AST Convergence: A Novel Pattern for Multi-Modal DSL Implementation

Daniel Szmulewicz
Independent Researcher
daniel.szmulewicz@gmail.com

***Abstract*—This paper introduces *AST Convergence*, a novel architectural pattern for implementing multi-modal Domain Specific Languages (DSLs) that elegantly solves the classic code duplication problem in language implementation. AST Convergence enables both external (text-based) and internal (code-based) DSL modalities to converge on identical Abstract Syntax Tree (AST) structures before semantic transformation, allowing a single transformation pipeline to handle both input sources. We present a formal definition of the pattern, demonstrate its implementation in the Bioscoop FFmpeg DSL, and provide quantitative evaluation showing significant reductions in code duplication while guaranteeing behavioral consistency between modalities. The pattern leverages Clojure's homoiconicity and metaprogramming capabilities while providing a generalizable approach applicable to other language ecosystems.**

***Index Terms*—Domain Specific Languages, AST, Multi-modal Processing, Code Generation, Clojure, Language Implementation**

I. Introduction

Domain Specific Languages (DSLs) have become essential tools in modern software development, providing expressive power for specialized domains while abstracting implementation complexity. A common challenge in DSL implementation is supporting multiple input modalities—typically external (text-based) and internal (code-based) forms—without duplicating transformation logic. Traditional approaches maintain separate processing pipelines for each modality, leading to code duplication, maintenance overhead, and potential behavioral inconsistencies.

This paper introduces *AST Convergence*, an architectural pattern that addresses these challenges by designing both external parsing and internal macro expansion to produce identical AST structures. This convergence enables a single transformation pipeline to handle both input sources, eliminating code duplication while ensuring consistent semantics.

Our contributions are:

- Formal definition of the AST Convergence pattern
- Implementation case study using the Bioscoop FFmpeg DSL
- Quantitative evaluation demonstrating code reduction and consistency guarantees
- Theoretical foundations connecting to programming language theory concepts
- Discussion of applications beyond the case study domain

II. Background and Related Work

A. Traditional Multi-Modal DSL Implementation

Traditional approaches to multi-modal DSL implementation follow separate processing paths (Figure 1):

Traditional Approach	
External DSL:	Text $\rightarrow$ Parser $\rightarrow$ AST_e $\rightarrow$ Transformer_e $\rightarrow$ Output
Internal DSL:	Code $\rightarrow$ Macro $\rightarrow$ AST_i $\rightarrow$ Transformer_i $\rightarrow$ Output

Fig. 1. Traditional multi-modal DSL implementation with separate transformation pipelines

This separation leads to several problems:

- *Code Duplication*: Two transformation logics must be maintained
- *Behavioral Inconsistency*: Different AST structures may lead to semantic differences
- *Maintenance Overhead*: Changes must be applied to both transformation systems

B. Related Work

Language-oriented programming in Racket [1] provides sophisticated tools for language creation, but typically maintains separate expander pipelines for different language modalities. Template engines and configuration systems often support multiple input formats, but rarely converge on identical intermediate representations before transformation.

Clojure's metaprogramming capabilities [2] and libraries like Instaparse provide the building blocks for AST Convergence, but the specific pattern of deliberate AST convergence appears to be a novel synthesis.

III. AST Convergence Pattern

A. Formal Definition

Let L be a DSL with external modality L_e (text-based) and internal modality L_i (code-based). Traditional implementation defines:

$$P_e : L_e \to AST_e \quad \text{(External parser)}$$
$$T_e : AST_e \to O \quad \text{(External transformer)}$$
$$M_i : L_i \to AST_i \quad \text{(Internal macro)}$$
$$T_i : AST_i \to O \quad \text{(Internal transformer)}$$

where $AST_e \neq AST_i$ and $T_e \neq T_i$.
AST Convergence redefines this as:

$$P_e : L_e \to AST \quad \text{(External parser)}$$
$$M_i : L_i \to AST \quad \text{(Internal macro)}$$
$$T : AST \to O \quad \text{(Single transformer)}$$

where both modalities produce identical AST structures, enabling a single transformation function T.

B. Architectural Overview

The AST Convergence pattern (Figure 2) consists of:

AST Convergence Pattern	
External Path:	Text → Parser → AST → **Single Transformer** → Output
Internal Path:	Code → Macro → AST → **Single Transformer** → Output

Fig. 2. AST Convergence pattern with unified transformation pipeline

1) *Unified AST Design*: The AST structure is explicitly designed to be producible by both parsing and macro expansion
2) *Convergence Point*: Both input paths converge on identical AST structures
3) *Single Transformation*: One transformation logic handles all ASTs regardless of source

IV. Case Study: Bioscoop FFmpeg DSL

We implemented AST Convergence in Bioscoop, a DSL for FFmpeg filtergraph generation. The system supports both text-based external DSL and Clojure-based internal DSL.

