation in a theorem prover such as Lean [9] could therefore seek a correspondence of the form

$$\boxed{\text{repair algebra} \longrightarrow \text{dependent-type specification} \longrightarrow \text{verified regenerative runtime.}}$$

This would strengthen the executable architecture developed in the present paper by integrating state representation, runtime behavior, and correctness evidence within a single formal system.

The next subsection considers a different extension of the current theory: the replacement of discrete deterministic state transitions by continuous runtime dynamics.

9.2 Continuous Runtime Models

The regenerative runtimes developed in this paper are formulated primarily as discrete transition systems. Biological systems, however, frequently evolve through continuous processes in which state variables change over time rather than through isolated transitions.

A natural extension is therefore to replace the discrete state trajectory

$$s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_n$$

with a continuous trajectory

$$x : [0, T] \to S.$$

When S is equipped with an appropriate geometric or topological structure, the runtime state becomes a time-dependent point

$$x(t) \in S.$$

The evolution of the system may then be governed by a dynamical law

$$\dot{x}(t) = F(x(t)),$$

where

$$F : S \to TS$$

is a vector field or, more generally, an evolution operator defining the continuous runtime dynamics.

This changes the mathematical representation of execution but not the central semantic problem.

The principal questions remain:

- which regions of state space correspond to viable operation;
- which observables characterize the relevant organization;
- which perturbations displace the system from those regions;
- which dynamics permit recovery;
- which invariants survive or are restored along the trajectory.

Let

$$V \subseteq S$$

be the viable region.

Definition 9.4 (Continuous Viable Execution)**.** A trajectory

$$x : [0, T] \to S$$

is *continuously viable* when

$$x(t) \in V$$

for every

$$t \in [0, T].$$

This is the continuous analogue of trace viability in a discrete runtime.
If

$$H \subseteq V$$

is a homeostatic region, maintenance corresponds to

$$x(t) \in H$$

for all t.

Regeneration instead permits displacement from H followed by return.

Definition 9.5 (Continuous Regenerative Trajectory)**.** Let

$$H \subseteq V.$$

A continuous trajectory

$$x : [0, T] \to S$$

is *regenerative relative to* H when there exists

$$t_d \in [0, T)$$

such that

$$x(t_d) \notin H$$

and there exists

$$t_r \in (t_d, T]$$

such that

$$x(t_r) \in H.$$

If the trajectory remains viable throughout the interval,

$$x(t) \in V \qquad \text{for every } t \in [0, t_r],$$

then regeneration occurs without loss of viability.

The distinction between maintenance and regeneration therefore survives the transition from discrete to continuous dynamics.

Maintenance requires

$$x(t) \in H \qquad \forall t,$$

whereas regeneration permits

$$x(t) \notin H$$

for some interval but requires subsequent return.

Semantic observables extend naturally to continuous trajectories.

Let

$$\eta : S \to \Omega.$$

The semantic trajectory is

$$t \mapsto \eta(x(t)).$$

A semantic invariant is continuously preserved when

$$\eta(x(t)) = \eta(x(0))$$

for every

$$t \in [0, T].$$

Semantic regeneration requires only eventual restoration:

$$\eta(x(t_r)) = \eta(x(0)).$$

Thus the distinction developed earlier remains:

$$\boxed{\text{continuous semantic preservation} \neq \text{continuous semantic recovery.}}$$

A system may leave its initial semantic class and subsequently return to it.

Continuous dynamics also permit recovery to be studied through attractor structure.

Suppose

$$H \subseteq S$$

is an invariant attracting region for the flow generated by F.

A state x_0 belongs to the basin of attraction of H when its trajectory approaches H under the runtime dynamics.

Writing the flow as

$$\varphi_t(x_0),$$

one may define the basin

$$\mathcal{B}(H) = \{x \in S : \operatorname{dist}(\varphi_t(x), H) \to 0 \text{ as } t \to \infty\}.$$

This provides a dynamical interpretation of regenerative capacity.

States in

$$\mathcal{B}(H)$$

are displaced states from which the autonomous dynamics can return toward the homeostatic region.

The size and geometry of

$$\mathcal{B}(H)$$

therefore characterize one aspect of robustness.

A larger basin indicates that a wider class of perturbations can be absorbed without irreversible loss of the designated organization.

However, attraction alone does not guarantee safe regeneration.

A trajectory may converge toward H while passing through an unsafe or nonviable region.

Let

$$K \subseteq S$$

be a safe region.

