Data Extraction

ACM Articles

D1: Authors
Paolo Arcaini; Elvinia Riccobene; Patrizia Scandurra

D2: Year
2017 (Article 2017; accepted 2016)

D3: Title
Formal Design and Verification of Self-Adaptive Systems with Decentralized Control

D4: Venue
ACM Transactions on Autonomous and Adaptive Systems (TAAS), Vol. 11, No. 4, Article 25

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations (specifically regarding reuse of adaptation logic)

1. No reuse method is proposed; the paper does not analyze or provide software reuse methods for adaptation logic or SASS.
2. No support for modeling uncertainty, probabilistic behavior, or stochastic decision-making.
3. No explicit time model; cannot handle real-time or time-bounded adaptation constraints.
4. Focuses mainly on functional correctness; non-functional properties or QoS are not addressed.
5. All analysis is design-time; no runtime assurance/online adaptation.
6. Scalability limits in model checking and model review for larger systems.

D7: Challenges (developers + reuse)

1. No explicit challenges about software reuse in SASS are reported.
2. Complexity of designing and reasoning about decentralized MAPE-K loops.

3. Interference between multiple loops (conflicting adaptations).

4. Need for formal models and verification to assure functional correctness.

5. Difficulty of modeling interactions between MAPE components and pattern instances.

D8: Proposed Solutions

1. Formalism: Self-adaptive Abstract State Machines (multiagent ASMs) to model managing vs. managed subsystems and MAPE-K loops as first-class entities.

2. Decentralized control modeling: Mechanisms to model distributed MAPE-K loops, interaction relations, and MAPE patterns (e.g., information sharing).

3. Validation techniques:
   3.1 Simulation of self-adaptive behavior.
   3.2 Scenario-based validation using Avalla/AsmetaV to exercise specific adaptation scenarios.

4. Verification techniques:
   4.1 Metaproperty-based model review (detect inconsistent updates, unused knowledge, over-specification).
   4.2 Automated verification of MAPE-K correctness via model checking (checking interaction relations between rules).
   4.3 Standard requirement-level properties (liveness, reachability) via CTL/LTL.

IEEE Articles

D1: Authors
Naeem Esfahani; Ahmed Elkhodary; Sam Malek

D2: Year
2013

D3: Title
A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems

D4: Venue
IEEE Transactions on Software Engineering (TSE), Vol. 39, No. 11, November 2013

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations (specifically regarding reuse of adaptation logic)

1. Assumptions about feature mappings and realizations; if a system cannot realize features cleanly, reuse is harder.

2. Dependence on accurate metrics and observability; limited observability reduces the ability to learn reusable behavior.

3. Domain and implementation coupling; transferring feature models across domains requires effort.

4. Evaluation scale and generality; empirical evaluation is limited in scope and industrial-scale reuse is not fully validated.

5. Handling crosscutting implementation details; complexity in enforcing crosscutting constraints at runtime complicates reuse of adaptation logic.

D7: Challenges (developers + reuse)

1. Concept drift undermines reusable assumptions; design-time models and reusable rules can become invalid at runtime.

2. Crosscutting dependencies complicate reusable adaptation units; adaptation logic often spans multiple components.

3. Configuration space explosion; without feature constraints, reusable adaptation search is infeasible at runtime.

4. Need for practical engineer knowledge encoding; structured knowledge is required for automated reuse.

5. Trade-off between black-box reuse and white-box guarantees; balancing convenience and correctness is challenging.

6. Realizing features in heterogeneous systems; different implementation mechanisms complicate cross-system reuse.

D8: Proposed Solutions

1. Feature-oriented representation of adaptation choices; feature models encode engineers' knowledge about valid configurations, reducing the search space.

2. Black-box, feature-based adaptation unit; allows adaptation logic to be reused without full white-box knowledge of internal structure.

3. Online learning to handle concept drift; incremental learning updates models for reusable decision logic under changing conditions.

4. Hybrid offline and online training; pre-deploy models with offline data and refine at runtime to maintain reuse.

5. Adaptation planning that minimizes disruption; algorithms compute feature selection and enactment to improve practical reuse.

6. Toolchain and prototype implementation; maps features to architecture, supports learning and adaptation, and facilitates reuse across deployments.

D1: Authors
Md Nafee Al Islam; Jane Cleland-Huang; Michael Vierhauser

D2: Year
2024

D3: Title
ADAM: Adaptive Monitoring of Runtime Anomalies in Small Uncrewed Aerial Systems

D4: Venue
SEAMS 2024 (19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. Platform heterogeneity and integration effort; porting detectors, effectors, and knowledge base across different flight controllers and companion computers requires nontrivial engineering.

2. Knowledge base provenance and generality; activation tree and strategies mined from forums and expert input may not generalize beyond the studied datasets and communities.

3. Safety and assurance for reused strategies; mode changes and mitigation strategies have safety implications, requiring additional validation across different vehicle types or missions.

4. Detector realization variability; detectors range from lightweight heuristics to deep-learning models, limiting reuse across platforms with different sensors or sampling rates.

5. Scalability and runtime constraints; multi-drone or high-fidelity detector deployments may face resource and performance limits when reused at larger scale.

6. Dependence on data quality and forum bias; biases or missing anomaly types in source data constrain completeness of reusable artifacts.

