SASSY: Self-Architecting Software SYstems

- A Framework for Utility-Based Service Oriented Design in SASSY

By Daniel Menasce, John Ewing, Hassan Gomaa, Sam Malek and Joao Sousa

- QoS Architectural Patterns for Self-Architecting Software Systems

By Daniel Menasce, Joao Sousa, Sam Malek, Hassan Gomaa

Presented by: Kaveh Razavi

Motivation

- Let domain experts, as opposed to software engineers, express system requirements in an easily understood high level visual language
- Allow for a uniform approach in SOA to
  - Automated composition
  - Adaptation
  - Evolution of software systems
- In face of emergent runtime problems
  - Automate the process of creating a new architecture
  - Elevate reaction on the scale of seconds, as opposed to hours or days it takes humans

Overview of SASSY

SOA Review

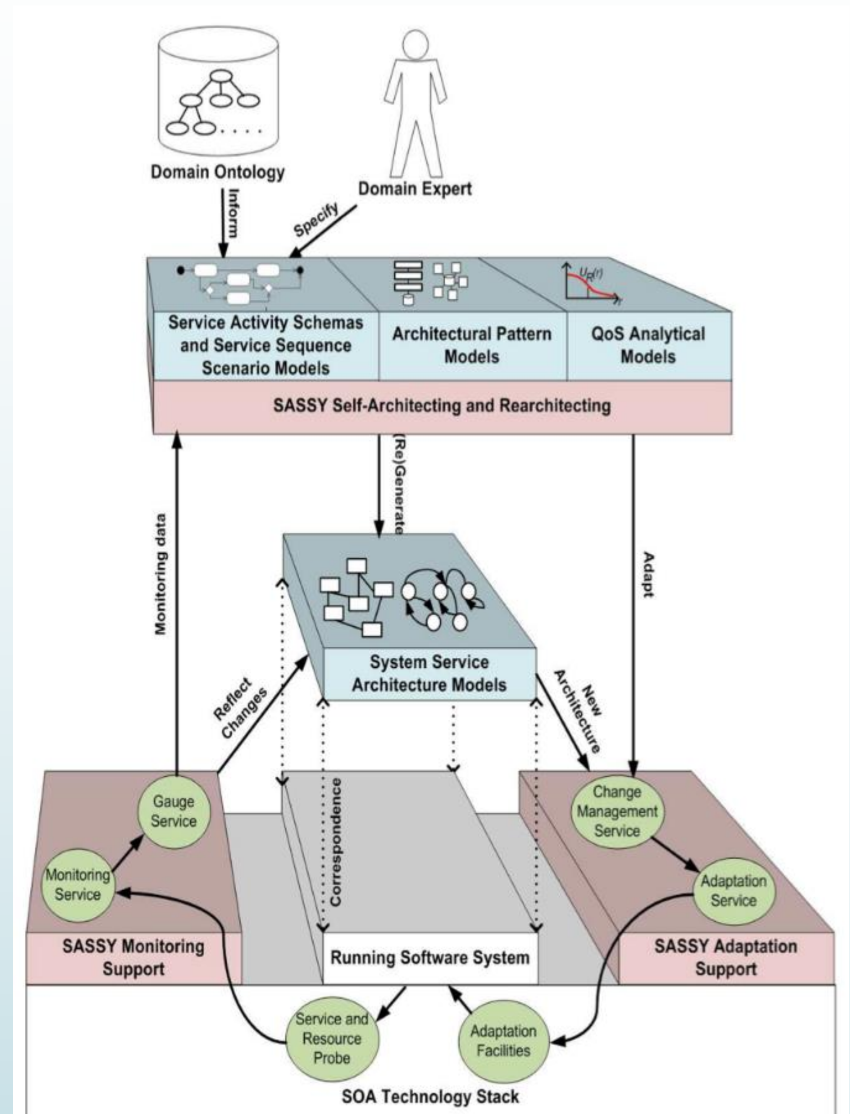
- SOA: A software architecture that consists of multiple distributed autonomous services
  - The same service type can be offered by different service providers (SPs)
  - Services and their providers can be discovered at runtime
  - New SPs may be brought into existence and existing SPs may fail or stop operating at any time
  - Different functionally equivalent SPs may exhibit different QoS levels at different costs

SASSY

- A model-driven framework for run-time self-architecting and re-architecting of distributed software systems
- Automatically generates a base architecture corresponding to the requirements
- Generates an architecture derived from the base architecture that optimizes a utility function for the entire system by:
  - Optimal selection of service providers for service types (done before)
  - Automatically generates a number of candidate architectural patterns to replace each service (novel idea)
- Once the architecture is deployed, there may be a need to re-architect due to changes in the environment (e.g., failures of service providers or changes in their QoS characteristics)

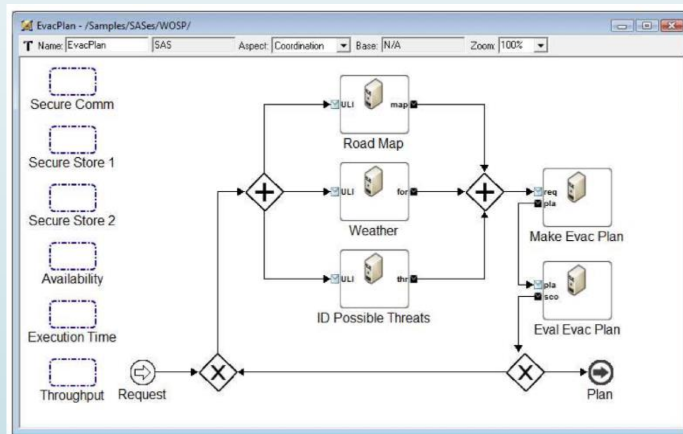
SASSY

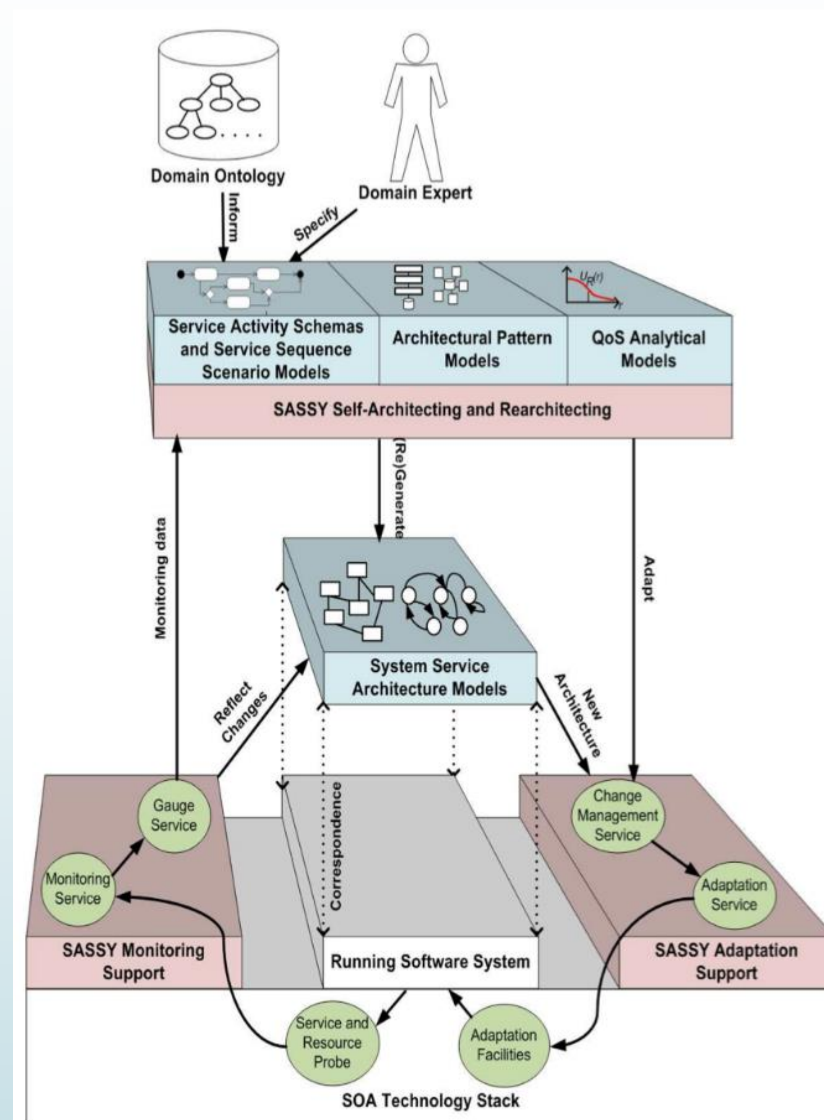
- Run-time Models
  - Service Activity Schemas (SASs) with their corresponding Service Sequence Scenarios (SSSs)
  - System Service Architecture (SSA)
  - QoS Architectural Patterns
  - QoS Analytical Models
- SASSY Logic
  - Monitoring Support
  - Adaptation Support
  - Self-Architecting and Re-architecting
- External Component
  - Running Software System



