Thinking…

…*inside* the box

Behavior Design Patterns: Engineering Human Behavior Representations

Glenn Taylor Robert Wray

Soar Workshop 24

10 June 2004

Research sponsored by Defense Advanced Research Projects Agency, Contract DAAH01-03-C-R242

Soar Technology
Thinking *inside* the box.

Soar Technology
Thinking *inside* the box.

Problem

- Reuse of HBR knowledge representations is difficult
  - even when tasks/domains are similar
- We recognize as behavior developers that there are recurring design problems & solutions
- How do we more easily *reuse* previous solutions?
  - Focus: Previous designs

Foundations

- Design Patterns (Gamma, et al)
  - Abstractions to address common OO reuse problems
  - Independent of specific application domains
  - Iterator, proxy, factory method, etc…
- Generic Tasks (Chandrasekaran)
  - General patterns of problem solving and representation: Hierarchical classification, hypothesis matching, knowledge-directed information passing, synthesis by plan selection and refinement, abductive hypothesis assembly
- Taxonomy of Human Behaviors (Fineberg)
  - Catalogue of human behavior “primitives”: Sensation, mediation, reaction, interaction

Assumptions

- Design patterns may not be applicable for cognitive models
  - Architectural dependencies & idiom
  - Quantitative human data drives low-level details
- Design patterns appear to be relevant in human behavior representation
  - Focus often more on knowledge level behavior than immediate behavior ("psychology")
  - Qualitative/descriptive validation is the norm
  - Aggregations (entity vs. individual)
- "Engineering" philosophy
  - Define & capture recurrent behavior design patterns
  - Use cognitive architectures to guide solution patterns (psychological constraint)

Design Patterns

- Reusable software elements that describe a common problem and solution (Gamma et al.)
  - Generally:
    - 1) pattern name (for communication/reference)
    - 2) clear definition of the problem
    - 3) clear definition of the solution
    - 4) consequences of using the pattern
  - Pattern captures aspects of behavior/structure that are invariant, and call out/encapsulate aspects that vary
  - DPs language-neutral; details/cost may differ across languages/architectures
  - Specific paradigm (OOP)
  - Specific goal (reuse)

Design Pattern example

- Name: Strategy
- Problem Addressed:
  - Need multiple variants of an algorithm
  - Have many modules that execute similar algorithms
- Solution:
  - Encapsulate a family of related algorithms from the objects that invoke them. Allows implementation to vary independently from object
- Consequences:
  - Reduces conditional statements in modules
  - Increased communication overhead
- Example: different memory management styles, etc.

Strategy pattern is similar to the notion of a generic task

“Behavior” Design Patterns (BDPs)

- Patterns that describe common problems and solutions in building human behavior representations
- Human behavior representation programming paradigm (from cognitive architectures):
  - Agent must strike balance between reactive and goal-directed behavior
    - Associative control flow (re-entrant execution is a basic requirement)
  - Least-commitment execution
    - Run-time decision making and conflict resolution
    - Weak encapsulation
  - Large, (possibly) changing knowledge bases
  - Human fidelity constraints

Classes of BDPs (1)

- Architectural: patterns for underlying processes, assumptions, constraints
  - Pattern matching, automatic subgoaling, etc.
  - Cognitive architectures
    - assume patterns of processing are fixed (or vary parametrically)
    - can be viewed as attempts to define collections of architectural patterns that are sufficient to describe human behavior

Classes of BDPs (2)

- Computational patterns: capture and represent low-level, domain-independent recurring computations
  - Iteration (perform f(x) on all objects x with property y)
  - Deliberate memory management
  - Compute vs. retrieve decisions
  - Process annotation
- Similar to (Gamma et al) patterns:
  - Abstractions from common problems arising from a particular language/design paradigm
  - May indicate directions for language/architecture evolution

Classes of BDPs (3)

- Behavior-level: knowledge, processes, tasks related to the specific individual and task being modeled
  - *Domain Patterns* – focus on domain/class-specific problems/tasks
    - Examples: Mission (air-to-ground attack), tactical (maneuver)
  - *Interface Patterns* – focus on entity interactions
    - Virtual vehicles, communications interfaces (eg, FIPA-ACL), perception, proprioception, etc.

