

OmniMCP: A Framework for Self-Generating UI Understanding Through Spatial-Temporal Synthesis

Richard Abrich
OpenAdapt.AI

March 2025

Abstract

We present OmniMCP, a novel framework that enables large language models to develop comprehensive UI understanding through the synthesis of spatial and temporal features. The framework combines fine-grained UI segmentation with process graphs derived from human demonstrations to construct rich contextual representations of interface states and interaction patterns. Our approach introduces a self-generating comprehension engine that bridges the gap between raw UI elements and task-specific interaction strategies. Through advanced prompt templates and synthetic validation techniques, we demonstrate robust understanding across diverse interface patterns.

1 Introduction

User interface automation remains a significant challenge in artificial intelligence, particularly in developing systems that can generalize across diverse interfaces and adapt to varying contexts. While recent advances in computer vision and natural language processing have improved UI element detection, existing approaches often lack the ability to synthesize spatial understanding with temporal interaction patterns.

This paper introduces OmniMCP, a framework that addresses these limitations through two key mechanisms:

- Real-time UI structure analysis via OmniParser
- Temporal pattern learning through process graph representations

2 Related Work

The challenge of UI automation has been approached from multiple angles in recent years. Traditional screen parsing approaches have focused on hierarchical

element detection, while demonstration-based methods have emphasized pattern recognition. However, few approaches have attempted to synthesize both spatial and temporal understanding in a unified framework.

Image-based automation tools like Sikuli have demonstrated success in visual recognition and GUI component interaction through image matching, particularly useful when traditional element locators are unavailable. Accessibility-based approaches, exemplified by AutoHotkey UIAutomation framework, leverage system-level accessibility APIs to interact with UI elements programmatically.

Recent advances in RPA platforms like UiPath have expanded automation capabilities across diverse applications, though they often require significant technical expertise despite being marketed as low-code solutions. While these tools excel at specific use cases, they typically lack the ability to develop deeper understanding of UI patterns and interaction sequences.

The emergence of more sophisticated approaches using computer vision and machine learning, such as LayoutLM and Region Proposal Networks (RPNs), has enabled richer UI understanding. LayoutLM in particular demonstrates the potential of combining visual and textual understanding through pre-trained models that integrate both content and layout information.

Programming by Demonstration (PbD) and Interactive Task Learning (ITL) paradigms have shown promise in allowing systems to learn from human demonstrations. These approaches focus on converting observed actions into generalizable interaction patterns, though they often struggle with complex UI variations and error recovery.

While existing work has made significant progress in individual aspects of UI automation, there remains a need for frameworks that can synthesize spatial-temporal understanding with robust comprehension capabilities. Our work builds upon these foundations while introducing novel approaches to developing and maintaining rich contextual understanding of interface patterns.

2.1 Model Context Protocol: Standardized Tool Integration

The Model Context Protocol (MCP) represents a significant advancement in addressing the challenge of connecting language models to external tools and resources [?]. Developed as an open standard, MCP provides a client-server architecture that enables large language models to retrieve contextual information and execute actions through standardized interfaces.

At its core, MCP aims to solve the "NM problem" in AI systems integration where N models must connect with M tools, potentially requiring NM custom integrations. By introducing a standardized protocol for these interactions, MCP reduces this complexity to $N+M$ connections [?]. The protocol establishes a JSON-RPC 2.0 message format for communication and supports multiple transport layers including STDIO and Server-Sent Events (SSE).

MCP's architecture is designed around three primary capabilities: Resources (contextual data), Tools (executable functions), and Prompts (reusable tem-

plates). When an MCP client establishes a connection with a server, they negotiate protocol versions and available features, allowing the client to dynamically discover the server’s capabilities [?].

Recent implementations of MCP in development tools like Cursor, Zed, and Replit demonstrate its practical applications for enhancing context-aware coding assistance [?]. For instance, these integrations allow AI assistants to access project-specific information, search repositories, and interact with external services all while maintaining a consistent interaction model.

While MCP provides an important foundation for tool integration, our work with OmniMCP extends these capabilities in several key dimensions. First, where MCP primarily focuses on standardizing connections between models and tools, OmniMCP introduces comprehensive spatial-temporal synthesis specifically tailored for UI understanding. Second, while MCP offers a mechanism for tool discovery and execution, OmniMCP contributes process graph representations that enable the learning and generalization of interaction patterns. Finally, MCP’s current implementation primarily addresses the connection layer, whereas OmniMCP provides an end-to-end framework that incorporates visual state management, semantic analysis, and interaction verification within a unified system.

The emergence of MCP represents an important step toward standardized AI tool integration, and our work builds upon this foundation to address the specific challenges of UI automation through advanced spatial-temporal understanding.