A. External DSL Implementation

The external DSL uses Instaparse for text processing:

```
(def dsl-parser
  (instaparse/parser
    (io/resource "lisp-grammar.bnf")
    :auto-whitespace :standard))

(defn compile-dsl [dsl-code]
  (let [ast (dsl-parser dsl-code)]
    (transform-ast ast (make-env))))
```

Listing 1. External DSL Parser

B. Internal DSL Implementation

The internal DSL uses macros that produce identical AST structures:

```
(defn form->ast [form]
  (cond
    (seq? form) [:list (map form->ast form)]
    (symbol? form) [:symbol (str form)]
    (number? form) [:number (str form)]
    :else [:literal form]))

(defmacro bioscoop [& forms]
  (let [ast-nodes (mapv form->ast forms)
        program-ast (vec (concat [:program] ast-
    nodes))]
    `(transform-ast ~program-ast (make-env))))
```

Listing 2. Internal DSL Macro

C. Unified Transformation

Both modalities use the same transformation logic:

```
(defmulti transform-ast (fn [ast env] (first ast)))

(defmethod transform-ast :program [[_ & exprs] env]
  (map #(transform-ast % env) exprs))

(defmethod transform-ast :list [[_ & elements] env]
  (apply (resolve (symbol (first elements)))
         (map #(transform-ast % env) (rest elements)
    )))

;; Additional methods for :symbol, :number, etc.
```

Listing 3. Unified AST Transformation

V. Evaluation

A. Quantitative Analysis

We compared the AST Convergence approach with traditional implementation across several metrics:

TABLE I
Comparison of Implementation Approaches

Metric	Traditional	AST Convergence
Transformation Functions	2	1
Code Duplication	High	None
Behavioral Consistency	Not Guaranteed	Guaranteed
Maintenance Points	2	1
Test Cases Required	2x	1x

B. Qualitative Benefits

- *Reduced Complexity*: Single transformation logic instead of two
- *Guaranteed Consistency*: Identical ASTs ensure identical semantics
- *Enhanced Maintainability*: Changes affect both modalities automatically
- *Improved Testability*: Transformation logic can be tested independently

VI. Theoretical Foundations

AST Convergence builds on several programming language theory concepts:

A. Homoiconicity

Clojure's homoiconicity—the property that code is represented as data—enables the internal DSL path. The =form-¿ast= function leverages this by treating Clojure forms as data that can be transformed into the target AST structure.

B. Language Homomorphisms

The pattern implements a homomorphism between the external and internal language modalities:

$$\phi : L_e \to L_i$$
$$\forall x \in L_e, \phi(P_e(x)) = M_i(\phi(x))$$

where both paths produce identical ASTs.

C. Multi-Modal Language Processing

AST Convergence provides a general framework for multi-modal language processing where different syntactic forms map to identical semantic representations.

VII. Applications and Future Work

A. Beyond FFmpeg DSLs

The AST Convergence pattern generalizes to numerous domains:

- *Configuration Systems*: Supporting both config files and programmatic configuration
- *Build Systems*: Declarative build files and programmatic build APIs
- *Query Languages*: Text queries and programmatic query construction
- *Template Engines*: Text templates and programmatic template generation

B. Future Research Directions

- Extending AST Convergence to dynamic languages beyond Clojure
- Formal verification of behavioral consistency
- Performance analysis of the convergence overhead
- Tooling support for AST Convergence pattern implementation

VIII. Conclusion

AST Convergence represents a significant advancement in multi-modal DSL implementation, elegantly solving the classic code duplication problem while ensuring behavioral consistency. By designing both external parsing and internal macro expansion to produce identical AST structures, the pattern enables a single transformation pipeline to handle multiple input modalities.

Our case study with the Bioscoop FFmpeg DSL demonstrates the practical benefits: reduced code complexity, guaranteed consistency, and improved maintainability. The pattern leverages Clojure's unique strengths while providing a generalizable approach applicable to diverse language implementation scenarios.

AST Convergence opens new possibilities for language design and implementation, particularly in domains requiring multiple interface modalities. Future work will explore extensions to other language ecosystems and formal verification of the consistency guarantees.

Acknowledgment

The author thanks the Clojure community for their contributions to the ecosystem that made this work possible.

References

[1] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 132–141, 2011.

[2] M. Fogus and C. Houser, *The Joy of Clojure*. Manning Publications, 2014.