SAS: Service Activity Schema

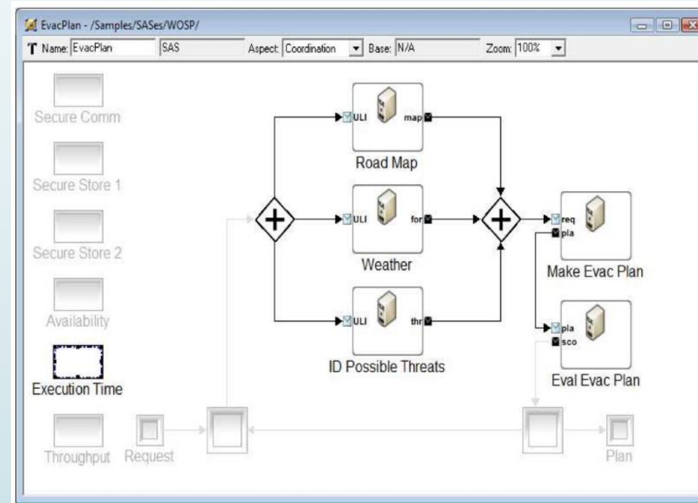
- Describes the features and logic of the application and
- Is written by domain experts in a graphical notation similar to Business Process Modeling Notation (BPMN)
- Modeling constructs like activities, service types and domain entities are defined in a domain ontology

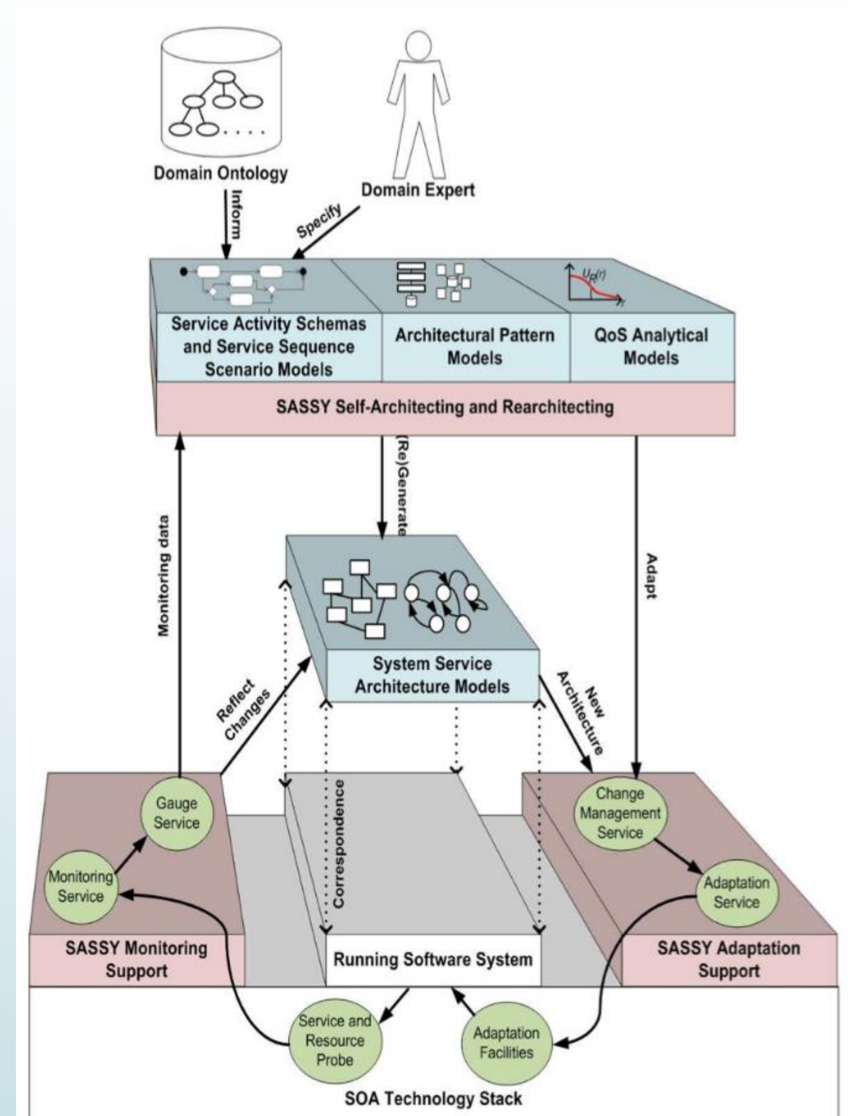




SSS: Service Sequence Scenario

- Sub-graphs of SAS: A set of one or more service types connected by links in an SAS
- The purpose: specifying end-to-end goals along the paths in the form of utility functions
- Each SSS is associated to one of the QoS metrics
- Two or more SSSs may have the same structure as long as they have different metrics associated with them





