Aspects and Soar: A Behavior Development Model

Jacob Crossman
jcrossman@soartech.com

Motivation: Why is Soar Useful?

- Soar Systems are often complex
  - Often require multiple processes
  - Are built of hundreds/thousands of complex conditions
  - Often intended to solve hard/complex problems
  - Often involve issues with quasi-realtime performance and reactivity
  - Often have performance/memory issues (that might be resolvable, but take some thinking and analysis)
- Because of this complexity I often ask the question: “What is Soar buying me?”
  - Could I just write this same program in Java (or pick your favorite language/framework/system)?
  - This question assumes that these other systems are easier to use, more robust, and have better documentation/training materials (which is true)
- General Answers:
  - Soar allows variety of problem solving methods
  - Soar is reactive to changes in the environment
  - Soar has X (e.g. truth maintenance, impasses, etc)
- But what is (or are) the core development patterns that leverage the capabilities of Soar and…
  - What are the characteristics of the problems it solves and…
  - What are the architectural features that lead to these capabilities?

What About Procedures?

- We know Soar gives us powerful pattern matching, reactivity (via the RETE), and runtime choice mechanism (via operators),…
- … but what about procedures?
- A core element of any *behaving* system (including a Soar model) is its procedures
- We know that coding procedures is hard (or replace with "tedious" or "error prone") in Soar
  - Primary Cause: production system model (there are no procedural constructs – though operators and tricky production logic can provide the function)
  - Consider counting problem
- Is there a way to think about/model procedures that fits better with Soar's strengths?

Aspect Oriented Programming (AOP)

- New approach to problem decomposition and program structure
  - Developed at Xerox PARC by a team led by Gregor Kiczales
- **Key Observation**: There are common problems that are impossible to encapsulate using OO techniques.
  - Typical examples: logging, security, error handling, observer pattern
- These problems are referred to as *cross cutting* problems because they cut across the standard problem decomposition technique (i.e. the object infrastructure)
- Can we find ways to encapsulate the structure and logic associated with crosscutting features? Yes – aspect oriented programming

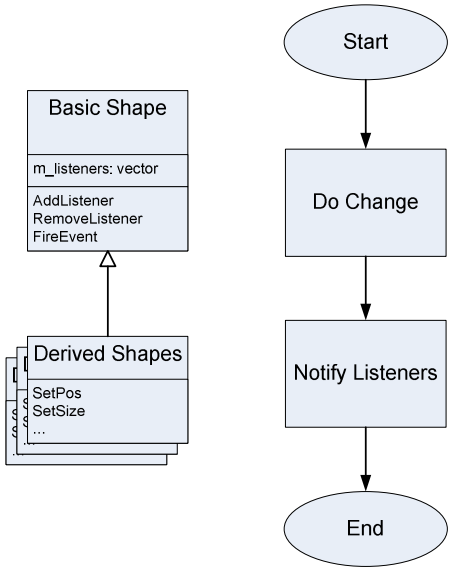
Key Elements

- Model of a program that defines nodes of execution explicitly and formally (*join point model*)
- Identification of specific places to insert or change behavior and data structures (*pointcuts*)
  - Requires pattern matching (often regular expressions)
  - Requires ability to introspect code (especially classes and functions)
- Insertion of code and data structures at appropriate places (*advice*)
  - Advice: code to slip into the function execution stream
  - + stuff for inserting additional data into objects
- Ability to encapsulate these features into modules (*aspects*)
- Typically, aspects are integrated into the core object code at compile time using an *aspect weaver*

Soar Technology
Thinking *inside* the box.