D7: Reported developer-facing challenges (affecting reuse)

1. Resource constraints limit reusable detector deployment; onboard CPU, memory, and power require selective monitoring.

2. Heterogeneous sensor and middleware interfaces; differences in flight controllers, sensor availability, and APIs complicate packaging detectors and effectors as reusable modules.

3. Mapping domain knowledge into reusable artifacts; converting forum-mined insights and expert rules into robust activation sequences and strategies is error-prone.

4. Assurance and safety when reusing mitigation strategies; reused adaptations may have mission-critical side effects requiring extra validation.

5. Detector portability and calibration; heuristic thresholds and learned models often need per-platform calibration.

6. Tradeoffs between local and offloaded processing; reusable architectures must consider network availability, latency, and bandwidth.

7. Evolving anomaly patterns; activation trees and detectors must be maintained and updated as new anomalies or co-occurrence patterns emerge.

D8: Proposed solutions and reusable artifacts

1. Activation sequence tree mined from literature, flight-controller docs, and forum posts to guide detector activation; reusable decision artifact.

2. Canary detectors as lightweight, reusable early-warning monitors that trigger more expensive detectors only when needed.

3. Adaptation strategies library (deactivate non-critical services, offload processing, change flight mode) packaged as reusable mitigation patterns with implementation guidelines.

4. Knowledge base construction process (clustering, canary selection thresholds, co-occurrence edges) provided for reproduction and adaptation to new domains.

5. Hybrid monitoring enactment combining onboard canaries, selective detector activation, and conditional offload to GCS to enable feasible detector reuse under resource constraints.

6. Empirical evaluation artifacts and prototype (simulation, flight log replay, real-drone tests) provided as reusable benchmarks and validation templates.

D1: Authors
Martin Pfannemüller; Christian Krupitzer; Markus Weckesser; Christian Becker

D2: Year
2017

D3: Title
A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems

D4: Venue
2017 IEEE International Conference on Autonomic Computing (ICAC)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. Developer expertise and modeling burden; authors must create context feature models, feature-attribute rules, priorities, and cost values, which limits immediate out-of-the-box reuse.

2. Toolchain dependence; reuse depends on the SAT mapping, chosen solver (Sat4J), and FESAS runtime; porting requires engineering effort.

3. Domain specificity of CFMs; feature models and attribute rules are domain-specific, so reusing adaptation logic across domains requires reauthoring or significant adaptation.

4. Runtime overhead and scalability; SAT solving at runtime may be costly for very large CFMs or highly dynamic contexts.

5. Mapping to managed resources; concrete mapping from selected features to runtime actions must be implemented per system, reducing plug-and-play reuse.

6. Partial/uncertain context handling; approach assumes accurate context-to-attribute rules; noisy or missing sensor data can degrade planner effectiveness and require additional adaptation.

D7: Reported developer-facing challenges (affecting reuse)

1. Capturing context and linking it to variability; explicit modeling of context features and cross-tree constraints is essential but time-consuming and error-prone.

2. Conflict resolution between features; unforeseen context combinations require developers to provide priorities/costs for resolution, adding artifacts to maintain for reuse.

3. Bridging model and implementation; mapping planner outputs to concrete runtime actions requires per-system effector implementations, limiting reuse of adaptation logic alone.

4. Tool and representation learning curve; CFMs, CNF translations, and SAT solver usage require expertise, creating adoption and reuse barriers.

5. Handling unanticipated contexts at runtime; planner fallback strategies reduce reliability of reused adaptation policies.

6. Performance constraints for runtime planning; real-time or highly dynamic systems may find SAT solving too slow without incremental solving strategies.

D8: Proposed solutions, artifacts, and reusable elements

1. Context Feature Model (CFM) embedded in the knowledge component; encapsulates context and variability as a reusable knowledge artifact.

2. SAT mapping and solver-based planner; generic mapping to DIMACS CNF and SAT solving allows reuse across deployments when CFM and mappings are supplied.

3. Priorities and costs for conflict resolution; reusable policy mechanism for selecting among multiple valid configurations or resolving conflicts.

4. Generic MAPE-K architecture with a pluggable knowledge component; isolates adaptation logic so the same AL implementation can be reused with different managed resources.

5. Implementation and integration pattern; FESAS framework, socket-based sensor/effector interfaces, and clear workflow provide a reusable template.

6. Support for partial knowledge; planner operates with incomplete context inputs, facilitating reuse in environments with intermittent sensing.

D1: Authors
Teruyoshi Zenmyo; Hideki Yoshida; Tetsuro Kimura

D2: Year
2006

D3: Title
A Self-Healing Technique based on Encapsulated Operation Knowledge

D4: Venue
IEEE (conference/journal proceedings; corporate R&D Toshiba)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. Assumption of correct and complete operation descriptions; incomplete or inaccurate descriptions reduce portability and reuse.

2. Manual effort to author constraint and operation descriptions; creating XML state charts and attribute evaluators requires expertise.

3. Integration and mapping overhead; per-host configuration is required to map component names to operation descriptions, limiting plug-and-play reuse.

4. Implicit trust in operation execution; safety-critical or heterogeneous environments need verification and rollback before reuse.

5. Tooling and platform dependence; AspectJ weaving and configuration file conventions limit cross-platform reuse.

6. Limited evaluation of large, dynamic systems; scalability and coordination challenges are not fully assessed, which may affect reuse.

D7: Reported developer-facing challenges (affecting reuse)

1. Separating component knowledge from system specifics; blending reduces reusability.

2. Capturing dependency information without hard-coding; finding reusable abstractions is difficult.

3. Authoring correct state and transition models; mismatches break reuse.

4. Propagating and resolving root causes across components; requires reliable CSR flows.

5. Non-intrusive integration into existing systems; weaving aspects correctly is challenging.

6. Maintaining operation knowledge over time; updates are needed when components or environments change.

D8: Proposed solutions, artifacts, and reusable elements

1. Component-level encapsulation of operation knowledge; packages constraint and operation descriptions with components for reuse.

2. Dependency injection mechanism (CSR flow); routes system-independent constraints to the appropriate host for execution.

3. Components Manager runtime service; reusable runtime artifact interprets operation descriptions, plans and executes recovery actions.

4. Self-healing aspects (weaving); AspectJ aspects generated from constraints enable non-intrusive, reusable integration.

5. XML-based operation description format and attribute libraries; reusable templates and helper libraries for evaluation and transitions.

6. Propagation and transparency pattern; CSR propagation allows high-level constraint reuse without exposing internal recovery logic.

D1: Authors
Eric Bernd Gil; Ricardo Caldas; Arthur Rodrigues; Gabriel Levi Gomes da Silva; Genaína Nunes Rodrigues; Patrizio Pelliccione

D2: Year
2021

D3: Title
Body Sensor Network: A Self-Adaptive System Exemplar in the Healthcare Domain

D4: Venue
SEAMS 2021 (International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. ROS dependency and platform coupling; porting nodes and message interfaces to non-ROS platforms reduces immediate reuse.

2. Domain specificity of models and knobs; feature mappings, parametric formulas, and controller parameters are tailored to BSN sensors.

3. Controller and parameter tuning burden; gains, setpoints, and monitoring frequencies must be retuned per deployment.

4. Assumptions about sensor models and accuracy; differences in real devices limit direct reuse of learned behaviors.

5. Scalability and multi-patient scenarios; the exemplar focuses on a single patient and needs additional coordination mechanisms for larger deployments.

6. Assurance and certification gaps; lacks integrated formal assurance artifacts or regulatory compliance evidence for clinical reuse.

D7: Reported developer-facing challenges (affecting reuse)

1. Trade-off engineering between reliability and energy; reusable logic must expose knobs and policies to balance QoS goals.

2. Uncertainty handling and scenario coverage; new uncertainties require extending knowledge base and injectors.

3. Mapping high-level goals to low-level actuators; translating setpoints into sensor sampling rates and hub processing rates complicates reuse.

4. Instrumentation and observability requirements; systems must provide equivalent telemetry to adopt the exemplar.

5. Controller portability and expertise needs; reusing the controller requires control-theory expertise.

6. Integration and configuration complexity; ROS launch files, topics, and parameterization must be adapted per deployment.

D8: Proposed solutions, artifacts, and reusable elements

1. Executable exemplar artifact (SA-BSN); packaged ROS artifact with sensors, central hub, probes/effectors, and Strategy Manager/Enactor for reuse.

2. Uncertainty injector and scenario library; reusable mechanisms to inject noise and faults for testing and validating adaptation logic.

3. Parametric formulas and knowledge repository; templates for estimating reliability and energy, adaptable to new sensors or domains.

4. Extensible controller interface and default proportional controller; pluggable controller with hooks to reuse or replace with domain-specific logic.

5. Launch file configuration patterns; documented approach to adapt sensors, patient profiles, and controller parameters.

6. Logging and evaluation artifacts; reusable logs and evaluation metrics to compare adaptation strategies across deployments.

D1: Authors
Cody Kinneer; Rijnard van Tonder; David Garlan; Claire Le Goues

D2: Year
2020

D3: Title
Building Reusable Repertoires for Stochastic Self-Planners

D4: Venue
2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. Domain dependence of tactics and language; repertoires may not transfer directly to other domains without reauthoring or mapping.

2. Offline cost and coverage; extensive chaos experiments and plan generation are needed to build the repertoire.

3. Evaluation overhead at runtime; selecting and evaluating repertoire fragments still consumes resources.

4. Toolchain and representation coupling; dependence on AST representations, Deckard, and Comby reduces plug-and-play reuse.

5. Quality and maintenance of transformation rules; human-crafted syntactic transformations require ongoing maintenance.

6. Scalability and optimality tradeoffs; reuse can improve timeliness but may reduce optimality in large or safety-critical systems.

D7: Reported developer-facing challenges (affecting reuse)

1. High cost of replanning for unanticipated changes; runtime search from scratch is expensive.

2. Constructing repertoires that generalize; identifying useful plan fragments for unseen scenarios is difficult.

3. Balancing evaluation cost and reuse benefit; trade-off between repertoire size/diversity and runtime evaluation.

4. Encoding and analyzing plans as programs; AST-based plan representations require tooling discipline.

5. Creating and validating transformation rules; syntactic transforms must preserve semantics and generality.

6. Tooling and integration effort; integrating clone detection, transformations, and planners requires engineering expertise.

D8: Proposed solutions, artifacts, and reusable elements

1. Chaos‑inspired scenario generation; offline mutation of scenarios to explore change space and create diverse strategies.

2. Clone detection for plan fragments; AST-based detection (Deckard) extracts generalizable plan components into compact repertoires.

3. Rule-based syntactic transformations; declarative templates (Comby) simplify plans and remove redundant subplans.

4. Two-phase workflow (offline repertoire construction + online seeded GP); reusable process separating expensive exploration from lightweight runtime reuse.

5. Empirical evaluation artifacts; exemplar, mutation strategy, and transformation rules serve as reusable templates and benchmarks.

D1: Authors
Emad Albassam; Jason Porter; Hassan Gomaa; Daniel A. Menascé

D2: Year
2017

D3: Title
DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems

D4: Venue
IEEE International Conference on Autonomic Computing (ICAC) 2017

D5: Quality Assessment
Total Quality Score: 9 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. Strong environmental assumptions; relies on fail-stop failures, reliable transport, synchronized clocks, and non-failing DARE components/RACs, limiting reuse in unreliable or Byzantine environments.

2. Architecture mapping dependency; recovery depends on DeSARM-discovered architectures and a DHT-based configuration map, requiring porting effort for systems without similar mechanisms.

3. Planner and placement simplicity; node selection is random in the prototype, limiting reuse in cost- or policy-aware deployments.

4. Limited evaluation scope; only tested on a single EMS exemplar and cluster sizes up to 30 nodes.

5. Tight coupling to RAC semantics; recovery correctness requires RAC-compliant components or adapters.

D7: Reported developer-facing challenges (affecting reuse)

1. Integrating with existing architectures; requires enabling runtime architecture discovery and mapping components to nodes.

2. Providing compatible connectors; RACs must implement quiescence and transaction recovery, adaptation needed for different middleware.

3. Managing environmental assumptions; developers must ensure reliable transport, synchronized clocks, and fail-stop behavior.

4. Coordinating distributed recovery ownership; deterministic election rules must be applied to avoid conflicting recoveries.

5. Extending planning logic for placement and optimization; prototype's simple/random placement must be replaced for production needs.

6. Testing and validating recovery behavior; realistic failure scenarios and end-to-end validation require substantial infrastructure.

D8: Proposed solutions, artifacts, and reusable elements

1. Architecture discovery via DeSARM; automatically traces messages and performs selective gossiping to map runtime components.

2. Distributed configuration map using DHT; supports decentralized decision-making and scales across nodes.

3. Recovery coordination policy; deterministic election ensures a single recovery coordinator per subnet.

4. RAC abstraction for transaction recovery and quiescence; encapsulates recovery logic into reusable connectors.

5. Reconfiguration templates; standardized steps (create, connect, activate, update map) for recovery and adaptation.

6. Empirical validation and tracing; instrumented execution traces and recovery times serve as reusable artifacts for tuning and understanding system behavior.

D1: Authors
Radu Călinescu; Lars Grunske; Marta Kwiatkowska; Raffaela Mirandola; Giordano Tamburrelli

D2: Year
2011

D3: Title
Dynamic QoS Management and Optimization in Service-Based Systems (QoSMOS)

D4: Venue
IEEE Transactions on Software Engineering (TSE), May/June 2011

D5: Quality Assessment
Total Quality Score: 10 / 12

D6: Limitations (specifically regarding reuse of adaptation logic and assets)

1. Heavy formalism and expertise requirement; teams must author QoS goals in probabilistic temporal logics (PCTL/CSL) and build Markov/MDP models.

2. Toolchain coupling; integration of ProProST, PRISM, KAMI, and GPAC limits reuse in other environments without porting or interface adaptation.

3. Model construction and maintenance cost; maintaining accurate Markov/MDP models is labor intensive and affects reuse.

4. Runtime computational cost; probabilistic model checking and optimization can be expensive, limiting online reuse in resource-constrained settings.

5. Observability and instrumentation assumptions; lack of adequate telemetry complicates reuse of the monitoring and adaptation logic.

6. Domain specificity; framework is tailored to service composition and resource allocation, requiring substantial adaptation for other domains.

7. Scalability and multi-objective trade-offs; reuse across larger, heterogeneous deployments may expose scalability or conflict-resolution gaps.

D7: Reported developer-facing challenges (affecting reuse)

1. Mapping high-level QoS goals to formal specifications; translating administrator goals into PCTL/CSL formulas is nontrivial and error-prone.

2. Modeling the system accurately; creating Markov/MDP models that capture interactions, probabilities, and dependencies requires domain knowledge.

3. Parameter learning and observability; runtime parameter estimation depends on reliable telemetry, complicating reuse when data is noisy or missing.

4. Computational overhead of verification/optimization; deciding when to run full analyses versus approximations affects reuse practicality.

5. Integration complexity; connecting QoSMOS components with service registries, orchestration engines, and legacy systems requires engineering effort.

6. Tool and process friction; multi-tool workflows increase cognitive and operational overhead.

7. Handling unanticipated changes; offline-derived policies may not generalize to unforeseen service behavior or failures without extra scenario coverage.

D8: Proposed solutions, artifacts, and reusable elements

1. Formal specification patterns and ProProST; templates and structured grammars reduce the burden of writing probabilistic temporal logic formulas.

2. Model-based evaluation with PRISM; reusable pipeline translating service compositions into Markov/MDP models and applying probabilistic model checking.

3. Runtime monitoring + Bayesian parameter adaptation (KAMI); reusable monitoring and learning components keep adaptation logic relevant.

4. Planning and execution via GPAC; reusable planning/execution component selects services and allocates resources using model-based analysis.

5. Two complementary adaptation mechanisms; dynamic service selection (binding) and resource allocation/parametrization for internally hosted services as reusable strategies.

6. Validation artifacts; TeleAssistance case study and simulation setups are reusable for evaluating QoS strategies in similar domains.

D1: Authors
Hassan Gomaa; Koji Hashimoto

D2: Year
2012

D3: Title
Dynamic Self-Adaptation for Distributed Service-Oriented Transactions

D4: Venue
SEAMS 2012 (International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Protocol specificity — Patterns are tightly coupled to Two-Phase Commit semantics; reuse for other coordination protocols requires adaptation.

2. Connector implementation burden — Developers must implement service and coordinator connectors that encapsulate state machines; platform-specific engineering is required.

3. Assumptions about quiescence and buffering — Systems must support quiescent states and buffering; real-time or unbufferable services limit reuse.

4. Stateful, concurrent services complexity — Mapping connector state machines to multithreaded/concurrent services can be error-prone.

5. Limited empirical/developer evidence — Prototype-based validation; industrial adoption and developer effort for reuse are not demonstrated.

6. Scalability and heterogeneity — Large-scale or heterogeneous deployments are not evaluated; reuse at scale is uncertain.

7. Tooling and integration gap — Dependency on SASSY runtime; teams without it must reimplement orchestration and connectors, increasing adoption cost.

D7: Reported developer-facing challenges (affecting reuse)

1. Mapping abstract patterns to concrete platforms — Translating diagrams and state machines into connector code, message formats, and deployment artifacts.

2. Ensuring safe quiescence — Determining true quiescent states under concurrency to prevent lost or inconsistent transactions.

3. Handling partial failures and timeouts — Integrating failure detection and recovery logic consistently with adaptation patterns.

4. Concurrency and buffering semantics — Correctly implementing queueing, ordering, and transactional guarantees.

5. Coordination between change-management and component control layers — Ensuring passivate/reactivate handshakes are properly timed.

6. Testing and validation of adaptation sequences — Verifying correctness of adaptation sequences under all scenarios.

7. Platform and middleware dependencies — Adapting connectors to different messaging stacks or deployment environments.

D8: Proposed solutions, artifacts, and reusable elements

1. Adaptation connectors (service and coordinator connectors) — Encapsulate adaptation state machines and hide complexity from services; primary reusable artifact.

2. Adaptation state machines and sequence diagrams — Formalized lifecycles and message flows as reusable blueprints.

3. Passivate / Reactivate protocol and buffering rules — Reusable protocol defining safe quiescence and reactivation of components.

4. Encapsulation of adaptation logic in connectors — Separates adaptation concerns from business logic; promotes reuse.

5. SASSY framework integration — Provides reusable infrastructure (templates, connectors, reconfiguration commands) for deployment.

6. Design patterns catalog — Contextualizes the transaction pattern among other adaptation patterns to guide selection for reuse.

D1: Authors
Sona Ghahremani; Holger Giese; Thomas Vogel

D2: Year
2017

D3: Title
Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures

D4: Venue
2017 IEEE International Conference on Autonomic Computing (ICAC 2017)

D5: Quality Assessment
Total Quality Score: 10 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Domain and model dependence — Utility patterns and adaptation rules are tightly coupled to the architectural metamodel and mRUBIS semantics; porting to new domains requires re-mapping.

2. Assumptions on rule independence and commutativity — Optimality relies on assumptions that may not hold in heterogeneous systems, limiting safe reuse.

3. Authoring burden — Defining utility patterns, sub-functions, and adaptation rules requires domain knowledge and careful tuning.

4. Utility elicitation and calibration — Transferring utility sub-functions and rule costs to new systems requires new measurements or expert input.

5. Runtime model requirement — Systems must have a causally connected architectural runtime model; absence requires instrumentation and model engineering.

6. Hidden assumptions about failure semantics — Pattern-based failure assumptions may not match other system behaviors, requiring redesign.

7. Partial evaluation of human factors — Developer effort, maintainability, and tooling for rule/pattern reuse are not fully assessed.

D7: Reported developer-facing challenges (affecting reuse)

1. Pattern and rule portability — Manual mapping of domain concepts (components, failures, interfaces) to patterns/rules is error-prone.

2. Utility definition and contextualization — Translating high-level goals into utility sub-functions must be repeated per system.

3. Ensuring assumptions hold — Correctness/optimality guarantees rely on assumptions that developers must verify for each reuse target.

4. Rule conflict resolution and ordering — Identifying conflict-free rule subsets and ordering them by impact requires careful reasoning and tooling.

5. Instrumentation and runtime model availability — Reuse depends on a causally connected runtime model; retrofitting legacy systems is challenging.

6. Authoring and maintenance cost — Maintaining patterns, rules, and utility functions over time adds sustained developer effort.

7. Scalability vs. expressiveness tradeoff — Richer patterns may break incremental evaluation assumptions; balancing expressiveness and efficiency is nontrivial.

D8: Proposed solutions, artifacts, and reusable elements

1. Pattern-based utility specification — Reusable templates for positive/negative architectural patterns and per-match utility sub-functions.

2. Pattern-based adaptation rules — Rules expressed as in-place model transformations tied to patterns; templates reusable across systems once mapped to the target metamodel.

3. Incremental utility computation — Algorithmic technique to update utility incrementally (compute Mnew, Mdel) rather than globally; reusable optimization.

4. Greedy, impact-ordered rule selection — Heuristic for conflict-free rule selection by impact and cost; reusable planning approach.

5. Model-driven engineering (MDE) toolchain — Metamodels, story patterns, model queries, and in-place transformations as portable engineering assets.

6. mRUBIS benchmark and experimental recipes — Evaluation setup, metrics, and baseline comparisons as reusable templates for testing in other systems.

D1: Authors
Javier Cámara; Pedro Correia; Rogério de Lemos; David Garlan; Pedro Gomes; Bradley Schmerl; Rafael Ventura

D2: Year
2013

D3: Title
Evolving an Adaptive Industrial Software System to Use Architecture-Based Self-Adaptation

D4: Venue
SEAMS 2013 (International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Legacy coupling and refactoring cost — Existing built-in adaptation had to be removed/rewritten; Rainbow reuse requires engineering effort.

2. Need to expose internals / invasive instrumentation — Probes/effectors require adding translation server and instrumenting code paths; closed or safety-critical systems limit reuse.

3. Translation/adapter maintenance burden — TCP server and data store are bespoke glue; maintaining adapters across versions is ongoing cost.

4. Domain-specific customization — Architectural style, operators, tactics, and utility preferences must be tailored; not plug-and-play.

5. Manual scale-out and operational steps — Some scaling remained manual; limits full automation reuse.

6. Performance overhead and timing assumptions — Instrumentation and strategy execution can affect timing; reuse in hard real-time systems may be problematic.

7. Testing and assurance gaps — Reusable assurance/certification for safety-critical systems not provided.

D7: Reported developer-facing challenges affecting reuse

1. Refactoring legacy adaptation — Removing/disabling built-in adaptation is time-consuming and error-prone.

2. Exposing and mapping internal state — Identifying probes and keeping model synchronized requires invasive changes.

3. Designing translation/adapter infrastructure — Building TCP server and local data store, mapping semantics, requires engineering judgment.

4. Defining operators/tactics/strategies — Translating legacy controls to Rainbow operators, composing tactics/strategies, requires domain expertise.

5. Performance tuning and validation — Ensuring throughput improvements without regressions requires profiling and iterative adjustments.

6. Handling partial automation and human processes — Manual operational tasks integration is challenging.

7. Testing and regression risk — Refactoring/instrumentation introduces risk; validation required.

8. Effort estimation and planning — Estimating man-hours and resource allocation for customization is difficult; prior benchmarks help.

D8: Proposed solutions and reusable artifacts

1. Rainbow platform — Reusable infrastructure: model manager, architecture evaluator, adaptation manager, strategy executor, Stitch language.

2. Translation/adapter pattern — TCP server + local data store mediates probes/effectors; reusable for integrating legacy systems.

3. Customization points / architectural style mapping — Mapping DCAS concepts to Rainbow elements (DeviceT, ProcessorNodeT, DBServerT), operators, properties; reusable methodology.

4. Operator/tactic/strategy templates — Reusable examples of operators, tactics, strategies adaptable to similar middleware.

5. Refactoring checklist and instrumentation guidance — Playbook for exposing internals, adding probes/effectors, validating model synchronization.

6. Empirical effort and performance data — Reusable benchmarks for planning integration projects.

D1: Authors
Sona Ghahremani; Holger Giese; Thomas Vogel

D2: Year
2017

D3: Title
Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures

D4: Venue
2017 IEEE International Conference on Autonomic Computing (ICAC 2017)

D5: Quality Assessment
Total Quality Score: 10 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Domain and model dependence — Utility patterns and adaptation rules are tightly coupled to the architectural metamodel and mRUBIS semantics; porting to new domains requires re-mapping.

2. Assumptions on rule independence and commutativity — Optimality relies on assumptions that may not hold in heterogeneous systems, limiting safe reuse.

3. Authoring burden — Defining utility patterns, sub-functions, and adaptation rules requires domain knowledge and careful tuning.

4. Utility elicitation and calibration — Transferring utility sub-functions and rule costs to new systems requires new measurements or expert input.

5. Runtime model requirement — Systems must have a causally connected architectural runtime model; absence requires instrumentation and model engineering.

6. Hidden assumptions about failure semantics — Pattern-based failure assumptions may not match other system behaviors, requiring redesign.

7. Partial evaluation of human factors — Developer effort, maintainability, and tooling for rule/pattern reuse are not fully assessed.

D7: Reported developer-facing challenges (affecting reuse)

1. Pattern and rule portability — Manual mapping of domain concepts (components, failures, interfaces) to patterns/rules is error-prone.

2. Utility definition and contextualization — Translating high-level goals into utility sub-functions must be repeated per system.

3. Ensuring assumptions hold — Correctness/optimality guarantees rely on assumptions that developers must verify for each reuse target.

4. Rule conflict resolution and ordering — Identifying conflict-free rule subsets and ordering them by impact requires careful reasoning and tooling.

5. Instrumentation and runtime model availability — Reuse depends on a causally connected runtime model; retrofitting legacy systems is challenging.

6. Authoring and maintenance cost — Maintaining patterns, rules, and utility functions over time adds sustained developer effort.

7. Scalability vs. expressiveness tradeoff — Richer patterns may break incremental evaluation assumptions; balancing expressiveness and efficiency is nontrivial.

D8: Proposed solutions, artifacts, and reusable elements

1. Pattern-based utility specification — Reusable templates for positive/negative architectural patterns and per-match utility sub-functions.

2. Pattern-based adaptation rules — Rules expressed as in-place model transformations tied to patterns; templates reusable across systems once mapped to the target metamodel.

3. Incremental utility computation — Algorithmic technique to update utility incrementally (compute Mnew, Mdel) rather than globally; reusable optimization.

4. Greedy, impact-ordered rule selection — Heuristic for conflict-free rule selection by impact and cost; reusable planning approach.

5. Model-driven engineering (MDE) toolchain — Metamodels, story patterns, model queries, and in-place transformations as portable engineering assets.

6. mRUBIS benchmark and experimental recipes — Evaluation setup, metrics, and baseline comparisons as reusable templates for testing in other systems.

D1: Authors
Nicolas Farabegoli; Mirko Viroli; Roberto Casadei

D2: Year
2024

D3: Title
Flexible Self-organisation for the Cloud-Edge Continuum: a Macro-programming Approach

D4: Venue
2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Domain and runtime coupling — Assumes a macro-programming/aggregate computing runtime; reuse outside that paradigm (traditional microservices, ADLs) requires substantial porting.

2. Middleware dependency — Requires middleware supporting forwarding chains, message packing, and neighbour state exchange; teams without such middleware must implement it.

3. Assumptions about network and timing — Formal results rely on assumptions (neighbourhood relations, message delivery semantics, asynchronous rounds); real networks may break these assumptions.

4. State externalization and stateless instance model — Reusing components that rely on local persistent state or complex resources requires redesign.

5. Heterogeneity and resource constraints — Simulation-based evaluation; practical reuse across heterogeneous devices requires engineering effort.

6. Operational complexity of forwarding chains — Multi-hop forwarding, surrogate collapse, and neighbour state routing must be managed for deployment.

7. Security, privacy, and governance — Offloading collective computations raises privacy/regulatory concerns; additional assurance needed for sensitive domains.

8. Tooling and developer ergonomics — Formal models and simulations exist, but limited IDE, packaging, or automated partitioning support for reuse.

D7: Reported developer-facing challenges (affecting reuse)

1. Partitioning macro-programs into reusable components — Deciding which functions are local vs collective is nontrivial.

2. Preserving self-stabilisation under different deployments — Offloading/forwarding must not change eventual behaviour.

3. Managing forwarding chains and surrogate devices — Mapping owner→surrogate chains and routing neighbour state complicates reuse.

4. Handling heterogeneity and constrained devices — Component implementations must adapt to devices with different compute, sensors, and energy budgets.

5. Testing and debugging distributed collective behaviour — Reproducing emergent behaviour across tiers is challenging.

6. Message packaging and timeliness — Ensuring neighbour state and inputs are consistent and timely across offloaded instances.

7. Deployment orchestration and lifecycle — Dynamic mobility/topology changes require redeployment or reconfiguration.

8. Expertise gap — Developers must understand aggregate semantics and formal guarantees to safely reuse components.

D8: Proposed solutions and reusable artifacts

1. Macro-component model (DAG of components) — Reusable design abstraction for local/collective components, ports, and bindings.

2. Forwarding/offloading mechanism and surrogate model — Reusable runtime pattern: forwarding chains, surrogate collapse, and message routing rules.

3. Formal operational semantics and deployment-independence result — Reusable correctness argument: self-stabilising components maintain behaviour across deployments.

4. Component instance execution model (asynchronous rounds) — Reusable execution contract detailing inputs, neighbour states, and outputs.

5. Implementation + simulation framework (open-source) — Reusable simulation tools and examples for testing partitioning and non-functional trade-offs.

6. Design guidelines for partitioning — Prefer local execution for sensor-bound tasks; offload compute-heavy services while preserving neighbour semantics.

7. Evaluation patterns — Reusable metrics and experiments for validating self-stabilisation, energy footprint, and deployability decisions.

D1: Authors
Gabriel Tamura; Norha M. Villegas; Hausi A. Müller; Laurence Duchien; Lionel Seinturier

D2: Year
2013

D3: Title
Improving Context-Awareness in Self-Adaptation using the DYNAMICO Reference Model

D4: Venue
SEAMS / IEEE SEAMS 2013 (conference paper)

D5: Quality Assessment
Total Quality Score: 9 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Tight coupling to specific middleware and formats — Implementation depends on RDF-based COb specifications, SCA/FRASCATI, QOS-CARE, and SMARTERCONTEXT; reuse requires adopting these technologies or substantial adaptation.

2. Artifact granularity and packaging not standardized — Monitoring strategies are delivered as bundles of class files and SCA descriptors; no standard repository or packaging model is provided.

3. Domain and exemplar specificity — Evaluation focuses on Znn.com and SOA governance; artifacts may not transfer without redesign.

4. Limited automation for cross-platform reuse — Strategy synthesis is tied to RDF/SLA and QOS-CARE; porting requires re-implementation.

5. Insufficient documentation of extension points — Lack of detailed API/contract descriptions hinders reuse by third parties.

6. Scalability and operational assumptions — Assumes deployable context gatherers and dynamic deployment; reuse in constrained/regulatory environments is limited.

D7: Reported developer-facing challenges (affecting reuse)

1. Heterogeneous platform dependency and portability — Artifacts tied to SCA/FRASCATI and QOS-CARE; reuse requires platform adoption or adaptation.

2. Specification-to-artifact gap (SLA → monitoring code) — Mapping high-level SLA constructs to concrete monitors is nontrivial; reuse requires maintaining templates and transformation rules.

3. Lack of standardized packaging and discovery — Manual packaging and ad-hoc deployment impede reuse at scale.

4. Interdependence between monitoring and adaptation artifacts — Tight coupling increases cost of reusing individual components.

5. Implicit assumptions about runtime deployment and third-party sensors — Reuse requires adapting artifacts to different deployment environments or providing shims.

6. Limited documentation of extension points and contracts — Absence of detailed API/contracts increases integration friction and risk.

D8: Proposed solutions and reusable artifacts

1. Model-driven synthesis of monitoring strategies from SLA (RDF → deployable artifacts) — Reusable pattern: formal SLA encoding generates monitoring artifacts automatically; requires compatible RDF vocabularies and middleware.

2. Separation of three feedback loops (CO-FL, A-FL, M-FL) as an architectural pattern — Reusable blueprint: modularization enables independent reuse of monitoring or adaptation components.

3. Use of graph-based reconfiguration (QOS-CARE / graph transformations) — Reusable mechanism for expressing structural changes to monitoring/adaptation infrastructures.

4. Dynamic deployment artifacts (monitoring strategy bundles) — Practical reuse unit: self-contained bundles of implementation files, descriptors, and service lists; could be standardized for cross-project reuse.

D1: Authors
Frederico Alvares; Gwenaël Delaval; Eric Rutten; Lionel Seinturier

D2: Year
2017

D3: Title
Language Support for Modular Autonomic Managers in Reconfigurable Software Components

D4: Venue
2017 IEEE International Conference on Autonomic Computing (ICAC)

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Toolchain coupling and platform dependence — Depends on Heptagon/BZR and FraSCAti; teams not using these tools must port models and runtime bindings.

2. Formal-methods expertise requirement — Reuse of generated artifacts requires understanding DCS and Heptagon/BZR semantics.

3. Residual scalability constraints — Modular DCS reduces explosion, but very large hierarchies may still be expensive.

4. Visibility vs. abstraction trade-off — Contracts must expose internal behavior; poorly designed contracts limit substitutability.

5. Runtime heterogeneity and distribution — Focuses on local deployment; reuse across heterogeneous runtimes or distributed systems is limited.

6. Maintenance of generated artifacts — Evolving source modules or target platform may require regeneration and retuning, complicating long-term reuse.

D7: Reported developer-facing challenges (affecting reuse)

1. Lack of modular language constructs prevents reuse and substitutability.

2. Combinatorial explosion in controller synthesis blocks practical reuse.
3. Need for explicit contracts and abstractions to enable safe substitution.
4. Tight coupling between behavioral models and verification tools reduces portability.
5. Difficulty of composing and distributing synthesized controllers.
6. Overhead of learning and using the generative toolchain.

D8: Proposed solutions and reusable artifacts

1. Language extensions for modularity (Ctrl-F) — Inheritance, overriding, and visibility control for components and behaviors; enables reuse of behavior fragments.
2. Contract-based abstractions — Contracts in Heptagon/BZR expressing assumptions/guarantees; reusable interfaces for adaptive components.
3. Modular compilation and modular DCS — Produces hierarchical nodes for local DCS, reducing synthesis cost and enabling reuse of controllers.
4. Generative toolchain and runtime integration — Automated generation of executable controllers and FraSCAti integration patterns; reusable across deployments using the same middleware.
5. Concrete exemplar and templates — RUBIS/Brownout case study and Ctrl-F component templates as reusable starting points.
6. Guidelines for contract design and component abstraction — Patterns and examples for selecting controllable variables and exposed behaviors to maximize substitutability and reuse.

D1: Authors
Mateo Sanabria; Ivana Dusparic; Nicolás Cardozo

D2: Year
2024

D3: Title
Learning Recovery Strategies for Dynamic Self-healing in Reactive Systems

D4: Venue
SEAMS 2024 (19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Application-specific learned strategies — Learned recovery sequences are tied to the observed state space and concrete actions; direct reuse across different systems is limited.

2. Dependency on runtime platform and COP — Extraction as Context-Oriented Programming variations depends on COP frameworks (ContextScala/ContextErlang) and reactive runtimes (REScala), reducing out-of-the-box portability.

3. Instrumentation and action exposure requirement — Learning requires corrective actions to be observable at runtime; systems lacking such instrumentation cannot reuse the approach without invasive changes.

4. