OmniMCP: A Spatial-Temporal Framework for Generalizable UI Automation

Richard Abrich
OpenAdapt.AI

March 2024

Abstract

We present OmniMCP, an innovative framework for enabling 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 dynamic feature synthesis.

OmniMCP leverages the Model Context Protocol (MCP) and Microsoft's OmniParser to provide AI models with deep UI understanding capabilities. This white paper details the framework's architecture, the spatial-temporal synthesis methodology, and the implementation of MCP tools that enable robust UI automation. While still under active development, key components have been implemented and show promising results in preliminary testing. We also discuss future research directions, including reinforcement learning integration and specialized model fine-tuning.

Our framework not only addresses current UI automation challenges but also lays the foundation for future work integrating reinforcement learning approaches, where the spatial-temporal representations can serve as a structured state space for learning optimal interaction policies through synthetic environments and LLM-guided reward functions.

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.

OmniMCP addresses these limitations through two key mechanisms:

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

Our key contributions include:

- A novel spatial-temporal synthesis framework that combines fine-grained UI parsing with interaction process graphs to create contextual understanding of interfaces
- A self-generating comprehension engine that bridges raw UI elements with task-specific strategies through dynamic feature synthesis
- An extensible MCP-based architecture enabling LLMs to develop, maintain and verify UI understanding
- A synthetic validation framework design that enables systematic testing of automation reliability across diverse UI patterns

Looking forward, we also explore how this spatial-temporal synthesis framework provides an ideal foundation for reinforcement learning approaches to UI automation. The structured representation of interface states and interactions creates a well-defined action space for RL algorithms, while our process graphs offer a mechanism for warm-starting policies based on human demonstrations.

2 Core Framework Components

OmniMCP's architecture integrates four tightly integrated components:

2.1 Visual State Manager

The Visual State Manager handles UI element detection, state management and caching, rich context extraction, and history tracking. It provides a comprehensive representation of the current UI state, enabling models to reason about available elements and their relationships.

2.2 MCP Tools Layer

The MCP Tools layer implements the Model Context Protocol to provide standardized interfaces for LLM interaction with the UI automation system. This includes tool definitions and execution, typed responses, error handling, and debug support. The layer ensures consistent and reliable communication between models and the UI automation system.

2.3 UI Parser

The UI Parser leverages OmniParser for element detection, text recognition, visual analysis, and understanding element relationships. It transforms raw screen captures into structured representations that models can reason about effectively.

2.4 Input Controller

The Input Controller manages precise mouse control, keyboard input, action verification, and movement optimization. It ensures reliable execution of model-directed actions and provides verification that operations completed successfully.

3 The Spatial-Temporal Understanding Pipeline

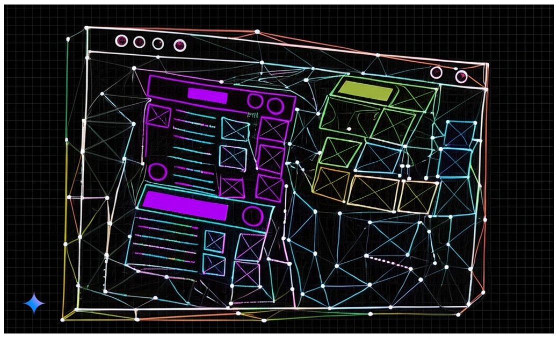

Figure 1: Spatial Feature Understanding

3.1 Spatial Feature Understanding

OmniMCP begins by developing a deep understanding of the user interface's visual layout. Leveraging microsoft/OmniParser, it performs detailed visual parsing, segmenting the screen and identifying all interactive and informational elements. This includes recognizing their types, content, spatial relationships, and attributes, creating a rich representation of the UI's static structure.

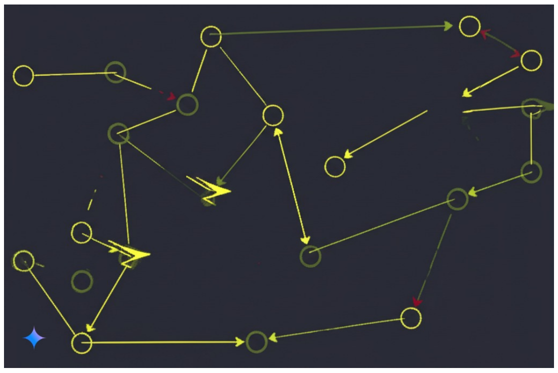

Figure 2: Temporal Feature Understanding

3.2 Temporal Feature Understanding

To capture the dynamic aspects of the UI, OmniMCP tracks user interactions and the resulting state transitions. It records sequences of actions and changes within the UI, building a Process Graph that represents the flow of user workflows. This temporal understanding allows AI models to reason about interaction history and plan future actions based on context.