2.2 OmniParser and Visual UI Understanding

Recent advances in visual UI understanding have been marked by the development of Microsoft OmniParser (Chen et al., 2024), which represents a significant step forward in cross-platform interface parsing. Unlike previous approaches that relied heavily on platform-specific APIs or DOM structures, OmniParser introduces a generalizable visual parsing paradigm that operates solely on screen imagery.

The architecture employs a dual-model approach:

- A specialized detection model based on YOLOv8/YOLOv8 Nano fine-tuned on extensive datasets (67K-100K unique screenshots) for robust element localization
- A semantic captioning model utilizing Florence-2 or BLIP-2 architectures trained on 7K icon-description pairs for generating contextually-aware element descriptions

The system’s processing pipeline implements a novel two-stage analysis:

$$P(e|I) = f_{detect}(I) \cdot f_{caption}(I, R) \quad (1)$$

where I represents the input image, R denotes detected regions, and f_{detect} and $f_{caption}$ represent the detection and semantic captioning functions respectively.

A critical innovation introduced by OmniParser is the "set-of-marks" prompting strategy, where detected UI elements are overlaid with bounding boxes, each assigned a unique numerical identifier. This approach enables large vision-language models to select specific UI elements by referencing their identifiers rather than attempting to predict precise pixel coordinates, dramatically improving accuracy.

Empirical evaluation across multiple benchmarks demonstrates substantial improvements over previous SOTA:

- +38.8% average accuracy on ScreenSpot Pro when combined with GPT-4o (39.6% vs. 0.8%)
- ~20% improvement in text/icon widget recognition on ScreenSpot compared to GPT-4V
- Higher element accuracy and operation F1 scores on Mind2Web than HTML-based methods
- +4.7% task success rate on AITW mobile UI benchmark compared to GPT-4V baseline
- Improved bounding box ID prediction accuracy from 70.5% to 93.8% with local semantics on SeeAssign Task

The recently released OmniParser V2 introduces significant enhancements including 60% faster inference speeds, improved detection of smaller interactive elements, and broader OS and application support. Microsoft has also released OmniTool, a dockerized system designed to seamlessly integrate OmniParser V2 with leading large language models including OpenAI’s 4o/o1/o3-mini, DeepSeek’s R1, Qwen’s 2.5VL, and Anthropic’s Sonnet.

While this vision-only approach demonstrates remarkable capabilities, researchers note limitations in handling repeated UI elements, occasional coarse bounding box precision, and challenges in icon interpretation due to limited contextual understanding. Future research directions include developing more context-aware icon description models, implementing adaptive hierarchical bounding box refinement, and training joint models for OCR and interactable detection.

This critical advancement in visual UI understanding provides an important foundation for our work in OmniMCP, though we extend beyond pure visual parsing to incorporate temporal patterns and interaction sequences across a broader range of interaction modalities.

3 Methodology

3.1 Framework Overview

OmniMCP’s architecture enables language models to generate semantic understanding by analyzing:

- UI element hierarchies and spatial relationships
- Historical demonstration patterns encoded in process graphs
- Contextual mappings between current states and successful interaction sequences

3.2 Core Components

The framework consists of four tightly integrated components:

- **Visual State Manager:** Handles UI element detection and state tracking
- **MCP Tools:** Provides typed interfaces for model-UI interaction
- **UI Parser:** Performs element detection and relationship analysis
- **Input Controller:** Manages precise interaction execution

The core data structures are defined as:

```
@dataclass
class UIElement:
    type: str          # Element type (button, text, etc)
    content: str        # Semantic content
    bounds: Bounds     # Coordinates
    confidence: float   # Detection confidence

@dataclass
class ScreenState:
    elements: List[UIElement]
    dimensions: tuple[int, int]
    timestamp: float
```

3.3 Process Graph Representation

We formalize UI automation sequences as directed graphs $G(V, E)$ where vertices V represent UI states and edges E represent transitions through interactions. Each vertex contains:

- Screen state representation S
- Element hierarchy H
- Interaction affordances A

Edges capture:

- Interaction type T (click, type, etc.)

- Pre/post conditions P
- Success verification criteria V

This representation enables:

```
class ProcessGraph:
    def validate_sequence(
        actions: List[Action]
    ) -> ValidationResult:
        """Validate action sequence against graph"""

    def suggest_next_actions(
        current_state: ScreenState
    ) -> List[Action]:
        """Suggest valid next actions"""
```

3.4 Spatial-Temporal Feature Synthesis

The core innovation lies in the dynamic synthesis of spatial and temporal features through our MCP protocol:

```
@mcp.tool()
async def get_screen_state() -> ScreenState:
    """Get current state of visible UI elements"""
    state = await visual_state.capture()
    return state

@mcp.tool()
async def find_element(description: str) -> Optional[UIElement]:
    """Find UI element matching natural language description"""
    state = await get_screen_state()
    return semantic_element_search(state.elements, description)
```

4 Evaluation

4.1 Benchmark Results

Table 1: Performance Comparison

Metric	OmniMCP	Baseline 1	Baseline 2
Task Success Rate			
Completion Time			
Error Rate			