Define the safe regenerative basin

$$\mathcal{B}_K(H) = \{x \in \mathcal{B}(H) : \varphi_t(x) \in K \text{ for all } t \geq 0\}.$$

Then

$$\mathcal{B}_K(H) \subseteq \mathcal{B}(H).$$

The safe regenerative basin identifies states from which recovery occurs without violating the designated trajectory constraints.

Repair operations may also be incorporated into continuous models.

Instead of a single autonomous vector field, consider a family

$$\{F_r : r \in \mathcal{R}\}$$

of repair-dependent dynamics.

Selecting a repair operation changes the evolution law:

$$\dot{x}(t) = F_r(x(t)).$$

A time-dependent repair policy may therefore produce

$$\dot{x}(t) = F_{\pi(t,x(t))}(x(t)),$$

where

$$\pi$$

selects the active repair regime.

This converts regenerative execution into a control problem.

More generally, one may write

$$\dot{x}(t) = F(x(t), u(t)),$$

where

$$u(t) \in U$$

is a control input representing admissible intervention or repair activity.

The regenerative problem then becomes:

find an admissible control u that drives a displaced state back to the required region.

If

$$x(0) = x_0 \notin H,$$

one seeks a control u and time T satisfying

$$x(T) \in H,$$

possibly subject to

$$x(t) \in V$$

or

$$x(t) \in K$$

throughout the trajectory.

This is the continuous analogue of searching for a repair word.

In the discrete framework, one seeks

$$w = (r_1, \ldots, r_n)$$

such that

$$[\![w]\!](d(s)) \in H.$$

In the continuous framework, one seeks a control trajectory

$$u : [0, T] \to U$$

such that

$$x(T) \in H.$$

The two problems share the same structural form:

displacement + admissible transformations + target invariant region = recovery problem.

Continuous models also permit repair cost to be represented through functionals.

For example,

$$J[u] = \int_0^T L(x(t), u(t))\, dt$$

may measure the cost of a recovery trajectory.

An optimal regenerative control problem then takes the form

$$\min_u J[u]$$

subject to

$$\dot{x} = F(x, u),$$

$$x(0) = x_0,$$

$$x(T) \in H,$$

and any required viability or safety constraints.

This extends repair-depth optimization into continuous state and control spaces.

The resulting framework connects regenerative runtimes with dynamical systems, stability theory, viability theory, and control.

It also sharpens the interpretation of regeneration.

Regeneration is not fundamentally tied to discrete repair events. It is a property of trajectories through state space relative to designated invariants and recovery regions.

The discrete and continuous theories can therefore be viewed as two realizations of a common structure:

$$\boxed{\text{state space} + \text{dynamics} + \text{viability} + \text{semantic invariants} + \text{recovery}.}$$

A complete theory may ultimately require hybrid runtimes combining both forms.

Continuous dynamics could represent ongoing biological evolution, while discrete transitions represent events such as damage, intervention, regulatory switching, or activation of repair programs.

Such a hybrid execution might satisfy

$$\dot{x} = F_q(x)$$

within runtime mode q, together with discrete transitions

$$(x, q) \longrightarrow (x', q')$$

when specified conditions are met.

Repair algebras would then govern discrete transformations and intervention structure, while continuous dynamics would govern evolution between those events.

This suggests a broader future model:

$$\boxed{\text{repair algebra} + \text{continuous dynamics} + \text{runtime policy} = \text{hybrid regenerative system}.}$$

Developing the mathematical conditions under which invariants, recoverability, repair correctness, and semantic equivalence survive this extension is a natural direction for future work.

9.3 Biological Policy Compilers

A regenerative runtime does not merely require a collection of possible repair transformations. It must determine which transformation should be applied, under which conditions, and in which order.

This decision structure is represented in the present framework by the runtime policy

$$\Pi.$$

The existence of a successful repair pathway establishes structural recoverability, but it does not guarantee that the runtime will discover or execute that pathway.

This distinction motivates a further computational problem: the synthesis of runtime policies from high-level specifications of viability, safety, and recovery.

We call a system performing this translation a *biological policy compiler*.

Definition 9.6 (Biological Policy Specification)**.** A *biological policy specification* is a tuple

$$\mathcal{P} = (S, V, H, K, \mathcal{T}, \mathcal{G}, \mathcal{O}),$$

where:

- S is the runtime state space;
- $V \subseteq S$ is the viable region;

- $H \subseteq V$ is a designated homeostatic or recovery region;
- $K \subseteq S$ is a designated safe region;
- $\mathcal{T}$ is the collection of admissible runtime transformations;
- $\mathcal{G}$ is a collection of required semantic or regenerative goals;
- $\mathcal{O}$ specifies optimization criteria used to distinguish among admissible policies.