Potential benefits of behavior level BDPs?

- Provide language neutrality
  - Focus on domain-specific (task) patterns, not patterns of implementation within a language
  - Transfer of design to other platforms
- Improve understandability
  - Concise descriptions of solutions
  - Make design elements & decisions explicit
  - Increased transparency (to developers & users)
- Potential results:
  - More reuse
  - Devote more effort to novel aspects of behavior representation
  - Improve capability/investment ratio

Describing patterns?

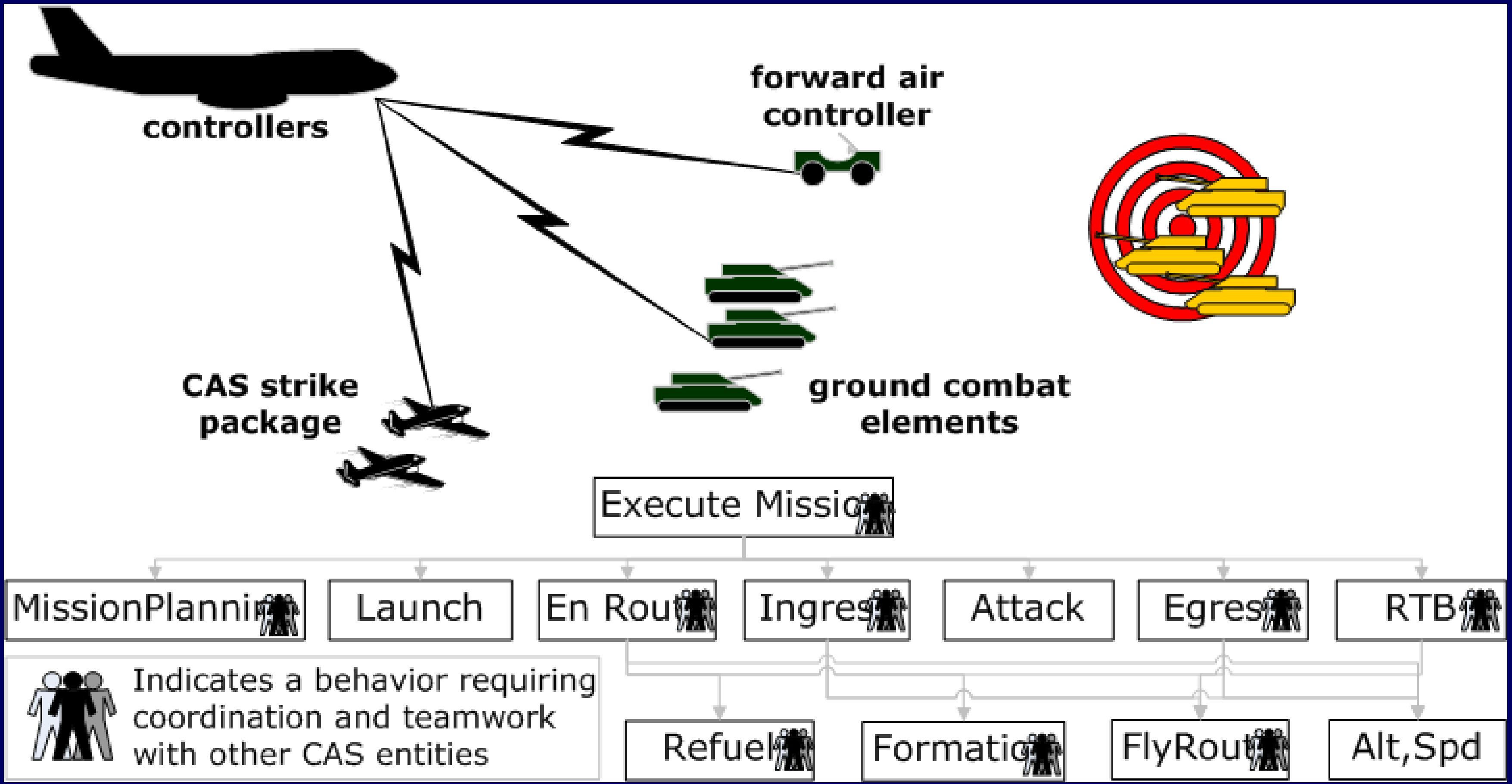
- Natural Language
- Diagrams
  - UML (State Diagrams, Sequence Diagrams, etc.) is standard in OOP Software Engineering, may not have right primitives; AUML may not be expressive enough
  - Bottom line: Multiple diagrammatic views are needed for many patterns
- Language-specific examples
  - Multiple language implementations of the same pattern can help make design trade-offs more explicit