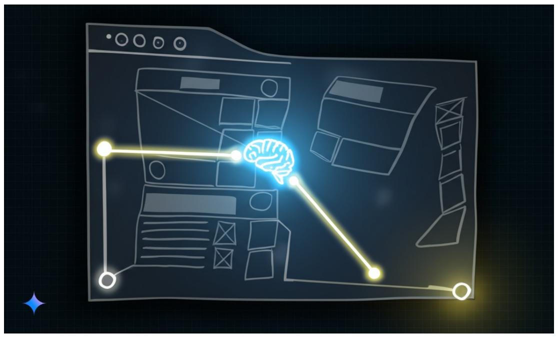

Figure 3: Internal API Generation

3.3 Internal API Generation

Utilizing the rich spatial and temporal context it has acquired, OmniMCP leverages a Large Language Model (LLM) to generate an internal, context-specific API. Through In-Context Learning (prompting), the LLM dynamically creates a set of functions and parameters that accurately reflect the understood spatiotemporal features of the UI. This internal API is tailored to the current state and interaction history, enabling precise and context-aware interactions.

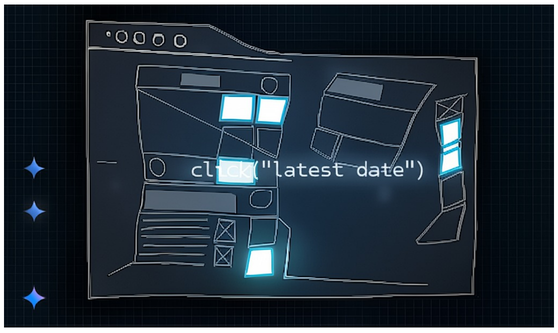

Figure 4: External API Publication (MCP)

3.4 External API Publication (MCP)

Finally, OmniMCP exposes this dynamically generated internal API through the Model Context Protocol (MCP). This provides a consistent and straightforward interface for both humans (via natural language translated by the LLM) and AI models to interact with the UI. Through this MCP interface, a full range of UI operations can be performed with verification, all powered by the AI model's deep, dynamically created understanding of the UI's spatiotemporal context.

4 Process Graph Representation

A key innovation in OmniMCP is the formalization of UI automation sequences as directed graphs G(V, E) where:

- Vertices V represent UI states, containing:
  - Screen state representation
  - Element hierarchy
  - Interaction affordances
- Edges E represent transitions through interactions, capturing:
  - Interaction type (click, type, etc.)
  - Pre/post conditions
  - Success verification criteria

This representation enables:

- Validation of proposed action sequences
- Suggestion of likely next actions based on current state
- Detection of deviations from expected patterns
- Transfer of knowledge between similar workflows

5 Spatial-Temporal Feature Synthesis

The core innovation lies in the dynamic synthesis of spatial and temporal features through our MCP protocol. We formalize this synthesis process as a function Φ that maps a current screen state S_t and a historical interaction sequence H_t to a contextual understanding representation U_t:

$$U_t = \Phi(S_t, H_t) \tag{1}$$

Where S_t represents the spatial information at time t including element positions, types, and hierarchical relationships, while H_t encapsulates the temporal sequence of interactions leading to the current state. The function Φ is implemented as a two-stage process:

$$\Phi(S_t, H_t) = f_{context}(f_{spatial}(S_t), f_{temporal}(H_t)) \tag{2}$$

The spatial function $f_{spatial}$ extracts hierarchical relationships between elements:

$$f_{spatial}(S_t) = \{e_i, r_{ij} | e_i \in E_t, r_{ij} \in R\} \tag{3}$$

Where E_t is the set of UI elements at time t and R is the set of spatial relationships (contains, adjacent-to, etc.)

The temporal function $f_{temporal}$ identifies patterns in the interaction history:

$$f_{temporal}(H_t) = \{p_k | p_k \in P, p_k \subset H_t\} \tag{4}$$

Where P is the set of known interaction patterns derived from process graphs.

Finally, the context function $f_{context}$ synthesizes these features into a unified representation that enables accurate action prediction:

$$a_{t+1} = \arg\max_{a \in A} P(a|U_t) \tag{5}$$

6 Implementation and API

OmniMCP provides a powerful yet intuitive API for model interaction through the Model Context Protocol (MCP). This standardized interface enables seamless integration between large language models and UI automation capabilities.

```
from omnimcp import OmniMCP
from omnimcp.types import UIElement, ScreenState, InteractionResult

async def main():
    mcp = OmniMCP()

    # Get current UI state
    state: ScreenState = await mcp.get_screen_state()

    # Analyze specific element
    description = await mcp.describe_element(
        "error message in red text"
    )
    print(f"Found element: {description}")

    # Interact with UI
    result = await mcp.click_element(
        "Submit button",
        click_type="single"
    )
    if not result.success:
        print(f"Click failed: {result.error}")

asyncio.run(main())
```