A policy compiler takes such a specification and attempts to construct a context-sensitive deterministic policy

$$\Pi : S \times \mathcal{C} \rightharpoonup \mathcal{T},$$

such that executions generated by the policy relative to admissible context trajectories satisfy the required conditions.

The synthesis problem can therefore be expressed as

$$\boxed{\text{specification} \longrightarrow \text{policy synthesis} \longrightarrow \text{runtime controller} \longrightarrow \text{verified regenerative behavior.}}$$

The simplest policy objective is viability preservation.

A policy Π is viability-preserving when every transformation selected from a viable state in an admissible context remains viable.

Formally, if

$$s \in V, \qquad c \in \mathcal{C},$$

and

$$\tau = \Pi(s, c)$$

is defined, then

$$\tau(s) \in V.$$

A stronger policy may require safety:

$$s \in K \quad \Longrightarrow \quad \tau(s) \in K.$$

Regenerative policy synthesis introduces an eventual recovery requirement.

Suppose

$$x \notin H$$

is a displaced state.

A regenerative policy should generate an execution

$$x = s_0 \longrightarrow s_1 \longrightarrow \cdots$$

that eventually reaches

$$s_n \in H.$$

If every policy-admissible execution from x reaches H, then the policy provides a universal recovery guarantee on x.

This leads naturally to a synthesis problem.

Definition 9.7 (Regenerative Policy Synthesis)**.** Given a runtime state space S, recovery region H, admissible transition structure $\mathcal{T}$, and a domain

$$B \subseteq S,$$

the *regenerative policy synthesis problem* asks for a policy Π such that every policy-admissible execution beginning in

$$x \in B$$

eventually reaches H.

When safety is required, the synthesis condition becomes

$$s_i \in K$$

for every intermediate state, together with eventual recovery

$$s_n \in H.$$

Thus the compiler must satisfy both a safety property and a liveness property.

The synthesis objective can be expressed schematically as

always remain safe	+	eventually recover.

These conditions may conflict.

A short recovery path may cross an unsafe region, while a safe path may require additional repair operations or temporary compensation.

Policy synthesis must therefore reason over complete trajectories rather than selecting transformations solely according to immediate local improvement.

This provides a formal role for compensatory states.

Suppose

$$C \subseteq V \setminus H$$

is a compensatory region.

A valid policy may deliberately move or maintain the runtime within C when immediate return to H is unavailable.

The trajectory may therefore have the form

$$x \longrightarrow C \longrightarrow \cdots \longrightarrow H.$$

The intermediate compensatory regime is not itself regeneration, but it may preserve viability while the runtime progresses toward a regenerative trajectory.

Policy compilation may also incorporate repair cost.

Let

$$c : S \times \mathcal{T} \to \mathbb{R}_{\geq 0}$$

assign a local cost to each selected transformation.

For a finite execution

$$\pi = s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} s_n,$$

define

$$C(\pi) = \sum_{i=1}^{n} c(s_{i-1}, \tau_i).$$

The compiler may seek a policy minimizing expected or worst-case recovery cost subject to safety and regeneration constraints.

In a deterministic finite runtime, a minimum-cost policy problem may take the form

$$\min_{\Pi} C(\pi_{\Pi}(x))$$

subject to

$$\pi_{\Pi}(x) \subseteq K$$

and

$$\pi_{\Pi}(x) \cap H \neq \varnothing.$$

Different objective functions yield different policy architectures.

One policy may minimize repair depth.

Another may minimize resource use.

Another may prioritize safety margins.

Another may prefer repair pathways that preserve a larger collection of semantic invariants.

The compiler therefore separates the definition of successful regeneration from the strategy used to obtain it.

This separation is important because the repair algebra describes what can be done, whereas the policy describes what the runtime chooses to do.

$$\boxed{\text{repair algebra} = \text{space of available transformations,}}$$

while

$$\boxed{\text{policy} = \text{strategy for selecting transformations during execution.}}$$

A policy compiler connects these levels.

The repair algebra provides the compiler's instruction set.

The regenerative specification provides its correctness conditions.

The compiler synthesizes a strategy over those instructions.

The resulting runtime executes that strategy.

This gives the architecture

$$\boxed{\text{repair algebra} \longrightarrow \text{policy synthesis} \longrightarrow \text{regenerative runtime} \longrightarrow \text{verified recovery.}}$$

Policy compilation also introduces an important distinction between offline and online synthesis.