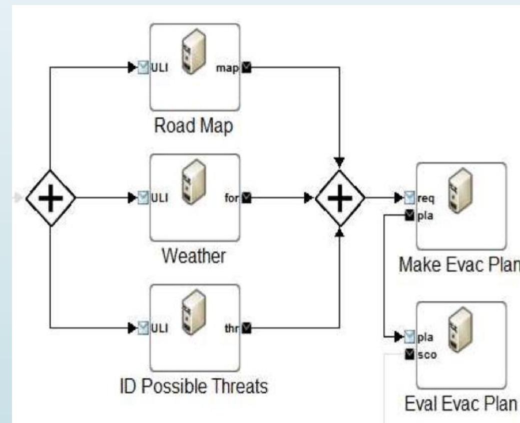
Utility Function

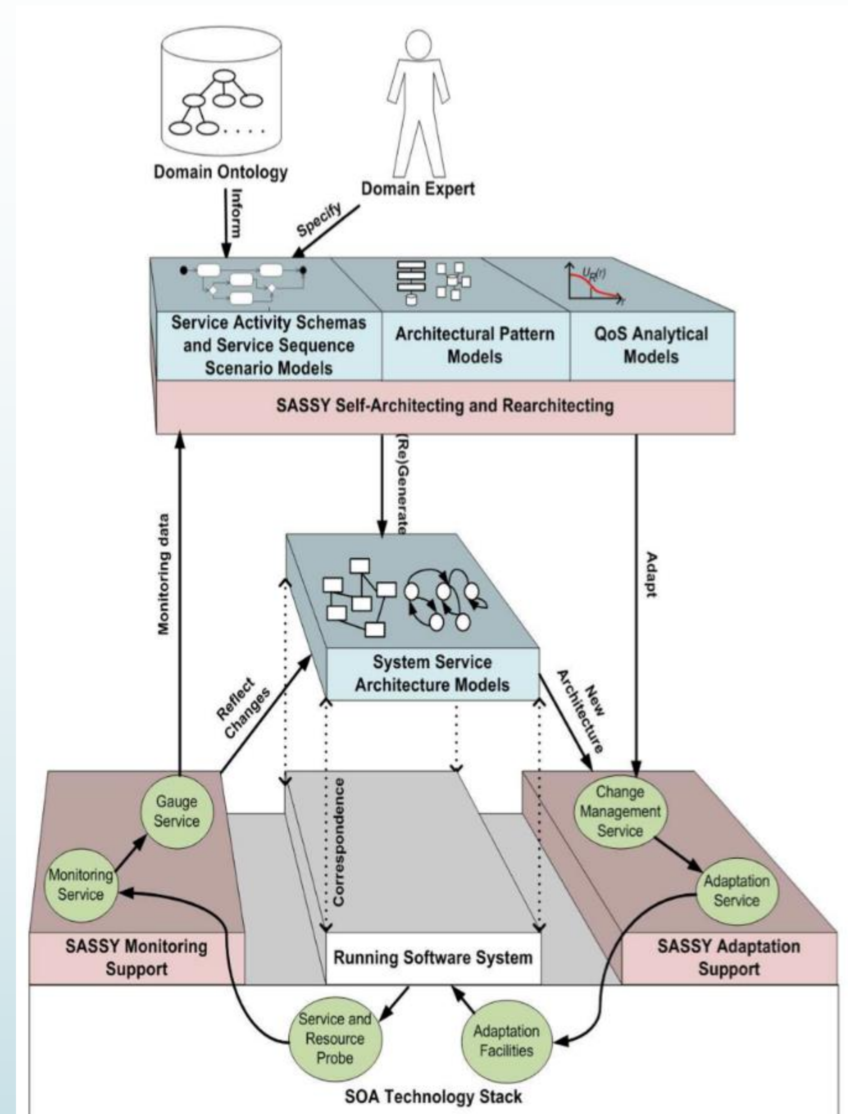
$$U_{\text{Availability}}(a) = \begin{cases} 0 & a < 0.9 \\ 0.5 & 0.9 \leq a < 0.95 \\ 1 & 0.95 \leq a < 1 \end{cases}$$

- Are used in economics and autonomic computing to assign a value to the usefulness of a system based on its attributes.
- Utility functions are established by the domain experts in consultation with the stakeholders to express the quality of a given architecture
- Can be:
  - Univariate: A function like the ones associated with each SSS
  - Multivariate: A function of several QoS metrics like the global utility function for the entire SAS
    - The overall utility of a system is defined as a composition of the individual utilities according to their relative importance.

Analytical Model

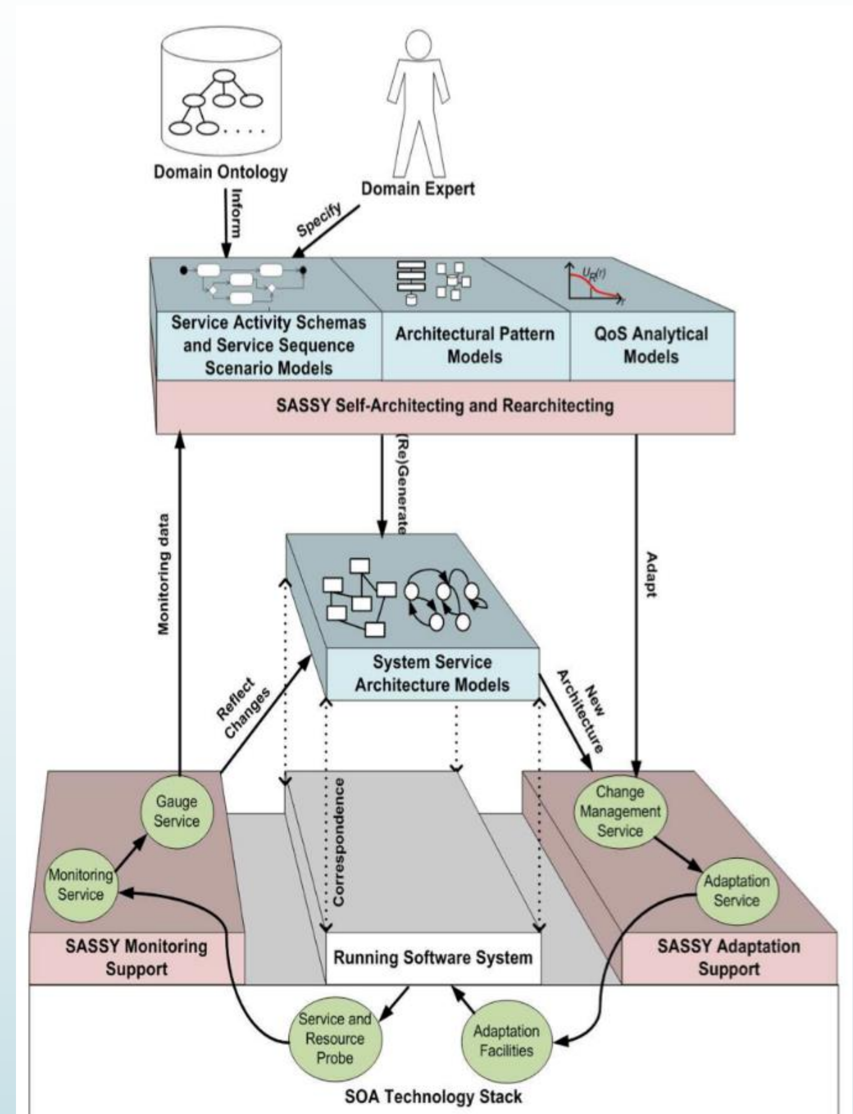
- Are used to derive and express the end-to-end QoS attributes, and hence the utility of the system
- Example:
  - $a = a_{Road\ Map} \times a_{Weather} \times a_{ID\ Possible\ Threats} \times a_{Make\text{-}Evac\text{-}Plan} \times a_{Eval\text{-}Evac\text{-}Plan}$
  - $e = \max\{e_{Road\ Map}, e_{Weather}, e_{ID\ Possible\ Threats}\} + e_{Make\text{-}Evac\text{-}Plan} + e_{Eval\text{-}Evac\text{-}Plan}$





