P-Programming:
Scope, Definitions, Realizations, and Settling

MJC and Graf

March 17, 2026

Abstract

We introduce the `#p_programming_*` namespace as the layer above any particular implementation of pattern-based programming. The UM runner specification is one realization of this layer; the `cmpr` system is another. The present note records a first stable set of definitions that can sit above those implementations without collapsing into either one, and sketches the stronger thesis that settling belongs to intrinsic P-programming semantics rather than to an external repair phase.

1 Position in the project

The top-level exploratory block is `#p_programming_workpad`. That workpad is not itself the implementation specification. It is the frontier block where candidate laws, constructions, and objections are worked out. As pieces stabilize, they should be moved into downstream `#p_programming_*` blocks and the workpad should be compressed by replacing settled draft material with references.

This yields the intended hierarchy:

Layer	Role
`#p_programming_workpad`	Frontier / expanding envelope
`#p_programming_definitions`	Stable core definitions
`#p_programming_operations`	Stable basic operators
`#p_programming_realizations`	Mapping to implementations
`#umr_spec`, cmpr blocks	Concrete realizations

2 Core state

We take a P-program state to be

$$u = (E, T, P).$$

Here:

- E is the available event structure, including events and their organization into event spaces.
- T is the current support state over those events.
- P is the available pattern structure: transfer rules, persistence mechanisms, conjunctions, gates, and learned patterns.

The broader UM also includes f and ω, but the claim here is that canonical P-programming is described primarily at the level of (E, T, P): what exists, what is active, and what transformations are available.

3 Why the name still starts with P

The name *P-programming* rather than *U-programming* is retained because the constructive burden lies mainly in P. Persistence, routing, transfer, conjunction, and learned structure are all pattern questions. Nevertheless E and T remain essential:

- P has no meaning without a domain of events in E,
- and no operational content without a current support state T.

4 Default persistence convention

A naturalistic default is *reset-by-default*: after each time step, support returns to zero unless persistence is explicitly programmed. Under this convention, T is sparse most of the time and memory is not assumed. Persistence must itself be built in P.

This is important because it shifts the interpretation of program state. Instead of taking variable persistence as primitive, we treat it as a construction. Small loops in P become candidate persistence devices, and the question of control can potentially be handled by sparsity and timing rather than by an external program counter.

5 Basic operators

We record a first family of namespace-level introduction operators:

$$u_0 = (\emptyset, \emptyset, \emptyset),$$

$$u + e, \qquad u + t, \qquad u + p_i.$$

Their intended readings are:

- $u + e$: architecture growth, adding event or event-space structure,
- $u + t$: assignment or activation, asserting support,
- $u + p_i$: rule introduction, adding an atomic pattern.

These are not yet claimed to be the final algebra of P-programming. They are stable enough, however, to serve as the first shared language for discussing constructions across implementations.

6 Imperative programs as pattern systems

An imperative step such as

$$a = b * c$$

can be read as a pattern-level relation from

$$ES_b \times ES_c \to ES_a.$$

On that reading, a variable naturally determines an event space whose events are the values that variable may take, and operational semantics becomes a question of which patterns are active at which times.

This leaves an open control question: whether control is best expressed via a dedicated event space, via context events, via timing structure, or through sparsity in T is not yet settled.

7 Realizations

The namespace sits above particular implementations.

UMR realization. `#umr_spec` is one implementation-specific realization of P-programming. It fixes a particular runner, threshold-creation mechanism, SN interchange format, and set of operational conventions.

cmpr realization. `cmpr` is another realization. Blocks, events, wants, working memory, and code-generation workflows form a different concrete system whose programming practice still sits naturally under the distinctions among E, T, and P.

8 Settling

The next semantic step is to distinguish raw propagation from semantic resolution. Starting from a current state

$$u = (E, T, P),$$

application of the active patterns in P to the current support state T yields a provisional next state, say T'. The strong claim is that T' is not yet the semantic output of the step. It is only a provisional state.

The semantic output is a *settled* state T^*. Settling is the further process by which the event spaces in E become semantically admissible. If an event space is oversupported or undersupported, the program has not yet reached its meaning.

This reframes surprise. Surprise is not merely a diagnostic emitted by an external observer. Oversupport and undersupport are semantic signs that the program state has not yet resolved. In the ambitious direction, response-to-surprise and settling belong to the program itself:

- raw propagation is internal,
- surprise is internal,
- settling is internal,
- and external interpreters are approximations to this stronger semantic picture.

Current implementations may externalize parts of settling, but the namespace above them should not. The more ambitious view is that a P-program does not merely propagate support; it settles.

9 Conclusion

The immediate goal is not to freeze the entire algebra of P-programming. It is to establish a namespace above the current implementations, so that stable definitions can accumulate without forcing the workpad to become either an archive or an implementation specification. The workpad expands the envelope; downstream `#p_programming_*` blocks stabilize it. The settling thesis marks the next major semantic escalation: from describing program state to describing how program meaning is internally resolved.