An *offline* compiler may construct a policy before runtime execution. For finite state spaces, it may analyze the transition graph and determine recovery strategies for an entire region of states.

An *online* compiler may instead synthesize or revise a policy as new runtime states are encountered.

The latter is particularly relevant when the complete state space or damage structure is not known in advance.

A future adaptive architecture might therefore alternate between execution and policy synthesis:

$$\text{observe} \longrightarrow \text{classify} \longrightarrow \text{synthesize} \longrightarrow \text{execute} \longrightarrow \text{verify} \longrightarrow \text{observe}.$$

Such a system would not merely execute a fixed repair program. It would construct repair strategies relative to the current runtime state and the invariants that must be preserved or restored.

A stronger architecture could require the policy compiler to emit a certificate together with the synthesized policy.

Definition 9.8 (Certified Regenerative Policy)**.** A *certified regenerative policy* on a region $B \subseteq S$ is a policy Π together with machine-checkable evidence that every execution generated from B satisfies the designated safety and recovery conditions.

The compiler output would then have the form

$$(\Pi, \Gamma_\Pi),$$

where

$$\Gamma_\Pi$$

is a correctness certificate.

Verification of the certificate would establish that the synthesized policy satisfies the formal specification.

This suggests a future compiler architecture in which regenerative behavior is not manually encoded as a fixed sequence of repair operations.

Instead, the programmer or modeler specifies the required invariants, viability conditions, safety constraints, and recovery objectives.

The compiler determines how those requirements can be realized.

The conceptual shift is therefore

program the repair sequence $\longrightarrow$ specify the regenerative invariants and synthesize the policy.

Such a compiler would represent a direct computational application of the theory developed in this paper.

Repair algebras would define the available transformation structure. Regenerative runtimes would provide the execution semantics. Verification conditions would define correctness. Policy compilation would synthesize the operational strategy connecting them.

Together, these components suggest a broader architecture for systems whose behavior is organized around preservation and recovery rather than fixed execution alone:

specification + repair algebra + policy synthesis + runtime verification = regenerative computation.

Developing such compilers would require substantial additional work, including algorithms for policy synthesis, representations of uncertainty, continuous and hybrid runtime models, scalable verification procedures, and empirically grounded biological state spaces.

Nevertheless, the present framework identifies the mathematical interfaces such systems would require and provides a foundation for their future formalization.

10 Conclusion

This paper has introduced repair algebras and regenerative runtimes as a formal framework for reasoning about systems capable of maintaining, repairing, and restoring their organization under perturbation.

The central premise is that regeneration should not be identified with literal return to an identical physical state.

A system may undergo substantial transformation while preserving or restoring the properties that define its relevant organization.

Accordingly, the fundamental object of the theory is not state identity but invariance through transformation.

Given a state space

$$S,$$

a viability region

$$V \subseteq S,$$

and a semantic observation map

$$\eta : S \to \Omega,$$

the framework distinguishes physical state from the semantic properties selected as relevant to system identity.

Two states may therefore differ while remaining semantically equivalent:

$$s \sim_\eta t \quad \Longleftrightarrow \quad \eta(s) = \eta(t).$$

This distinction makes it possible to define successful repair without requiring exact restoration.

For a damage transformation

$$d \in \mathcal{D}$$

and repair transformation

$$r \in \mathcal{R},$$

successful semantic repair requires

$$r(d(s)) \in V$$

and

$$\eta(r(d(s))) = \eta(s).$$

Equivalently,

$$r(d(s)) \in V \cap [s]_\eta.$$

Thus the repaired system may satisfy

$$r(d(s)) \neq s$$

while nevertheless restoring the designated invariant structure.

This leads to the central interpretation of the framework:

Regeneration is invariance through transformation.

Repair algebras formalize the transformation structure underlying this process.

Damage transformations describe admissible forms of disruption.

Repair transformations describe possible restorative operations.

Composition describes multi-stage repair.

Repair words and repair pathways expose the internal structure of recovery.

Semantic equivalence determines which distinctions between states matter for the modeled notion of restoration.

Repair depth and related cost measures quantify the complexity of recovery.

Commutativity and idempotence identify algebraic conditions under which apparently different repair sequences collapse to equivalent effective operations.

Repair-algebra morphisms extend the theory across representations.

They make it possible to ask whether semantic equivalence, successful repair, exact repair, recoverability, and repair composition survive translation from one representation to another.