6.1 Core Types

OmniMCP implements clean, typed responses using dataclasses:

```
@dataclass
class UIElement:
    type: str            # button, text, slider, etc
    content: str         # Text or semantic content
    bounds: Bounds       # Normalized coordinates
    confidence: float    # Detection confidence
    attributes: Dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> Dict:
        """Convert to serializable dict"""

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

    def find_elements(self, query: str) -> List[UIElement]:
        """Find elements matching natural query"""

@dataclass
class InteractionResult:
    success: bool
    element: Optional[UIElement]
    error: Optional[str] = None
    context: Dict[str, Any] = field(default_factory=dict)
```

6.2 MCP Tools Implementation

The framework implements a comprehensive set of MCP tools designed specifically for robust UI understanding and automation:

```
@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)

@mcp.tool()
async def take_action(description: str, image_context: Optional[bytes] = None) -
    """Execute action described in natural language with optional visual context
    # Parse the natural language description into structured action
    action = await action_parser.parse(description, image_context)

    # Execute the appropriate action based on type
    if action.type == "click":
        element = await find_element(action.target)
        if not element:
            return ActionResult(success=False, error=f"Element␣not␣found:␣{actio
        return await input_controller.click(element, action.parameters.get("clic

    # Additional action types (type, drag, etc.) handled similarly

@mcp.tool()
async def wait_for_state_change(
    description: str = "any␣change",
    timeout: float = 10.0
) -> WaitResult:
    """Wait for UI state to change according to description within timeout"""
    start_time = time.time()
    initial_state = await get_screen_state()

    while time.time() - start_time < timeout:
        current_state = await get_screen_state()
        match = await state_matcher.matches_description(current_state, descriptio
        if match:
            return WaitResult(success=True, state=current_state, match_details=m
        await asyncio.sleep(0.5)

    return WaitResult(success=False, error=f"Timeout␣waiting␣for:␣{description}"

@mcp.tool()
async def get_interaction_history() -> List[Interaction]:
    """Get recent interaction history"""
    return await process_graph.recent_interactions()

@mcp.tool()
async def predict_next_actions(current_state: Optional[ScreenState] = None) -> L
    """Predict possible next actions based on process graph and current state"""
    if current_state is None:
        current_state = await get_screen_state()
    return await process_graph.suggest_next_actions(current_state)

@mcp.tool()
async def verify_action_result(action: str, expected_result: str) -> Verificatio
    """Verify that an action produced the expected result"""
    current_state = await get_screen_state()
    return await verification_engine.verify(action, expected_result, current_sta
```

7 Performance Optimizations

Critical optimizations in OmniMCP 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

```
@dataclass
class DebugContext:
    """Rich debug information"""
    tool_name: str
    inputs: Dict[str, Any]
    result: Any
    duration: float
    visual_state: Optional[ScreenState]
    error: Optional[Dict] = None

    def save_snapshot(self, path: str) -> None:
        """Save debug snapshot for analysis"""
```

8 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

9 Future Research Directions

9.1 Reinforcement Learning Integration

A promising direction for future research is the integration of LLM-guided reinforcement learning to achieve superhuman performance in UI automation tasks. This approach would enhance OmniMCP's spatial-temporal framework with adaptive learning capabilities.

We envision formulating UI automation as a Markov Decision Process (MDP) where states are ScreenState objects, actions correspond to MCP tool operations, and rewards quantify interaction effectiveness. Rather than traditional RL algorithms, we propose Direct Preference Optimization (DPO) using LLMs to generate and evaluate interaction preferences.

Key components of this future enhancement would include:

1. **Synthetic Data Generation**: Creating safe exploration spaces through LLM-generated UI variations, enabling the policy to learn from diverse scenarios without operational risks.

2. **Process Graph-Augmented Memory**: Extending beyond the Markovian assumption to capture long-term patterns in successful UI interactions, storing embeddings of key states and retrieving similar interaction patterns when encountering familiar UI states.

3. **LLM as Reward Function**: Employing the LLM itself as a contextualized reward function through structured evaluation prompts, considering progress toward goals, interaction efficiency, robustness, and safety.

4. **Forecasting-Enhanced Exploration**: Introducing predictive modeling of future UI states resulting from potential action sequences, enabling mental simulation of multi-step interactions without execution.

The training pipeline would consist of initialization from process graphs, refinement through synthetic UI interactions, and final tuning with limited real application interaction. This approach would enable OmniMCP to develop superhuman automation capabilities while maintaining interpretability and safety guarantees.