Examples

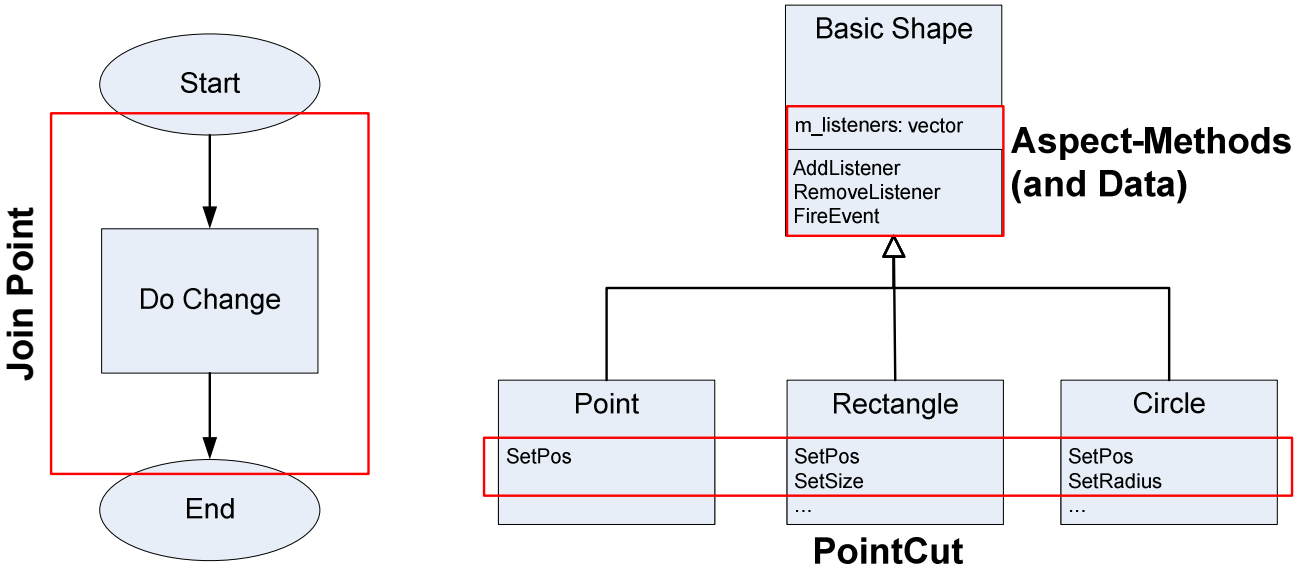
- Consider the observer pattern for updating a screen when a shape changes




```
public SetPos(…) {
    // Do position setting stuff
    x = posx
    y = posy
    // Update listeners
    FireEvent(SHAPE_UPDATED)
}
```

- Many of the most common crosscutting features are embedded in languages or the OS
  - Memory management
  - Task management
  - Function call stack management

Simplified Aspect for Observer



Advice

```
after() returning: shapeModify {
  // Update listeners
  FireEvent(SHAPE_UPDATED)
}
```