The resulting perspective separates properties intrinsic to a particular state representation from properties preserved by the structure of the repair system itself.

The computational-complexity results further show that even highly restricted repair systems can contain nontrivial computational structure.

In particular, the bounded repair-depth problem remains computationally hard even when primitive repairs are deterministic, monotone, idempotent, and pairwise commuting.

The difficulty can therefore arise from selecting the appropriate repair effects rather than from complicated interaction among the repair operations themselves.

Repair algebra alone, however, describes structural possibility rather than runtime behavior.

For this reason, the paper introduced regenerative runtimes.

A regenerative runtime

$$\mathfrak{R} = (S, V, \Omega, \eta, \mathcal{D}, \mathcal{R}, \mathcal{T}, \mathcal{C}, \Pi)$$

adds transition dynamics and runtime policy to the algebraic structure.

This permits the theory to distinguish:

what transformations exist

from

what transformations are actually executed.

The distinction is essential.

A system may be structurally recoverable while its runtime policy fails to realize any successful recovery pathway.

Conversely, a regenerative policy may select transformations that preserve viability, navigate compensatory states, and eventually return the system to a designated homeostatic region.

The runtime formulation therefore turns repair from a static relation into an executable process.

Homeostasis becomes persistence within a designated region.

Damage becomes displacement.

Compensation becomes viable operation outside the designated homeostatic regime.

Regeneration becomes return to the required invariant region through an admissible trajectory.

Failure becomes the loss of required viability or recovery conditions.

These distinctions make it possible to characterize regenerative behavior at the level of complete trajectories rather than isolated endpoint states.

The framework also separates maintenance from recovery.

An invariant may be preserved continuously:

$$I(s_i) = I(s_0) \qquad \text{for every } i,$$

or it may be temporarily violated and later restored:

$$I(s_n) = I(s_0).$$

The first describes preservation.

The second describes recovery.

This distinction is fundamental to regenerative systems because successful regeneration need not imply uninterrupted invariance.

A damaged system may leave its homeostatic or semantic class before later returning to it.

Regeneration is therefore a temporal property of transformation and restoration.

The verification framework developed in this paper makes these distinctions machine-addressable.

Viability, safety, semantic preservation, semantic recovery, repair correctness, and regenerative reachability can all be formulated as explicit predicates over states and executions.

A successful runtime may consequently produce not merely a recovered state but a certificate showing that the recovery trajectory satisfies its formal requirements.

This yields the architecture

$$\boxed{\text{state} \longrightarrow \text{perturbation} \longrightarrow \text{repair trajectory} \longrightarrow \text{recovered invariants} \longrightarrow \text{verification.}}$$

The framework thereby moves regeneration from qualitative description toward formal executable semantics.

This movement also clarifies the role of semantic descent.

A high-level notion such as regeneration cannot be executed directly by a machine.

It must be decomposed into mathematically explicit components:

states, predicates, transformations, composition laws, invariants, transition rules, recove

Each level of implementation introduces additional representational detail.

The central correctness question is whether the semantics required by the higher level survive that descent.

Thus executable encodings, regenerative runtimes, and compiler correctness are instances of the same general problem:

$$\boxed{\text{which properties survive successive transformations of representation and execution?}}$$

This observation motivates the connection between the present theory and verified executable encodings.

A regenerative runtime may ultimately be lowered into a machine representation in which states, repair transformations, traces, and verification conditions are encoded as executable objects.

Correctness then requires the encoded execution to realize the same abstract transformation structure and preserve the same designated semantics.

The complete architecture is therefore

$$\boxed{\text{repair algebra} \longrightarrow \text{regenerative runtime} \longrightarrow \text{executable encoding} \longrightarrow \text{verified machine execution.}}$$

This also gives operational meaning to the demand for mathematical precision.

Execution forces every distinction relevant to computation to become explicit.

A runtime cannot operate on an undefined notion of repair.

A verifier cannot prove recovery without a recovery predicate.

A compiler cannot preserve semantics unless the relevant semantics have been specified.

A policy cannot guarantee regeneration unless the target invariant region and admissible transition structure are defined.

In this restricted operational sense,

$$\boxed{\textbf{execution is a test of operational semantic completeness.}}$$

This does not refer to logical completeness in the classical sense.

Rather, it expresses the requirement that a theory intended for execution must resolve the ambiguities that matter to its operational behavior.

The biological interpretation of the framework remains deliberately abstract.

Nothing in the algebra alone determines which state variables, viability conditions, semantic observables, damage transformations, or repair operations correctly describe a particular biological system.

Those structures must be supplied and validated by the relevant empirical science.

Formal verification establishes consequences of the model:

$$M \models P.$$

It does not establish, by itself, that

$$M$$

is an empirically adequate representation of the biological system.

This boundary is essential for any computational application of the theory.

Within that boundary, however, the framework provides a common mathematical language for several questions that are often treated separately:

- What constitutes damage?
- Which properties define continued viability?
- Which transformations count as repair?
- When does compensation differ from regeneration?
- Which damaged states remain recoverable?
- How difficult is recovery?
- Which invariants must survive or be restored?
- Which recovery pathways are safe?
- Which properties survive changes of representation?
- Can a runtime guarantee recovery rather than merely permit it?
- Can recovery itself be represented and verified computationally?

The framework suggests that these are not independent questions.

They are different aspects of a common mathematical structure organized around transformation and invariance.

This perspective also provides a criterion for identifying regenerative behavior more generally.

A system exhibits regeneration relative to a specified model when a perturbation displaces it from a designated invariant regime and its subsequent dynamics restore that regime through admissible transformation.

The final state need not be identical to the initial state.

What must return is the structure designated as relevant.

Schematically,

$$\boxed{\text{perturbation} + \text{transformation} + \text{restoration of designated invariants} = \text{regeneration.}}$$

This formulation makes regeneration a property that can, in principle, be compared across different physical realizations.

Different systems may possess different state spaces, mechanisms, and repair pathways while exhibiting the same abstract recovery structure.

Future work may therefore investigate equivalence classes of regenerative systems, invariant-preserving translations between biological models, continuous and hybrid regenerative dynamics, dependent-type formulations, certified policy synthesis, and compilers that translate high-level recovery specifications into verified runtime machinery.

The broader objective is not merely to describe systems that recover.

It is to determine the mathematical structure that makes recovery possible, the invariants that define what must survive, the transformations that can restore those invariants, and the computational machinery required to execute and verify the process.

The resulting theory may be summarized by the sequence

$$\boxed{\text{invariants} \longrightarrow \text{repair structure} \longrightarrow \text{recovery dynamics} \longrightarrow \text{executable semantics} \longrightarrow \text{verification.}}$$

Under this view, a regenerative system is characterized not by remaining unchanged, but by its capacity to preserve or recover what matters through change.

That principle provides the unifying interpretation of repair algebras and regenerative runtimes:

$$\boxed{\textbf{Regeneration is not sameness of state. It is invariance through change.}}$$

References

[1] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I." *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[2] Adrian Diamond. *Executable Gödel Encodings: A Verified Runtime for Logical Syntax*. AAD Systems, March 2026.

[3] Adrian Diamond. *Executable Σ-Types: A Runtime for Proof-Carrying State Transitions via Gödel Encoding*. AAD Systems, April 26, 2026.

[4] Gordon D. Plotkin. "A Structural Approach to Operational Semantics." *The Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[5] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." *Communications of the ACM*, 12(10):576–580, 1969.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[7] Richard M. Karp. "Reducibility Among Combinatorial Problems." In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pp. 85–103. Plenum Press, New York, 1972.

[8] Per Martin-Löf. *Intuitionistic Type Theory*. Notes by Giovanni Sambin. Studies in Proof Theory, Vol. 1. Bibliopolis, Napoli, 1984.

[9] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)." In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction – CADE-25*, Lecture Notes in Computer Science, Vol. 9195, pp. 378–388. Springer, 2015.

[10] Walter B. Cannon. *The Wisdom of the Body*. W. W. Norton & Company, New York, 1932.

[11] Hiroaki Kitano. "Biological Robustness." *Nature Reviews Genetics*, 5:826–837, 2004.

[12] Uri Alon. *An Introduction to Systems Biology: Design Principles of Biological Circuits*. CRC Press, 2006.

[13] Jason Pellettieri and Alejandro Sánchez Alvarado. "Cell Turnover and Adult Tissue Homeostasis: From Humans to Planarians." *Annual Review of Genetics*, 41:83–105, 2007.

[14] Geoffrey C. Gurtner, Sabine Werner, Yann Barrandon, and Michael T. Longaker. "Wound Repair and Regeneration." *Nature*, 453:314–321, 2008.

[15] Kenneth D. Poss. "Advances in Understanding Tissue Regenerative Capacity and Mechanisms in Animals." *Nature Reviews Genetics*, 11:710–722, 2010.