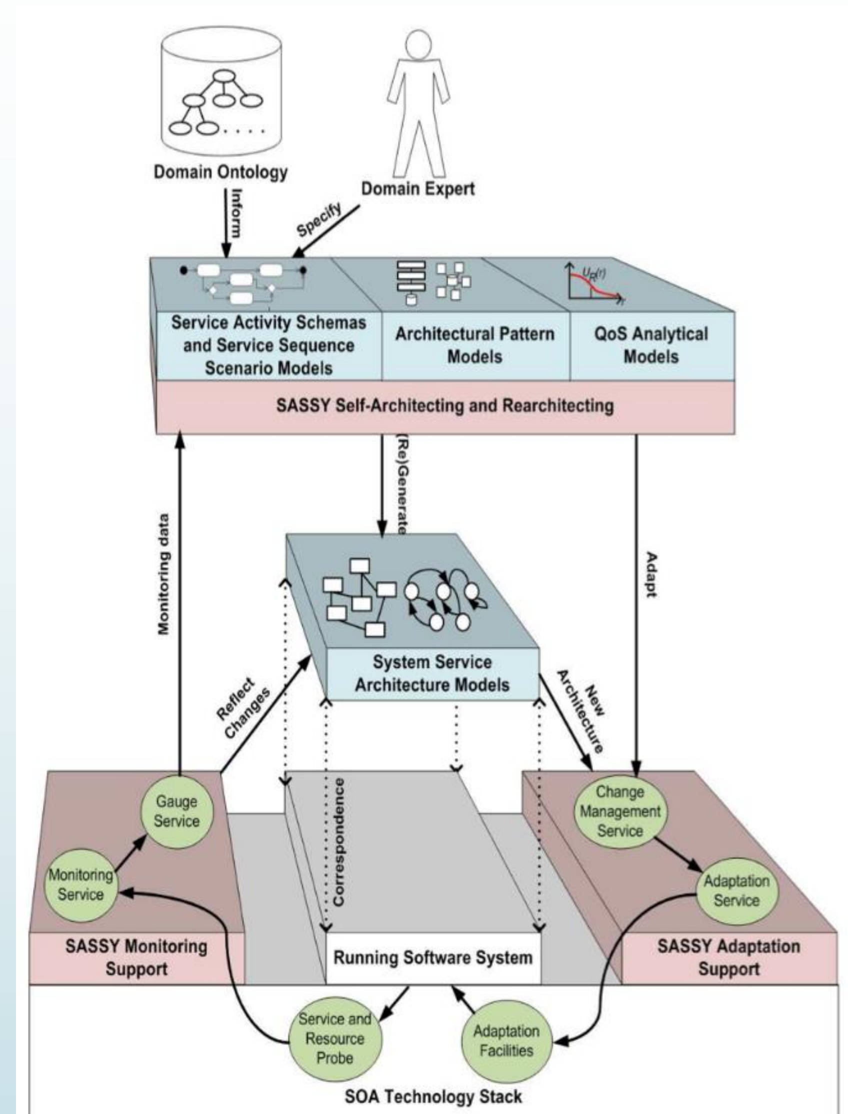
Monitoring Support

- Probes the SOA implementation
- Generates triggers that cause self-adaptation
  - When service providers fail or are unable to meet their QoS goals



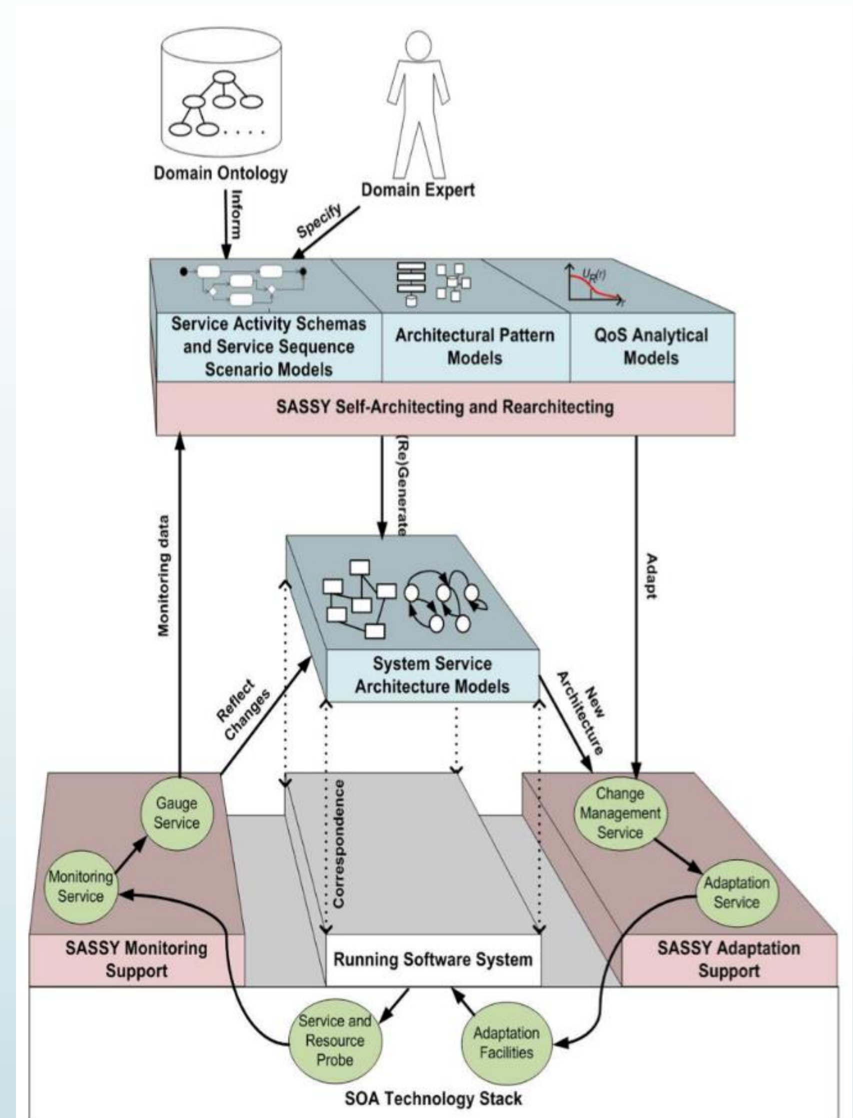
Adaptation Support

- Uses the widely accepted 3-layer architecture model of self-management:
  - Goal Management Layer: Planning for change – often human assisted
  - Change Management Layer: Execute the change in response to changes in state (environment) reported from lower layer or in response to goal changes from above
  - Component Control Layer: executing architecture, actually implements the run-time adaptation



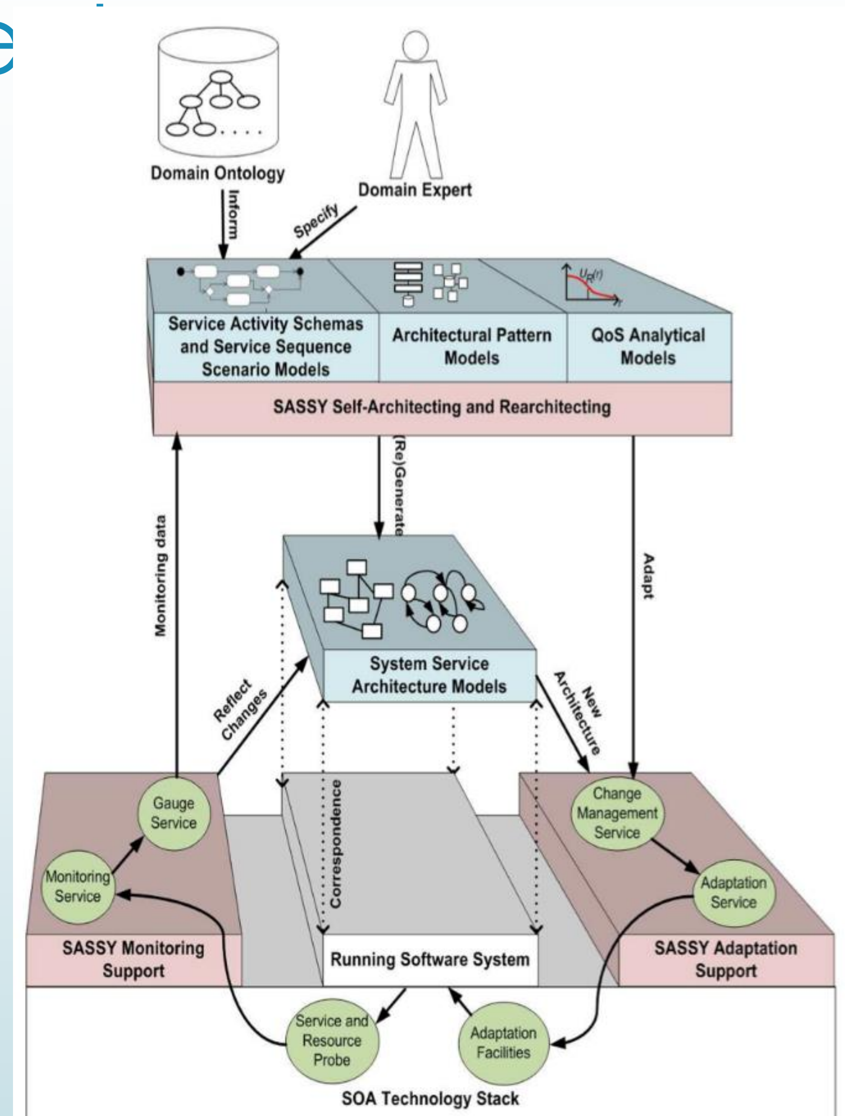
SSA: System Service Architecture

- Is an accurate representation of the running software system
  - Provides mapping between each service instance and the concrete service provider
  - Is intended for use at run-time, unlike traditional models used during design
- Base SSA is automatically generated:
  - Associates one component to each service type
  - Creates one component to represent the logic of a coordinator that orchestrates communication between service types
- SASSY automatically generates SSA's behavioral models, executable logic of service coordination in SOA, based on the requirements specified by domain expert in SAS



SASSY Self-architecting & Re-architecting Compone

- Automatically generates a near-optimal SSA, and maintains that optimality in the face of changes
  - Detected by the Monitoring Support—self-healing and adaptation
  - Made by users in the SAS and SSS—system evolution.
- Focuses on a set of SSS with greater room for improving their contribution to the overall utility.
- Generates variations to the system architecture by replacing each service along an SSS with candidate architectural patterns that are functionally equivalent but improve some aspect of QoS.