- All are needed to fully capture a pattern for reuse

Patterns in existing HBRs

- Analysis of TacAir-Soar
  - Comprehensive suite of FWA missions and aircraft: Tactical air combat, close-air support, refueling, etc.
  - Focus: close-air support mission
    - mission phases, roles, entities, domain concepts
- Methodology
  - Look for generalities, repetition in processes, structures
  - Isolate functionality, encapsulate, generalize
  - Describe as patterns

Close-air support



Soar Technology
Thinking *inside* the box.

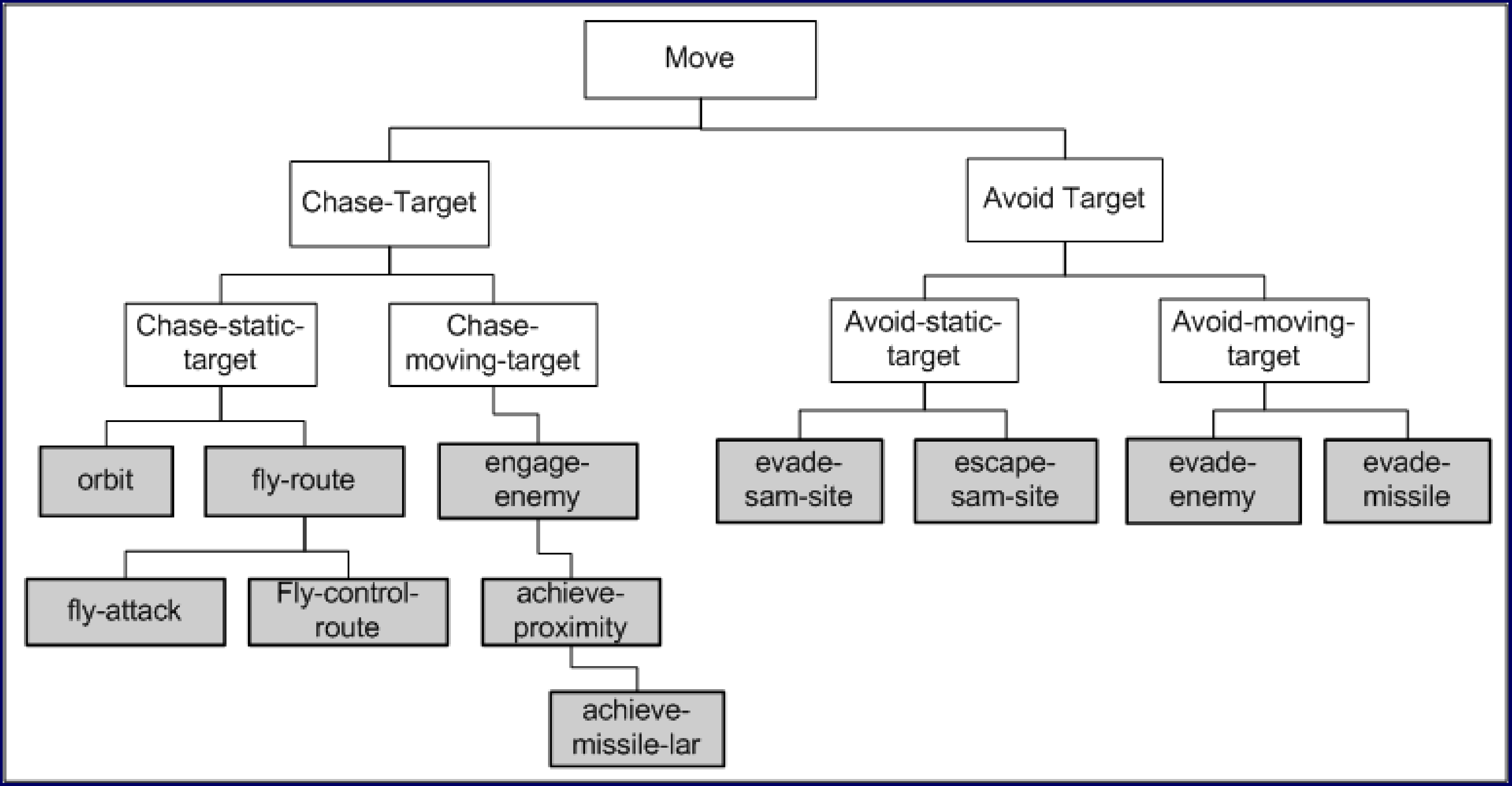
TacAir-Soar analysis

- Assigned individual knowledge representations into eight classes:
  - Communications: How and when to talk to other agents, and how to interpret incoming communications
  - Missions: Mission-specific behaviors, such as for air-to-ground missions (CAS) or air-to-air missions (DCA)
  - Control: How to direct other units to take action
  - Coordination: How to work with other friendly agents
  - Flying: How to fly a plane
  - Navigation: How to decide where to go and how to get there
  - Situational Awareness: How to manage information about the environment
  - Action: Atomic interactions with the simulation platform
- Are there patterns within the categories?

TacAir-Soar analysis

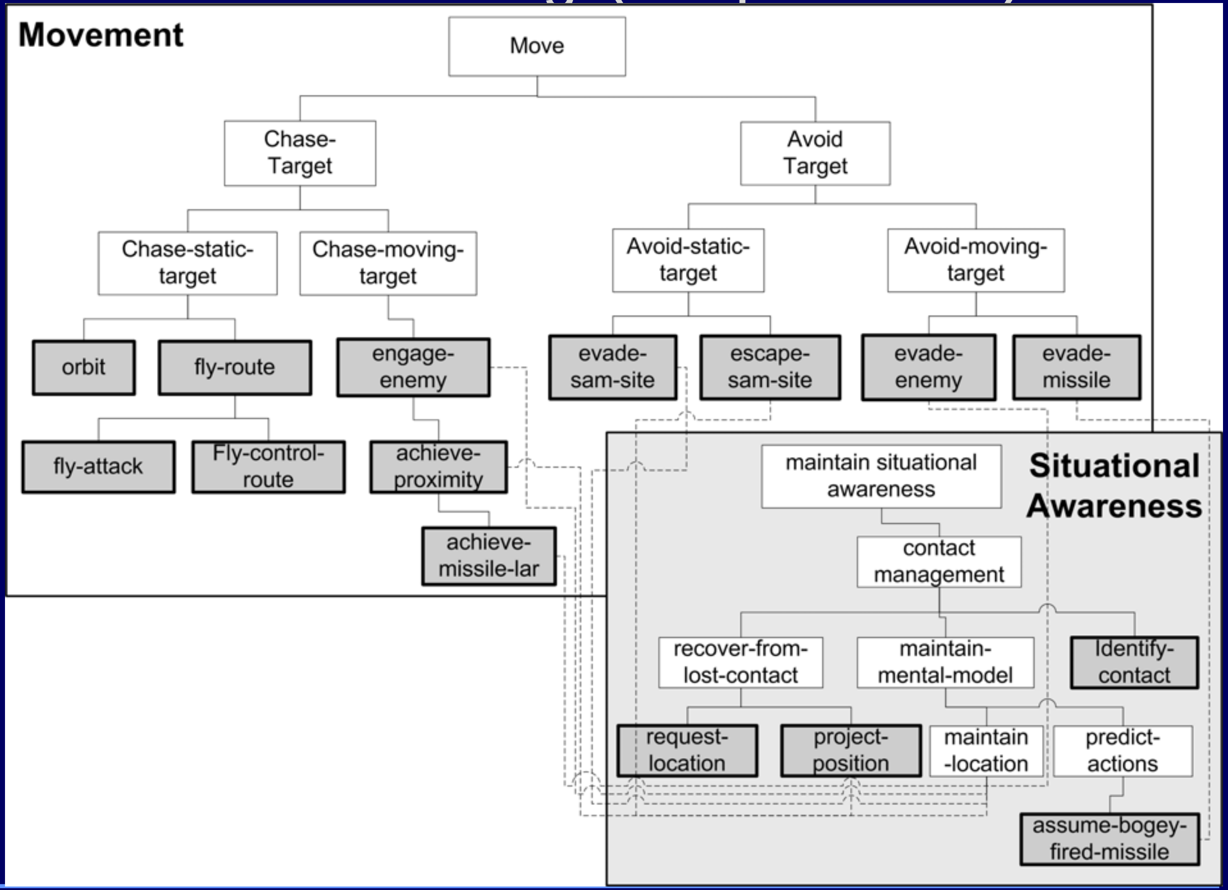
- Potential patterns at many different levels of abstraction:
  - Mission-level patterns
    - Air-to-ground attack
  - Tactical patterns
    - Chase-target
  - Multi-agent interaction patterns
    - Directive pattern (command/response)
    - Patterns adopted from MAS/AOP community (ACLs, conversation policies)
  - Information processing patterns
    - Situation awareness
  - Interaction patterns
    - Vehicle/movement control

Movement behaviors



Behavior interactions

- Many interactions between behaviors/patterns
- Many patterns are cross-cutting ("aspectual")
  - comms
  - situational awareness

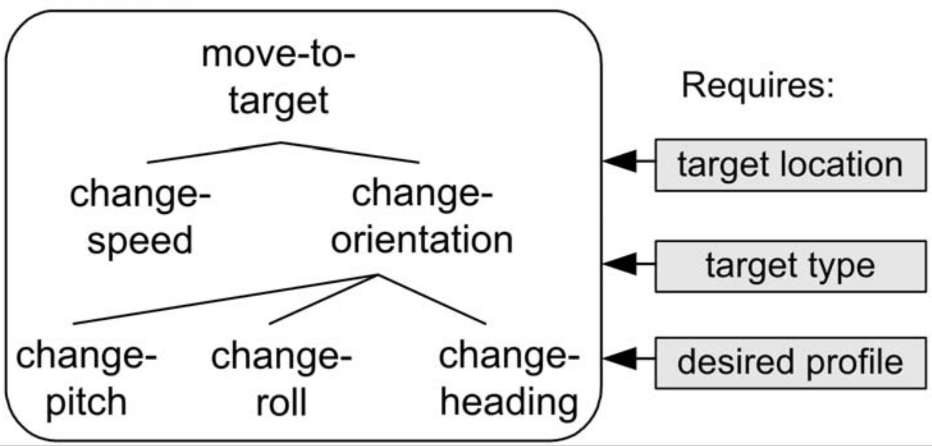


Chase-Target pattern

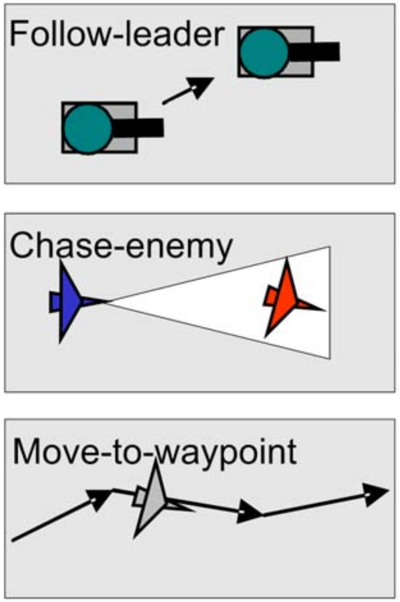
Problem: many types of movement behaviors, each with variations in unit type, orientation, etc.

Consequences: simplifies movement behavior; allows one movement base behavior with variations, rather than multiple separate types of movement

Structure



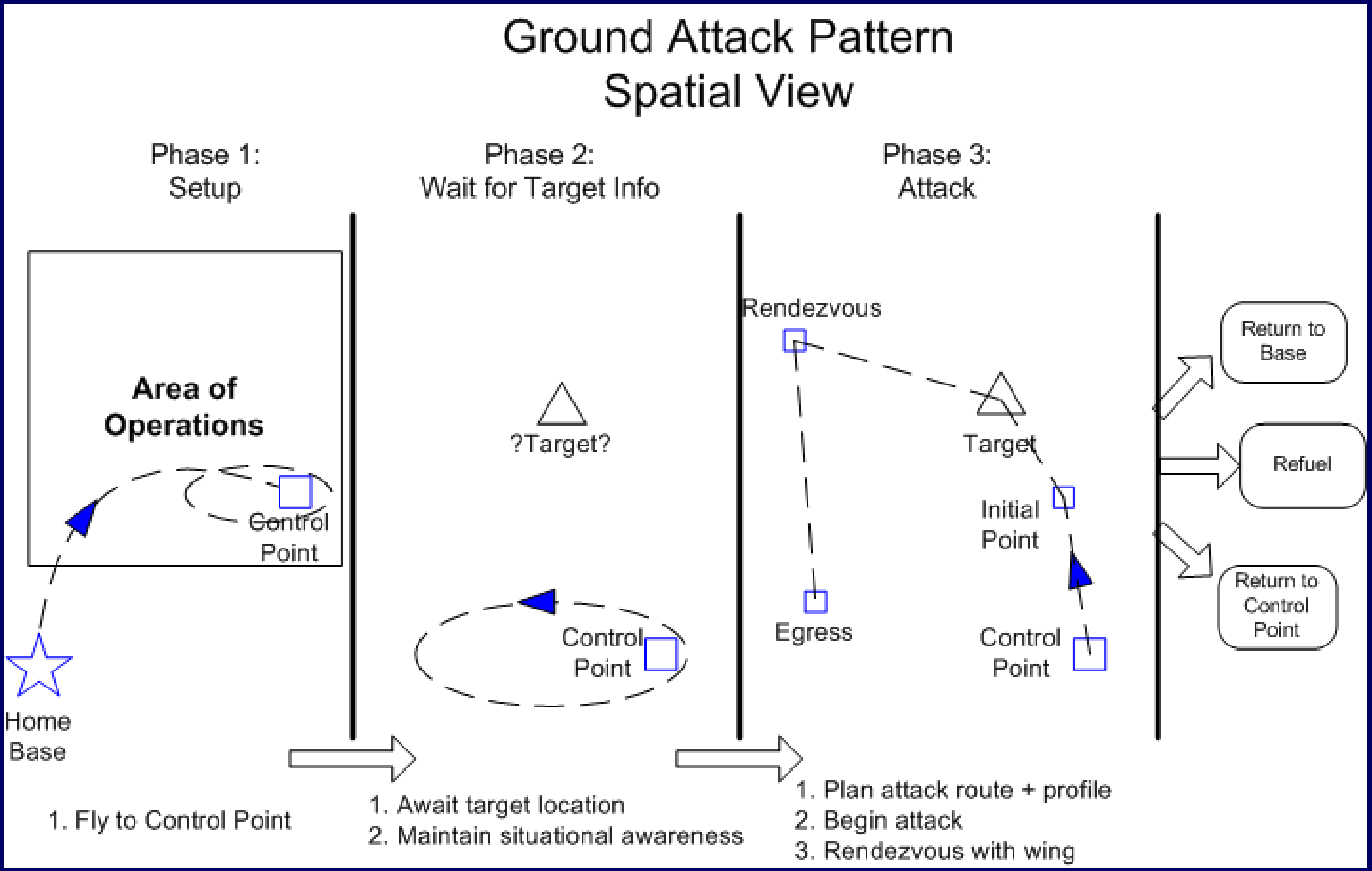
Used By:



Variations:

- Unit type
- Degrees of Freedom

Air-to-Ground Attack

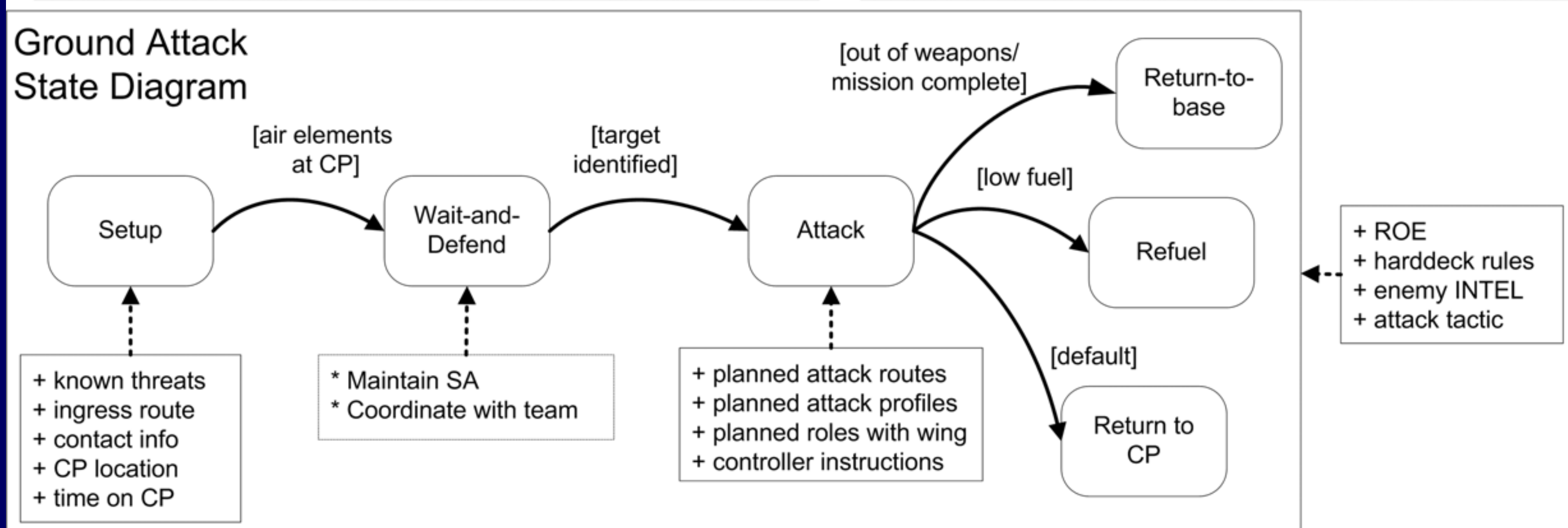


Air-to-Ground Attack

Pattern: Ground Attack

Problem:
Multiple variations on the ground attack theme tend to create a slew of specialized ground attack behaviors. Want to isolate the invariants and allow template-based specialization.

Consequences:
Cleaner separation of ground attack behavior from details of situation; can be reused in different mission types (CAS, SEAD) using different parameters, rather than developing specialized behaviors for each.



Coordinates with:
* Wingman
* AWACS
* Forward Air Controller

Interacts with:
* Situation Awareness
* Follow-Route
* Coordinate with team

Used by:
* Close Air Support
* TCT CAP
* Strike

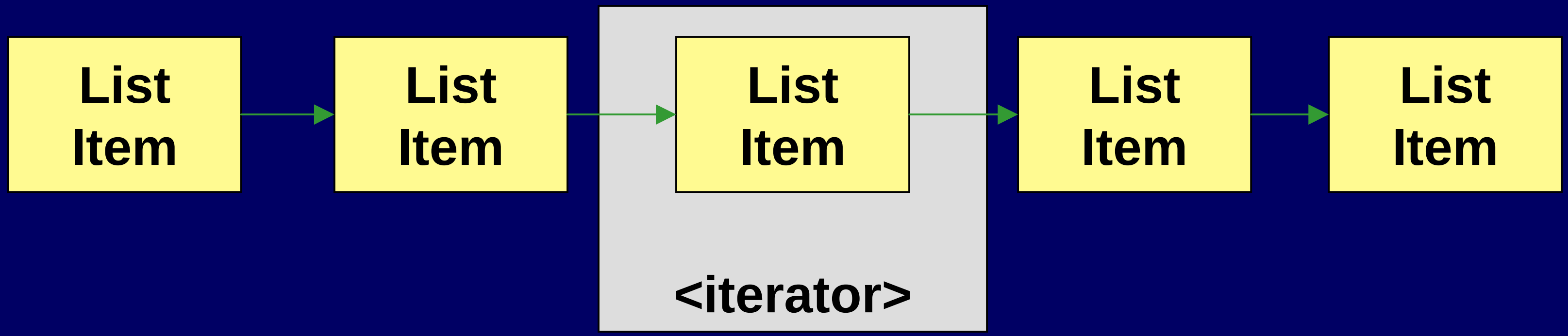
Variations:
* Source of Target Info
* Enemy unit type
* Attack Tactic
* Rules of engagement

How to use patterns?

- Indirectly: read the “BDP Catalog”, find pattern that fit your problem, and use the design solution to build your own implementation
- (more) Directly: Use generative programming techniques to create templates for generating solution instances
  - Tcl -> Soar

Simple example: Iterator Pattern

- Purpose: *Provide a way to access the elements of an aggregate sequentially without exposing its underlying representation* (GOF)



- Example: processing elements in a list

Possible Iterator Variations

- Sequential order pre-determined (list)
- Sequential order not pre-determined, but not important (unordered set)
- Sequential order determined as part of processing (search)
- ...

- Examples in HBR:
  - route following
  - iterating over targets
  - dealing with messages in priority order
  - etc.

Iterator Pattern Template

- Use iterator object to keep track of current element
- Create a new context in which to iterate
  - Explicit operator for iterating to encapsulate process from unrelated operators (template)
  - Proposal based on matching against existing iterator
  - Retraction based on iterator absence
  - Use other sub-operators for processing on current element
- User responsible for creating iterator instance (template)
- Auto-generate iterator destructor (template)
- User responsible for either telling iterator how to find the next element as part of instantiation (if well defined), or actually determining the next element as part of processing

Template continued...

- Create the iterator structure

(as production RHS):

```
iterator-constructor \
    location iterator-name \
    first-element
```

- Base iterator template generator:

```
iterator-template \
    production-base-name \
    opname lhstest rhsset \
    iterator-name {next-test NULL}
```

➔ Generates 3-4 productions to deal with iteration over specific structure

External processing of element...

Current & future work

- Refine/extend/define notation for behavior design patterns
  - Define “interacts with” subclasses/relationships
    - Hierarchical
    - Compositional
    - Aspectual
  - Evaluate/extend pattern definition
  - Create additional diagrammatic views
    - Task priority (potential interruptions)
    - Communication protocols
    - Tension: succinctness <--> behavior complexity

Current & future work

- Demonstrate (re-)use of patterns in new behavior systems
- Demonstrate reuse of patterns for other platforms
  - Long-term goal: Platform independent catalogue of BDPs
- Catalog behavior design patterns
- Refine evaluation metrics
  - What constitutes a “good” pattern?
    - Chase-target pattern: psychological relationship between follow leader and engaging enemy?
  - What are good/best/most effective ways of cataloging/ communicating patterns
- From description to programmability
  - Improving encapsulation & interfaces for cognitive archs
  - Template support

Summary

- Demonstrated "behavior design patterns" are evident in existing HBR systems
  - Analysis/review of TAS made many implicit design patterns evident
- Proposed strawman for behavior level BDPs
  - Computational abstraction: cognitive architecture
  - Multiple diagrammatic views to convey pattern succinctly
  - Formal ontology of interaction types to express complex relationships (hierarchical, aspectual, compositional, etc.) between interacting patterns