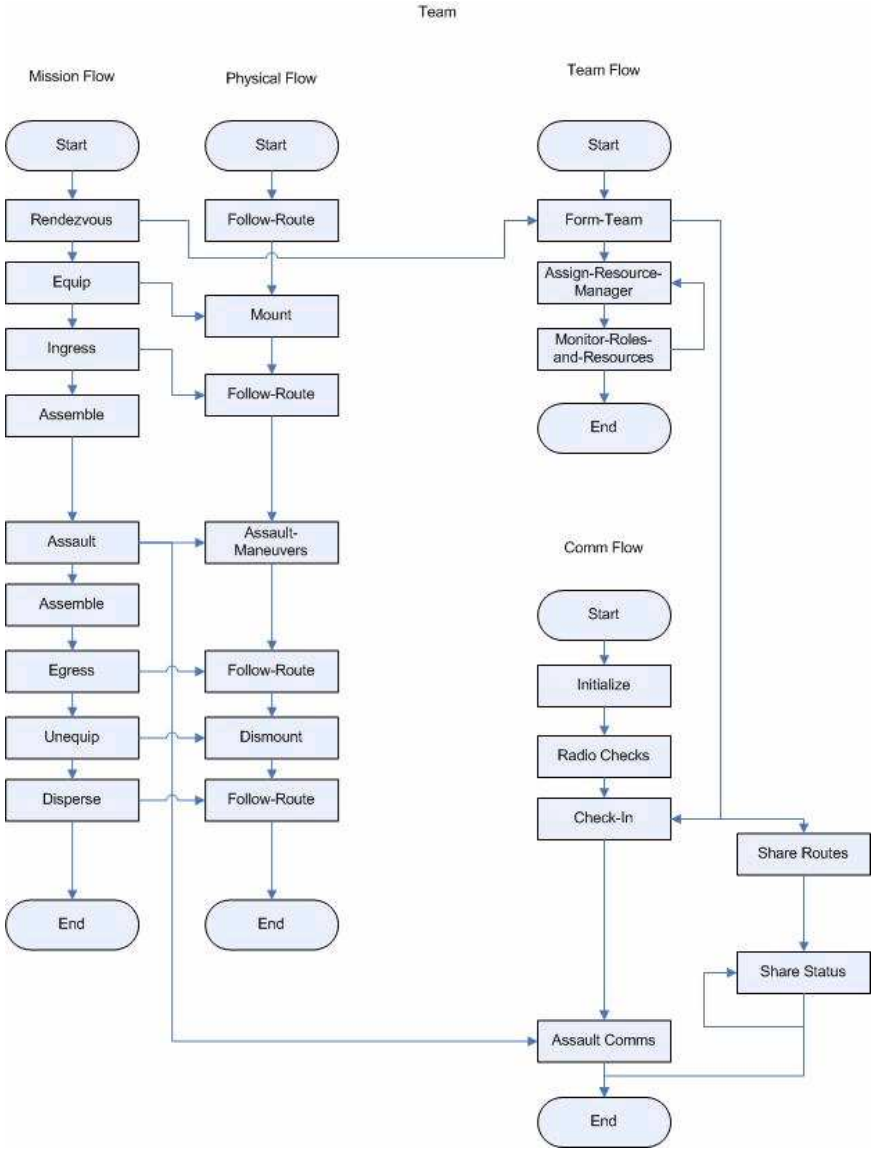
- The aspect combines the pointcuts and advice (as well as some other related elements) forming modular solutions to crosscutting concerns

“Aha” Moment

- Pattern matching, adapting behavior, crosscutting augmentation of process and data structure, ability to work across functional units,…. HMMM
- … Sounds like Soar
- Except Soar programs do this all the time, at RUNTIME!
- Is this the basis of a programming pattern in Soar?
- Is this this a useful way to understand how to leverage Soar’s unique capabilities?

Randy's Observation

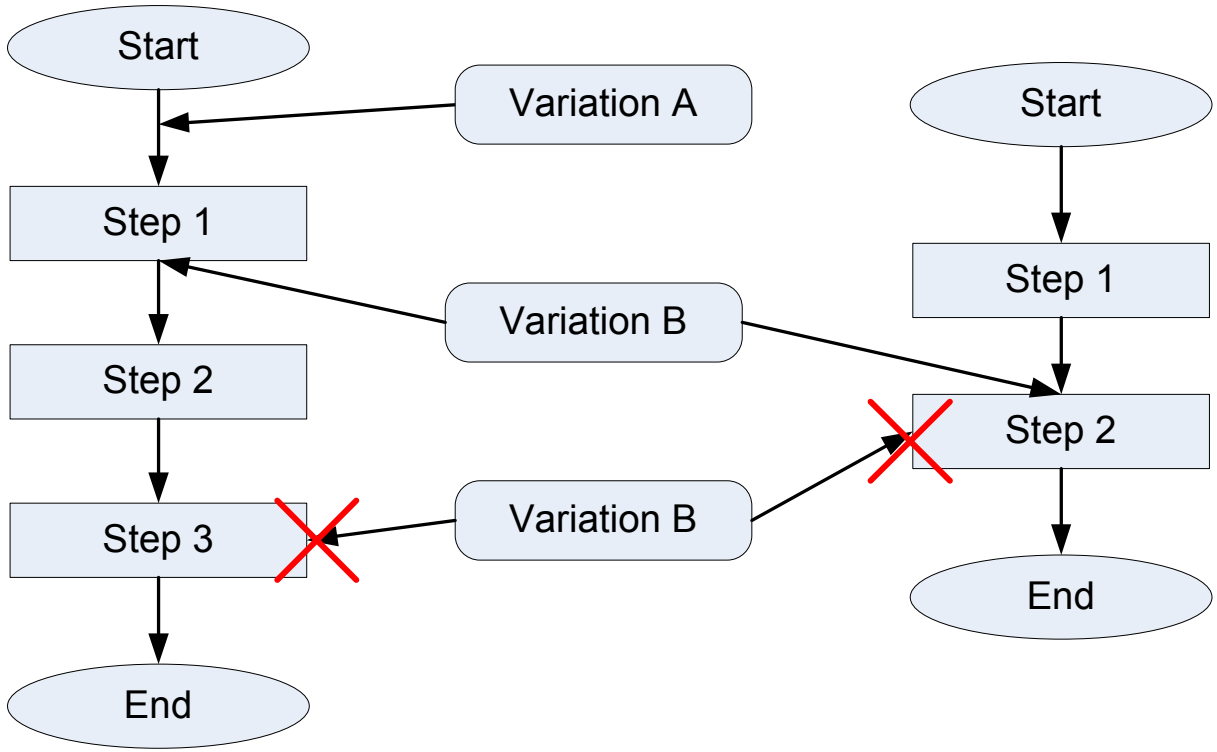
- For "expert behavior models"
  - SMEs often provide flow charts of behavior
  - However, these flow charts are just a template for real behavior
  - Detailed analysis and actual execution in the target environment are required to understand how these behaviors vary from the template.



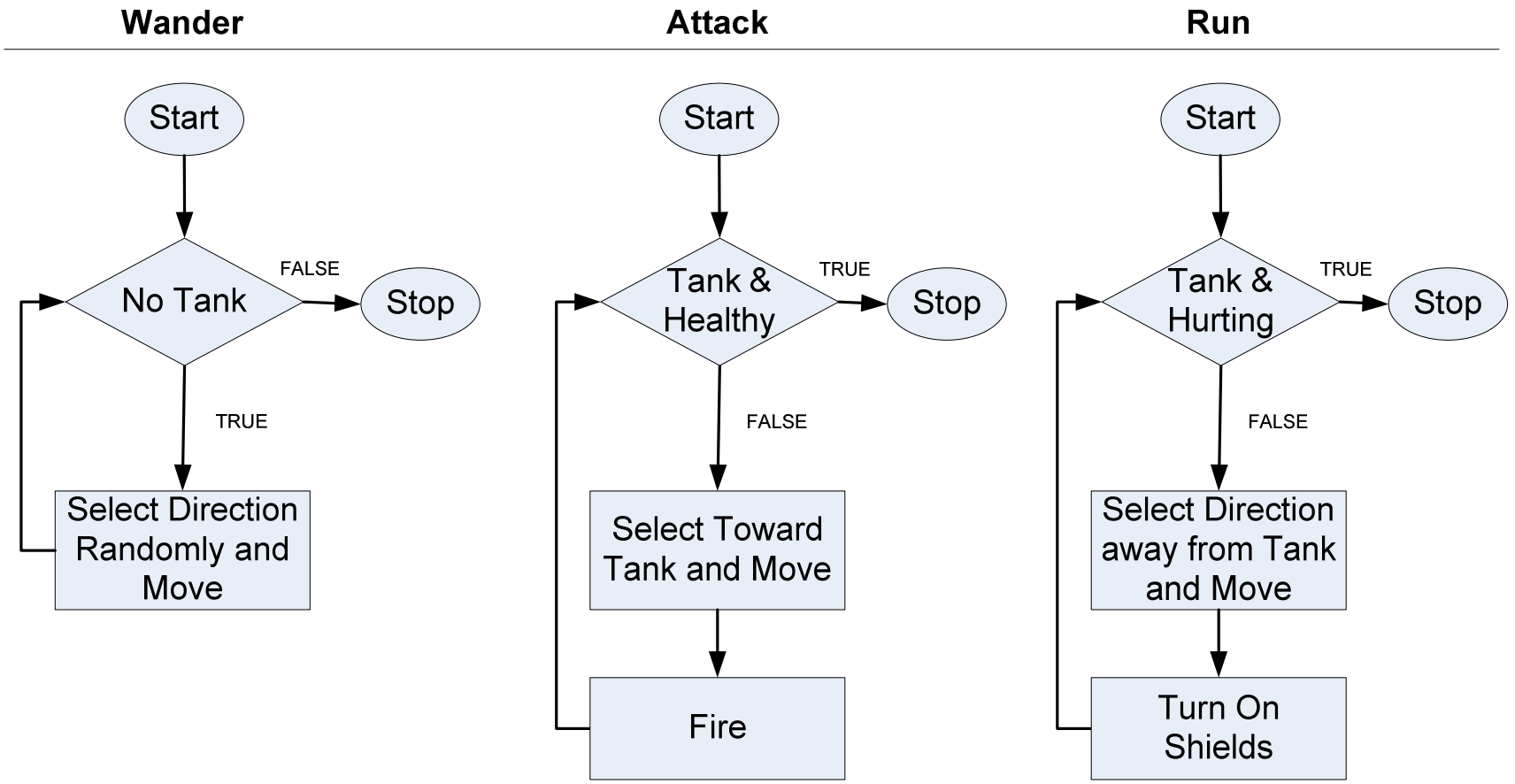
Soar Technology
Thinking *inside* the box.

Core Idea

- Can we think of Soar behavior models as having several core algorithms (e.g. the “doctrinal behavior”) augmented with variations on this behavior (possibly crosscutting) based on context?
- Can an aspect oriented approach tell us something about how this can work in a consistent, robust manner?

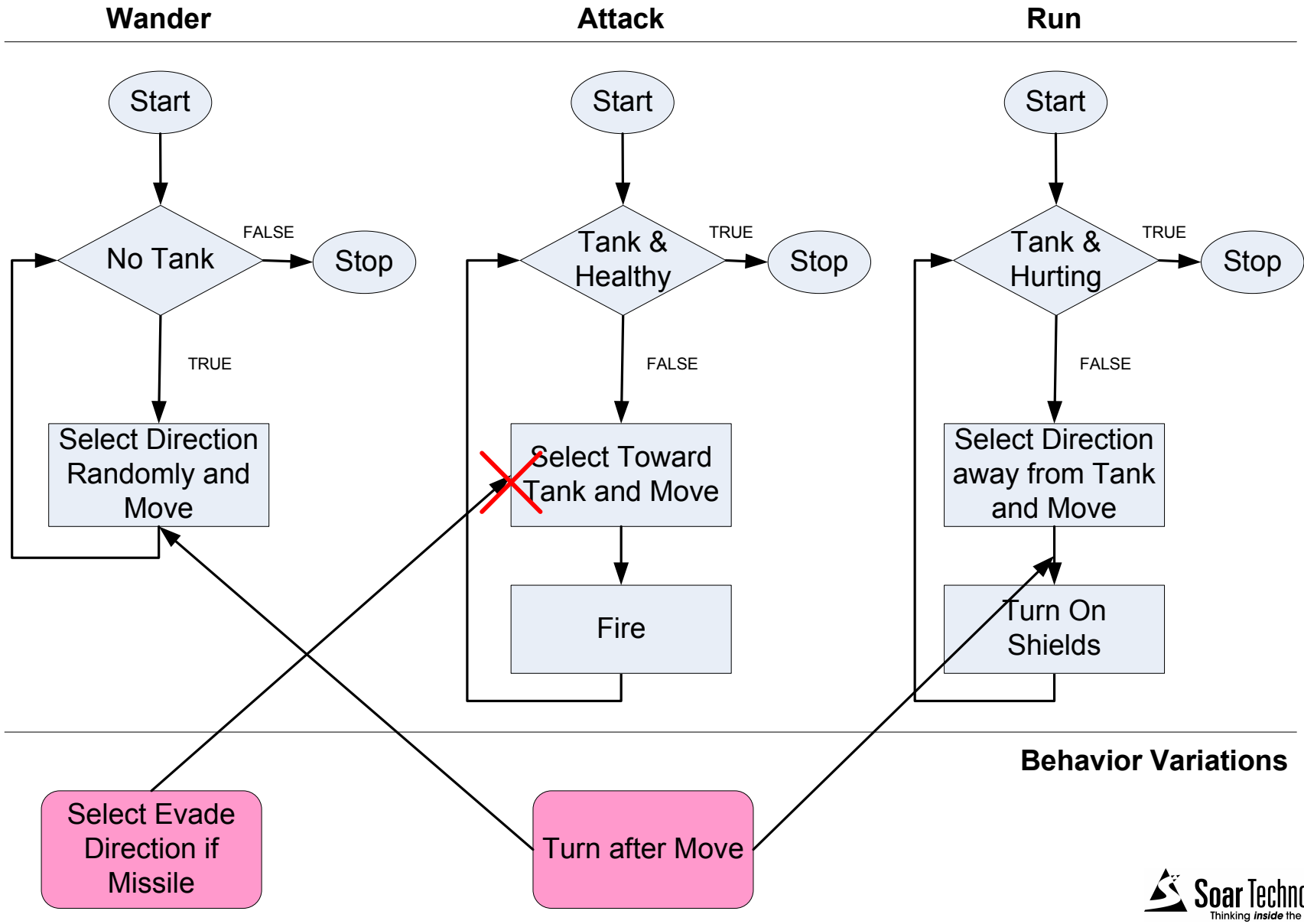


Example: Tank Soar (Simplified)



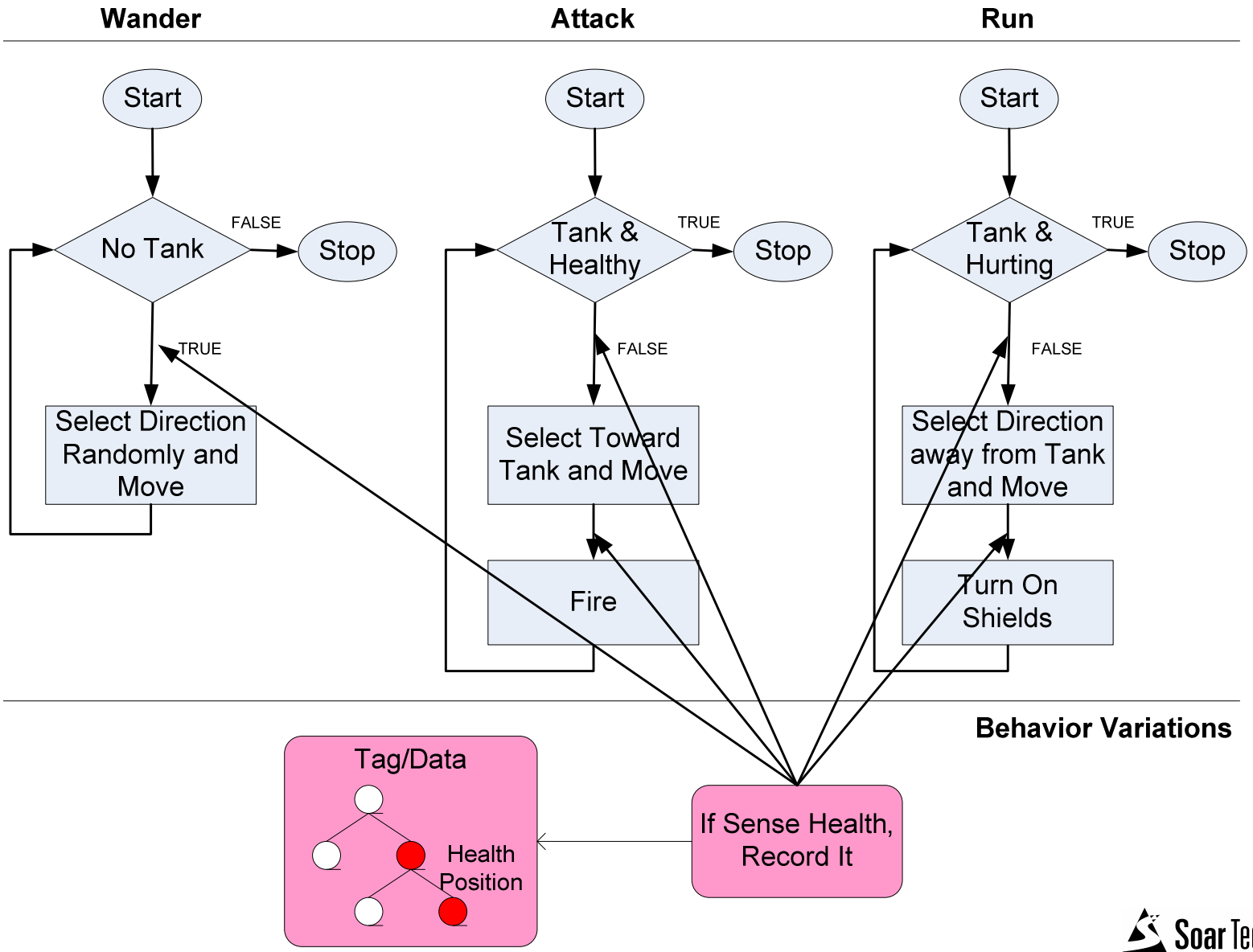
- Core Algorithm Template (Attack, Wander, Run)

Behavior Insertion and Replacement



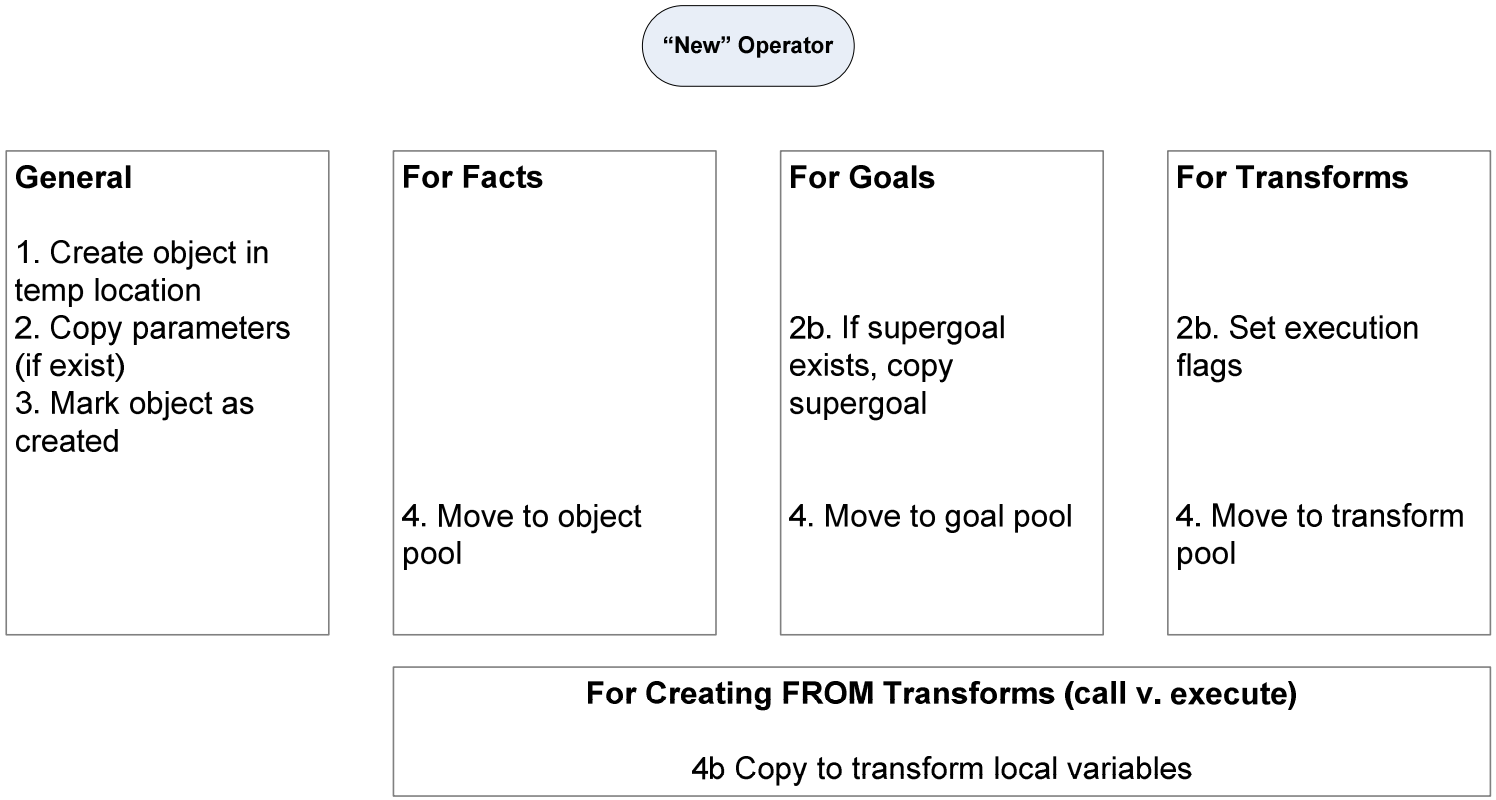
Soar Technology
Thinking *inside* the box.

Full Crosscutting and Tagging



Soar Technology
Thinking *inside* the box.

Example from HLSR (within operator)



- In OO, this would be 3 methods, plus logic embedded in a separate context to handle 4b
- Here, we've actually *reduced the need* for multiple algorithms by packaging variations (one is crosscutting)

Key Elements of AOP Mapped to Soar
(enabling architecture components)

- Join Point Model: **operators**
  - Internal – within an operator we have the proposal and apply phases
  - External – the operator selection and preference process
  - NOTE: Even the sometimes maligned “o-support via operators” means that a developer can depend on permanent actions *always using the same mechanism* (when theory is followed).
- Pointcuts: **symbolic patterns**
  - Provides a more general and robust way (v. current AOP approaches) to identifying appropriate point-cuts
- Advice: **operators and productions**
  - Productions: just create productions that fire over the appropriate set of activities
  - Operators: requires consistent use of preferences (e.g. “<” could give you “before advice”)
- Aspects: requires an HLL (but can be approximated using files)
- Aspect Weaver: The **Soar decision cycle** and preference mechanisms – but this happens all at *runtime*

Caveats (Coal)

- It is *possible* to implement powerful runtime aspect oriented behavior using Soar (and this is done sometimes), but…
- There is no support for modularity in Soar: best use the file system to maximum effect
- Requires consistent use of Soar features/capabilities such as operators and preferences
- Consistent implementations would rely heavily on *convention* to keep the model maintainable and robust
- Solution: Implementation of this capability in an HLL
  - HLSR is implementing *some* of these features this year

Recommendation (Nuggets)

- Soar is especially well suited to implementing aspect oriented behavior, particularly
  - contextual behavior insertion
  - crosscutting behavior
- When building a Soar model it is worthwhile to
  - Analyze the problem to determine if it requires significant contextual variation and/or contains crosscutting elements
    - If not, can you write it as a standard procedural algorithm? If you can, do it that way (maybe tying results into Soar)
    - If so, consider decomposing the algorithm along lines of core algorithms, variations, and crosscutting behavior as shown
- This is not necessarily the *only* pattern for which Soar is well suited, but it appears to be a model for which Soar provides significant advantages over traditional development approaches
- Indicating problem/solution characteristics:
  - You are building a model with explicitly represented knowledge (i.e. you are not building a general purpose reasoner)
  - Your domain is the real world or a complex simulated environments
  - Your solution is required to interact in near realtime to changes

References

- AOP in General: http://aosd.net/
- AspectJ: http://www.eclipse.org/aspectj/
- Google Video on AOP: http://video.google.com/videoplay?docid=8566923311315412414&q=Kiczales
- Kiczales: http://www.cs.ubc.ca/~gregor/