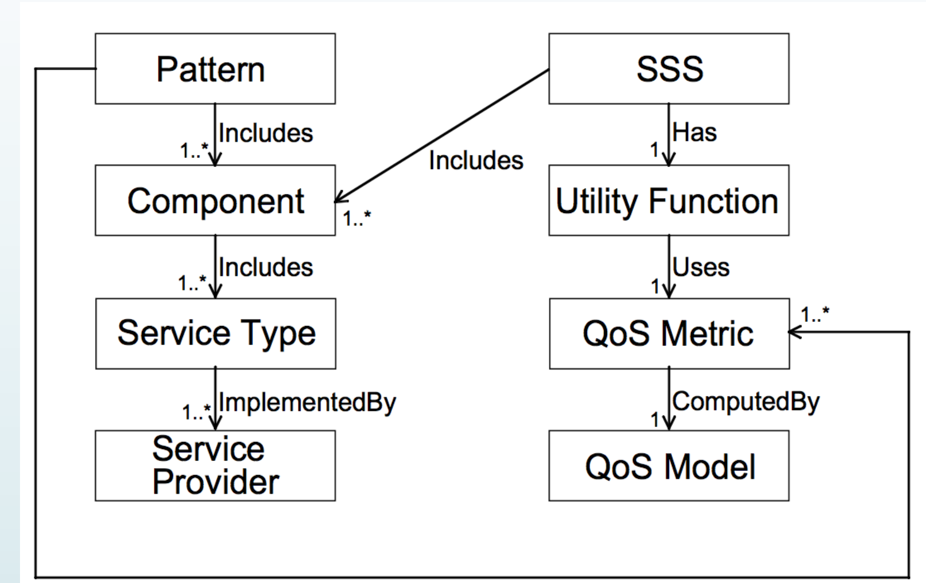
SASSY QoS Architectural Patterns

QoS Architectural Patterns

- choosing a pattern that promotes certain aspects of quality normally has a negative effect on some other aspects of quality.
- The task of an architect is to make tradeoffs that reflect the priorities of stakeholders.
  - This task is especially complex for large systems
- SASSY uses efficient and scalable search heuristics to identify the optimal patterns, making it possible to perform self-architecting both at system deployment and at run time for purposes of self-adaptation

QoS Architectural Patterns

- SASSY uses a library of architectural patterns to assist in the self-architecting process
- Each pattern consists of 1 or more components
- Each component may be associated with 1 or more service type
- Each service type is instantiated by 1 or more service provider
- Each pattern also includes 1 or more QoS metrics and their corresponding QoS model
- Each SSS has a single utility function
- That utility function is a function of a single QoS metric
- An analytic QoS is used to compute the value of that QoS metric



Basic Pattern

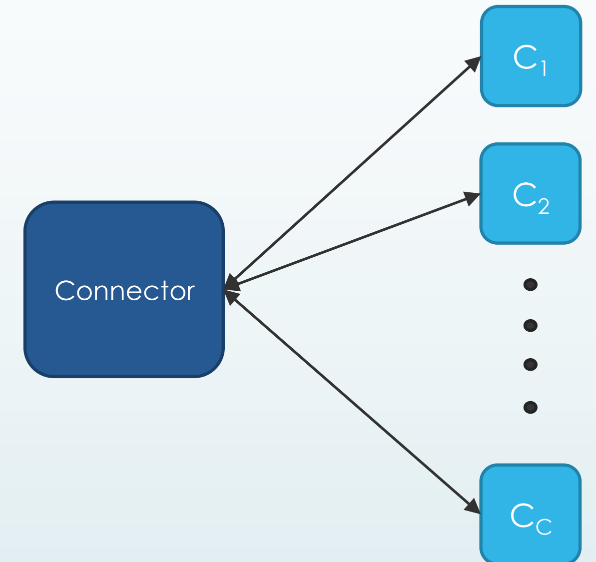
- The simplest possible pattern in terms of structure.
- It consists of a single component c and no connectors.
- Its behavior corresponds to asynchronous message-passing
- The availability of the basic pattern reflects the probability that it is available to receive the message
- Its execution time reflects the time it takes to act on the message received

$$a = \mathcal{M}_a(v_{a,c}) = v_{a,c}$$

$$e = \mathcal{M}_e(v_{e,c}) = v_{e,c}$$

Fault-Tolerant, First-to-Respond

- Consists of C components C1,⋯,C_C and a connector that receives requests and sends them in parallel to all C components
- It is assumed that components fail independently of one another
- All components process the request and send their replies to the connector, which replies to its requester as soon as the first component replies
- Promotes availability
- Redundant usage of resources negatively impacts scalability

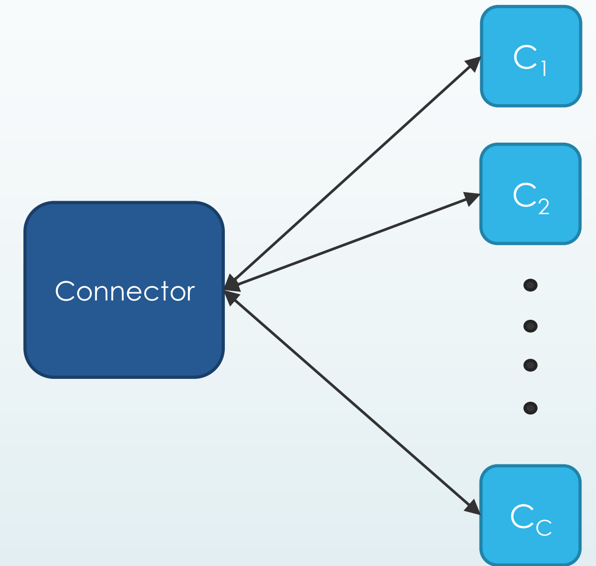


$$a = \mathcal{M}_a(v_{a,C_1}, \cdots, v_{a,C_C}) = 1 - \prod_{j=1}^{C}(1 - v_{a,C_j})$$

$$\begin{aligned} e \quad = \quad & \frac{1}{a}\sum_{\forall\ \vec{\epsilon}}\prod_{j=1}^{C}[1 - (\epsilon_j + (-1)^{\epsilon_j} v_{a,C_j})] \times \\ & \min\{\frac{v_{e,C_1}(1+\alpha)}{\epsilon_1 + \alpha}, \cdots, \frac{v_{e,C_C}(1+\alpha)}{\epsilon_C + \alpha}\} \end{aligned}$$

Fault-Tolerant, Two-Phase Commit

- Has the same structure
- Connector receives a request, sends it for processing to all components, waits for all to respond, and then sends a commit request to all of them
- Promotes fault-tolerance when information has to be maintained at more than one location
- Increased availability comes at the expense of reduced execution time

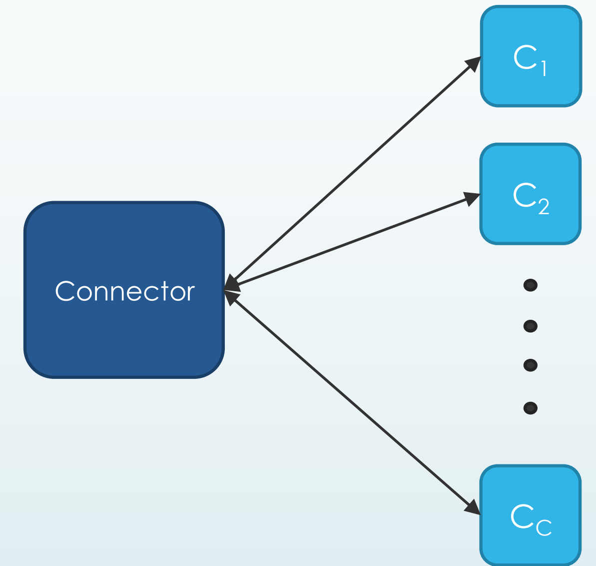


$$a = \mathcal{M}_a(v_{a,C_1}, \cdots, v_{a,C_C}) = \prod_{j=1}^{C} v_{a,C_j}$$

$$e = 2 \times \max\{v_{e,C_1}, \cdots, v_{e,C_C}\}$$

Load Balancing

- Has the same architectural structure as the two previous ones but with different behavior.
- Connector sends requests to one and only one of the components at a time. When it receives a reply, the connector issues a reply.
- Promotes scalability and availability
- Execution time is less pronounced: the expected time is a weighted average of the response times, in contrast to a guaranteed fastest available response time.

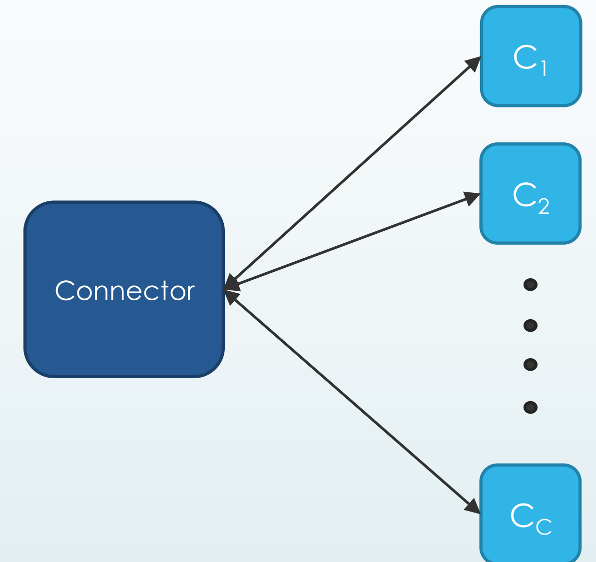


$$a = \mathcal{M}_a(v_{a,C_1}, \cdots, v_{a,C_C}) = \frac{1}{C}\sum_{j=1}^{C} v_{a,C_j}$$

$$e = \frac{1}{C}\sum_{j=1}^{C} v_{a,C_j} \times v_{e,C_j}$$

Parallel Invocation

- A connector receives a request and breaks it down into sub-requests that are sent in parallel to all components. Connector merges all replies from the components and replies to the original request.
- The connector and all components have to be available for the pattern to be available.
- Promotes the reduction of execution time given that the overall work can be broken down in smaller pieces to be executed in parallel.
- Reduced availability.



$$a = v_{a,\text{connector}} \prod_{j=1}^{C} v_{a,C_j}$$

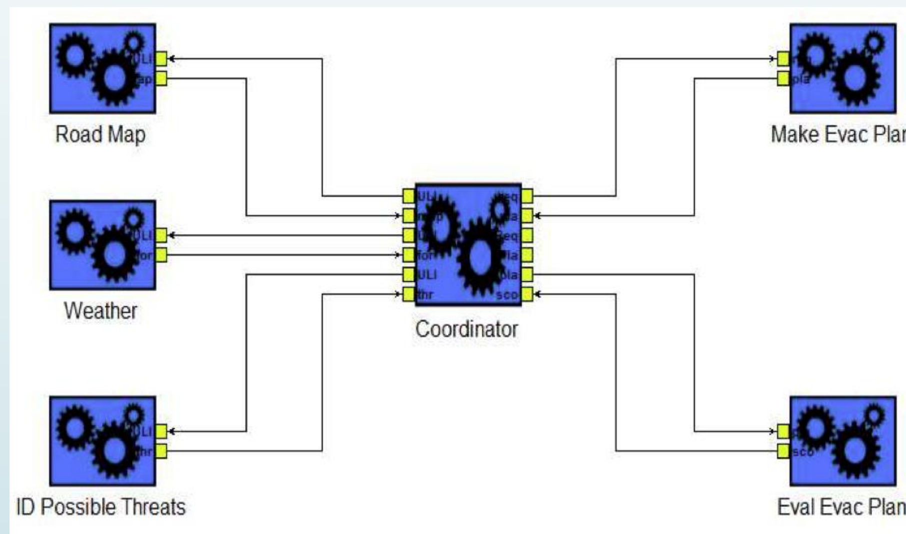
$$e = 2 \times \max\{v_{e,C_1}, \cdots, v_{e,C_C}\}$$

Composition of QoS Architectural Patterns

- A utility function, associated to a QoS metric, is assigned to each SSS. Then, all these utility functions are combined into a global utility function
- The end-to-end QoS metric along an SSS depends on the values of that metric for each component or pattern in the SSS
  - The end-to-end execution time is the sum of the execution times of basic components or composite components (i.e., patterns) along an SSS
  - The end-to-end availability is the product of the availabilities of basic components or composite components (i.e., patterns) along an SSS

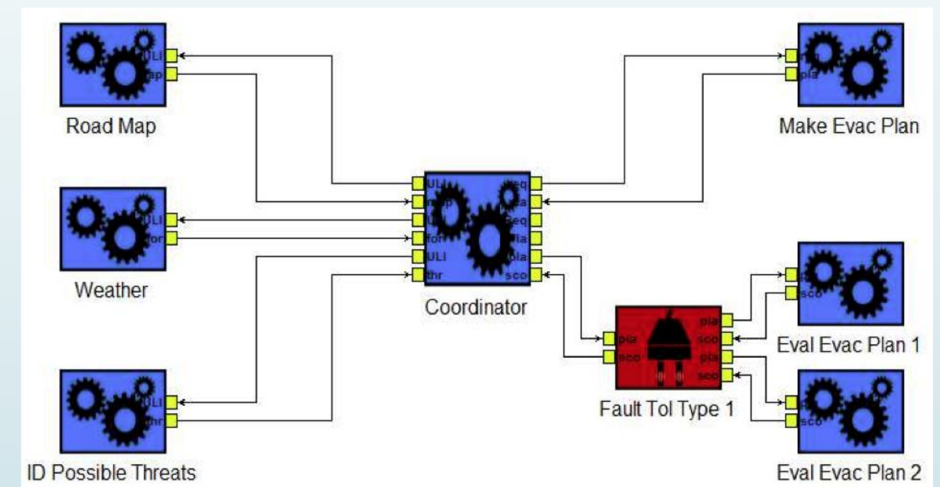
Architectural Patterns - Example

Base SSA



Replacement of a component with a fault tolerant component

$A = 1 - (1 - a1)(1 - a2)$



The Optimization Problem

Optimization Problem

- Finding an architecture (A*) and a set of service provider allocation (Z*) that implement service types in SAS in a way that optimizes the SAS utility function Ug
  - $(A^*, Z^*) = \text{argmax}(A, Z)\ U_g(A, Z)$
  - There could be a modified cost-constrained case in which there is a cost associated with each service provider for a certain QoS level
- If
  - p: average number of architectural patterns that can be used to replace any component
  - n: number of components in architecture
  - s: average number of service providers that can be used to implement each component
- Then
  - The number of different architectures is $O(p^n)$
  - The number of possible service provider selections for an architecture is $O(s^n)$
  - The size of the solution space for the optimization problem is $O((s \times n)^n)$
  - The problem is NP-hard
- SASSY uses a heuristic-based search technique

General Search Approach

```
1:  function GeneralSearch ()
2:  A_visited ← A_base; /* initialization */
3:  OptimalServiceProviderSelection (A_visited);
4:  U_g ← Utility (A_visited) /* utility computation */
5:  Searching ← TRUE
6:  while Searching do
7:     A_opt ← A_visited
8:     N ← GenerateNeighborhood (A_visited, k)
9:     for all A_i ∈ N do
10:       OptimalServiceProviderSelection (A_i)
11:    end for
12:    A ← argmax_{A_i∈N} {Utility(A_i)}
13:    if Utility (A) > U_g then
14:       A_visited ← A
15:       OptimalServiceProviderSelection (A_visited);
16:       U_g ← Utility(A_visited) /* utility computation */
17:    else
18:       Searching ← FALSE
19:    end if
20: end while
21: end function
```

- Step 1. Start with architecture A_{base} and corresponding service selection Z_{base} of service providers for the service types of A_{base}.
- Step 2. Identify the SSSs with the lowest contribution towards overall utility. How many SSSs to consider is a parameter of the heuristic.
- Step 3. Find a neighborhood N of architectures derived from A_{base} by replacing QoS architectural patterns in A_{base} by other candidate QoS architectural patterns that improve the utility of the SSSs identified in Step 2.
- Step 4. Perform a near-optimal service provider allocation for each architecture in N. This is also an NP-complete problem for which SASSY uses a heuristic described
- Step 5. Compute the global utility Ug for each architecture in N and pick the architecture A_{opt} with the largest utility in N
- Step 6. If the utility of A_{opt} represents a"good enough" improvement over the previous value of the global utility, stop and return A_{opt}. Otherwise, make A_{base} equal to A_{opt} and go to step 2

Generate Neighborhood

```
1:  function GenerateNeighborhood (A_visited, k)
2:  N ← φ; /* initialize with empty neighborhood */
3:  S_k ← Set of SSSs with k lowest contribution to U_g
4:  for all s ∈ S_k do
5:     m ← s.QoSmetric
6:     C ← Set of components of s
7:     P ← Set of patterns that improve m
8:     for all c ∈ C do
9:        for all p ∈ P do
10:          N ← N ⋃ Replace(A_visited, c, p)
11:       end for
12:    end for
13: end for
14: return N
15: end function
```

- Unfiltered
  - Replaces every component of every SSS with an architectural pattern that improves the metric associated with the SSS
  - Large neighborhood
  - Large neighborhoods imply larger computational costs
- Filtered
  - Reduces the size of the neighborhood by concentrating on the SSSs that can provide better gains to the utility function
  - The smaller the neighborhood, the higher the likelihood that the search will be trapped in a local optimum

Case Study Results

Case Study I

Service Type	Service Provider	Execution Time (msec)	Availability
Building Locator	BL1	50	98%
	BL2	70	97%
	BL3	60	99%
Occupancy Awareness	OA1	120	99%
	OA2	150	95%
	OA3	100	98%

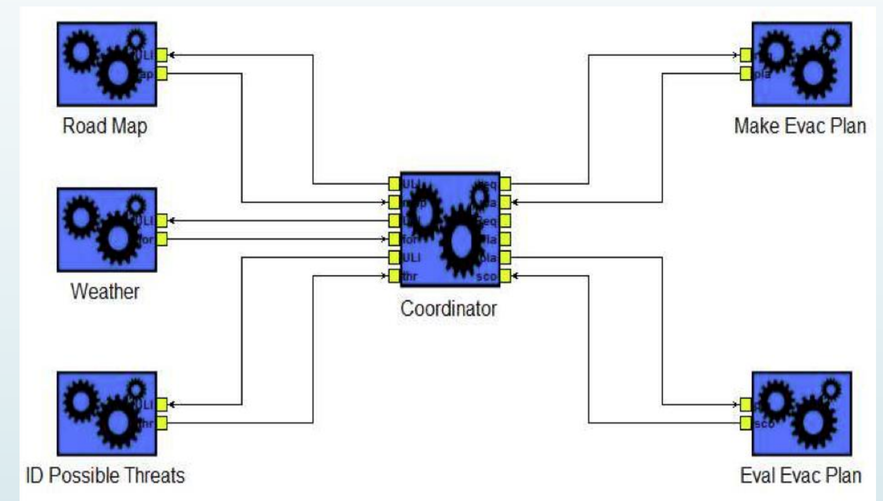
No	Building Locator	a	e	Occupancy Awareness	a	e	a_{SSS}	e_{SSS}	$U_a(SSS)$	$U_e(SSS)$
1	BC (BL1)	0.980	50.000	BC (OA1)	0.960	120.00	0.9408	170.00	0.5	0.99331
2	BC (BL1)	0.980	50.000	BC (OA2)	0.950	150.00	0.9310	200.00	0.5	0.00005
3	BC (BL1)	0.980	50.000	FFT (OA1,OA2)	0.998	121.14	0.9780	171.14	1.0	0.98821
4	BC (BL1)	0.980	50.000	FFT (OA1, OA3)	0.999	100.38	0.9792	150.38	1.0	1.00000
5	LB (BL1, BL2)	0.975	58.450	BC (OA1)	0.960	120.00	0.9360	178.45	0.5	0.68460
6	LB (BL1, BL2, BL3)	0.980	58.767	BC (OA1)	0.960	120.00	0.9408	178.77	0.5	0.64946
7	LB (BL1, BL2)	0.975	58.450	FFT (OA1,OA2)	0.998	121.14	0.9731	179.59	1.0	0.55079
8	LB (BL1, BL2, BL3)	0.980	58.767	FFT (OA1, OA3)	0.999	100.38	0.9792	159.15	1.0	0.99997

Case Study II

SPs and their Characteristics

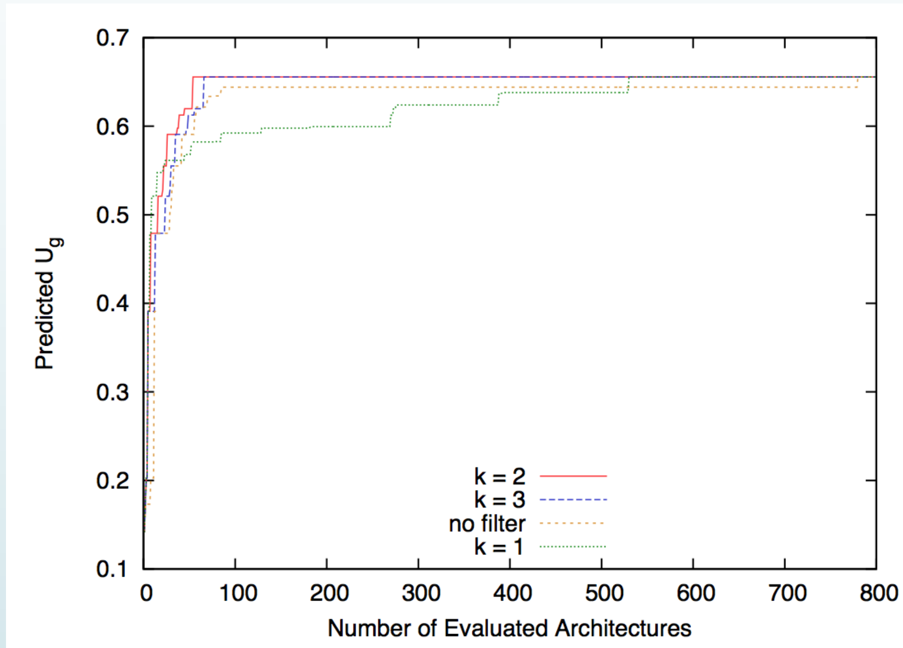
SP i	Capacity C_i (in tps)	E_i (in sec)	Availability (A_i)	Cost (in US$)
Road Map ACME	15.0	0.2	0.900	50
Road Map Pinnacle	12.5	3.0	0.990	100
Road Map ServiceTron	7.5	0.3	0.985	150
Road Map Apex	17.5	1.0	0.975	250
Weather Acme	16.5	0.1	0.980	100
Weather Pinnacle	13.5	5.0	0.999	200
Weather ServiceTron	10.0	0.8	0.995	300
Weather Apex	18.0	0.6	0.990	250
ID Threat Intellifort	13.0	1.5	0.990	500
ID Threat InfoSafe	15.5	2.9	0.985	400
ID Threat CryptIT	17.0	1.8	0.995	550
Make Plan DataCrunch	15.0	48.5	0.940	1500
Make Plan OR Gurus	19.0	92.0	0.990	2000
Make Plan Master Plan	7.0	83.2	0.965	1600
Eval Plan DataCrunch	17.0	5.2	0.985	150
Eval Plan OR Gurus	14.5	9.8	0.995	200
Eval Plan Master Plan	7.5	3.9	0.990	250

Base SSA

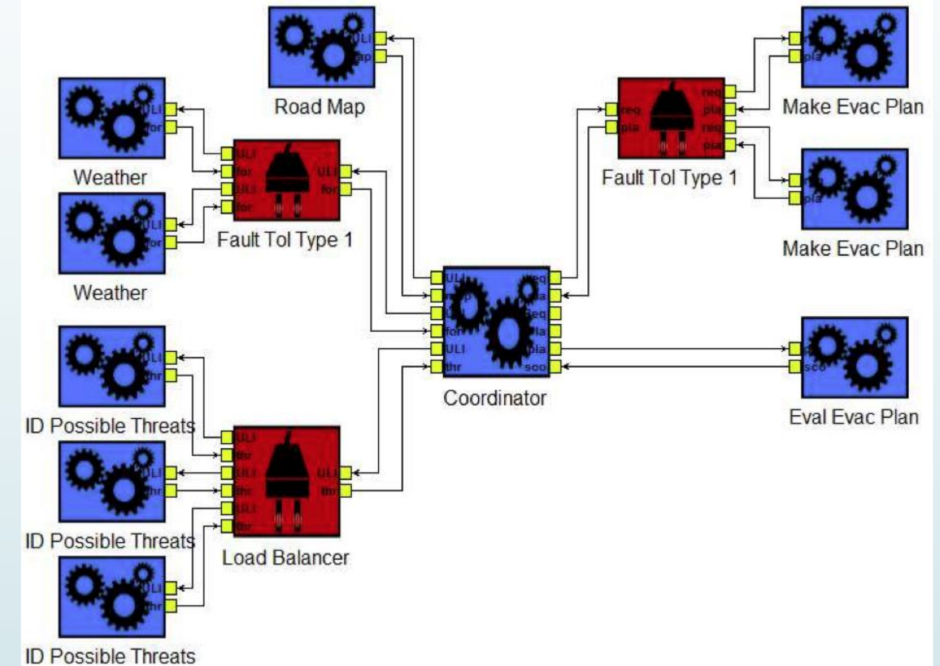


Case Study II – After near optimal self-architecting

Variation of the global utility

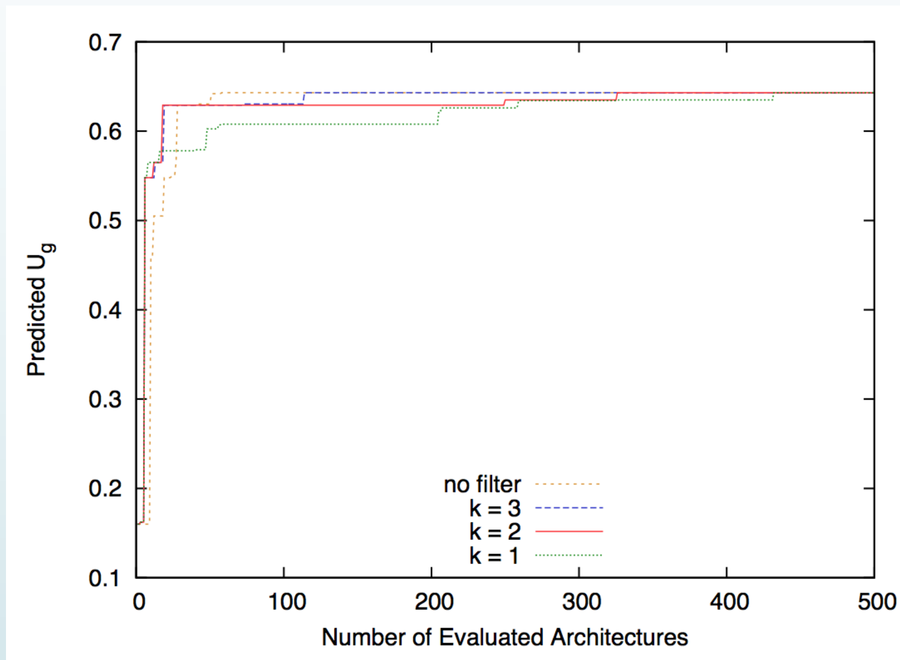


Optimal architecture

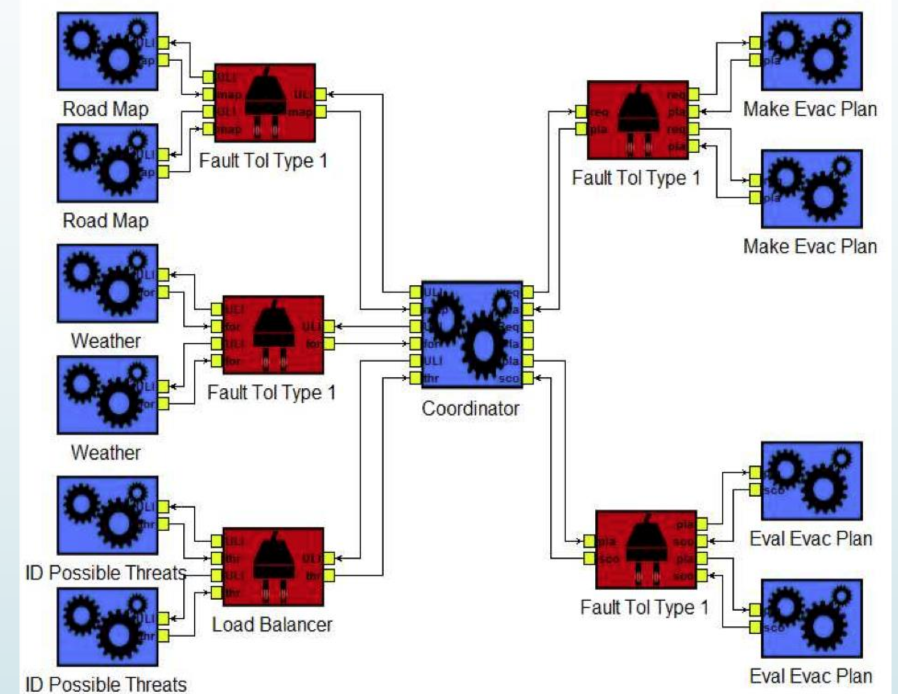


Case Study II – after a performance degradation in an SP

Variation of the global utility during the search due to adaptation



Optimal architecture after adaptation



Conclusion

- SASSY's approach can be used for
  - Self-adaptation
  - Self-healing
  - Self-optimization
  - Evolution: when requirements change, they need to be reflected at the SAS level. From that point, SASSY regenerates a near optimal architecture that satisfies the requirements of the evolved system
- A small change in the environment can lead to substantial changes in the structure and features of near-optimal architectures
- Autonomic management can elevate the end user experience by reacting to emergent problems on the scale of seconds, while human administrators would need hours, possibly days, to devise and implement a new architecture that would properly restore the application's performance