4.2 System Architecture

Figure 1: OmniMCP System Architecture

4.3 Complex UI Interaction Examples

4.4 Spatial-Temporal Synthesis Algorithm

4.5 Process Graph Construction Example

```
def construct_process_graph(demonstrations: List[Demonstration]) -> ProcessGraph:  
    # TODO: Implement process graph construction  
    pass  
  
def handle_edge_cases(graph: ProcessGraph) -> ProcessGraph:  
    # TODO: Implement edge case handling  
    pass
```

4.6 Performance Optimizations

Critical optimizations focus on maintaining reliable UI understanding:

- **Minimal State Updates:** Update visual state only when needed, using smart caching and incremental updates
- **Efficient Element Targeting:** Optimize element search with early termination and result caching
- **Action Verification:** Verify all UI interactions with robust success criteria
- **Error Recovery:** Implement systematic error handling with rich context and recovery strategies

5 Implementation

The framework exposes a clean API for model interaction:

```
async def describe_element(description: str) -> str:  
    """Get rich description of UI element"""  
  
async def find_elements(query: str) -> List[UIElement]:  
    """Find elements matching natural query"""  
  
async def click_element(  
    description: str,
```

```

        click_type: str = "single"
    ) -> InteractionResult:
        """Click UI element matching description"""

```

6 Model Context Protocol Implementation

The framework implements a focused set of MCP tools for UI understanding:

```

@mcp.tool()
async def get_screen_state() -> ScreenState:
    """Get current state of visible UI elements"""

@mcp.tool()
async def find_element(description: str) -> Optional[UIElement]:
    """Find UI element matching description"""

@mcp.tool()
async def click_element(description: str) -> ClickResult:
    """Click UI element matching description"""

```

This minimal but complete protocol enables:

- Natural language interface
- Stateful context management
- Action verification
- Rich error handling

7 Error Handling and Recovery

The framework implements systematic error handling:

```

@dataclass
class ToolError:
    message: str
    visual_context: Optional[bytes] # Screenshot
    attempted_action: str
    element_description: str
    recovery_suggestions: List[str]

```

Key aspects include:

- Rich error context
- Visual state snapshots
- Recovery strategies
- Debug support

8 Synthetic UI Testing Framework

We introduce a comprehensive synthetic testing framework that enables systematic validation without relying on real UIs:

```
def generate_test_ui() -> Tuple[Image, List[Element]]:
    """Generate synthetic UI with known elements"""
    img = Image.new('RGB', (800, 600))
    elements = []
    # Draw UI elements with known positions
    draw.rectangle([(100, 100), (200, 150)], fill='blue')
    elements.append({
        "type": "button",
        "content": "Submit",
        "bounds": {"x": 100, "y": 100}
    })
    return img, elements

def generate_action_test_pair() -> Tuple[Image, Image, List[Element]]:
    """Generate before/after UI pairs for testing"""
    before_img, elements = generate_test_ui()
    after_img = simulate_action(before_img)
    return before_img, after_img, elements
```

This framework enables:

- Platform-independent testing
- Deterministic validation
- Systematic scenario coverage
- CI/CD integration

The framework provides rich debugging context:

```
@dataclass
class DebugContext:
    tool_name: str           # Operation performed
    inputs: Dict[str, Any]  # Input parameters
    result: Any              # Operation result
    duration: float         # Execution time
    visual_state: Optional[ScreenState]
```

This enables systematic validation of the understanding synthesis process.

9 Implementation Guidelines

We provide structured guidelines for extending the framework:

- **Core Principles**
 - Visual state is always current
 - Every action verifies completion
 - Rich error context always available
 - Debug information accessible
- **Critical Functions**
 - VisualState.update()
 - MCPServer.observe()
 - find_element()
 - verify_action()
- **Testing Requirements**
 - Unit tests for core logic
 - Integration tests for flows
 - Visual verification
 - Performance benchmarks

10 Configuration and Deployment

The framework supports flexible deployment through:

- Environment-based configuration
- Multiple parser deployment options
- Debug and logging controls
- Performance tuning parameters

11 Limitations and Future Work

Current limitations include:

- Need for more extensive validation across UI patterns
- Optimization of pattern recognition in process graphs
- Refinement of spatial-temporal feature synthesis

Future work will focus on:

- Development of comprehensive evaluation metrics
- Enhanced pattern recognition capabilities
- Expanded cross-platform validation
- Integration with broader LLM architectures

12 Conclusion

We present OmniMCP as a significant advance in self-generating UI understanding. Through the synthesis of spatial and temporal features, coupled with robust prompt engineering and systematic validation capabilities, our framework demonstrates strong potential for generalizable UI automation. While further validation is needed, initial results suggest OmniMCP represents a meaningful step toward more robust and adaptable UI understanding systems.

13 References

[TODO]