9.2 Additional Research Directions

Beyond reinforcement learning integration, we plan to explore:

- **Fine-tuning Specialized Models**: Training domain-specific models on UI automation tasks to improve efficiency and reduce token usage. By fine-tuning models on extensive datasets of UI interactions, we can potentially create more efficient specialists that require fewer tokens to understand context and generate appropriate actions.
- **Process Graph Embeddings with RAG**: Embedding generated process graph descriptions and retrieving relevant interaction patterns via Retrieval Augmented Generation. By creating a searchable knowledge base of UI patterns and successful interaction strategies, we can enable the system to efficiently query similar past experiences when encountering new UI states.
- **Development of Comprehensive Evaluation Metrics**: Creating standardized benchmarks and metrics for evaluating UI automation systems across dimensions of reliability, efficiency, and generalization capabilities.
- **Enhanced Cross-Platform Generalization**: Extending the framework to handle interactions across diverse operating systems and platforms with unified understanding.
- **Integration with Broader LLM Architectures**: Exploring how OmniMCP can be integrated with emerging LLM architectures to leverage improvements in reasoning and visual understanding.
- **Collaborative Multi-Agent UI Automation**: Developing approaches where multiple specialized agents coordinate on complex tasks, each handling specific aspects of UI interaction.

```
class ProcessGraphRAG:
    def __init__(self, embedding_model):
        self.embedding_model = embedding_model
        self.vector_store = VectorDatabase()
    async def index_process_graph(self, graph, description):
        """Index process graph with description for retrieval"""
        embedding = self.embedding_model.embed(
            description + "\n" + graph.to_text()
        )
        self.vector_store.add(
            embedding,
            metadata={"graph_id": graph.id, "description": description}
        )

    async def retrieve_similar_graphs(self, current_state, task):
        """Retrieve relevant process graphs for current context"""
        query = f"Task: {task}\nUI State: {current_state.summarize()}"
        query_embedding = self.embedding_model.embed(query)

        matches = self.vector_store.search(
            query_embedding,
            k=5,
            threshold=0.75
        )

        return [self.load_graph(match.metadata["graph_id"])
                for match in matches]

    async def augment_context(self, llm_context, current_state, task):
        """Augment LLM context with relevant process graphs"""
        similar_graphs = await self.retrieve_similar_graphs(
            current_state, task
        )

        augmented_context = llm_context + "\n\nRelevant interaction patterns:\n"
        for graph in similar_graphs:
            augmented_context += f"- {graph.summarize()}\n"

        return augmented_context
```

10 Project Status and Limitations

OmniMCP is under active development with core components implemented and functional. The current implementation demonstrates promising capabilities in UI understanding and interaction, though several aspects remain under development:

10.1 Current Capabilities

- Visual element detection and analysis using OmniParser
- MCP protocol implementation for LLM interaction
- Basic process graph construction from demonstrations
- Reliable UI interaction with verification

10.2 Current Limitations

- Need for more extensive validation across diverse UI patterns
- Optimization of pattern recognition in process graphs
- Refinement of spatial-temporal feature synthesis
- Limited testing across different operating systems and application types

11 Installation and Configuration

```
pip install omnimcp

# Or from source:
git clone https://github.com/OpenAdaptAI/omnimcp.git
cd omnimcp
./install.sh
```

Configuration can be managed through environment variables or a .env file:

```
# .env or environment variables
OMNIMCP_DEBUG=1                    # Enable debug mode
OMNIMCP_PARSER_URL=http://...  # Custom parser URL
OMNIMCP_LOG_LEVEL=DEBUG        # Log level
```

12 Conclusion

OmniMCP represents a significant advancement in self-generating UI understanding through the synthesis of spatial and temporal features. By combining the powerful visual parsing capabilities of Microsoft's OmniParser with our novel process graph approach for temporal context, the framework enables more robust and adaptable UI automation than previously possible.

While development is ongoing, the current implementation demonstrates the viability of our approach. The integration of the Model Context Protocol provides a standardized interface for LLMs to develop and maintain UI understanding, while our typed response system ensures reliable operation across diverse interfaces.

The framework's architecture was intentionally designed not only as a solution to current UI automation challenges but also as a foundation for future reinforcement learning approaches. The spatial-temporal representations?particularly the process graph structure and state formalization?provide an ideal abstraction layer for RL algorithms.

As we continue development, OmniMCP will evolve into an increasingly powerful tool for generalizable UI automation, enabling more natural and reliable human-computer interaction through the combination of advanced visual understanding and temporal context.

13 Contact

- Repository: `https://github.com/OpenAdaptAI/omnimcp`
- Issues: GitHub Issues
- Questions: Discussions
- Security: security@openadapt.ai

Note: OmniMCP is under active development. While core components are functional, some features described in this white paper are still being refined and extended. The API may change as development progresses.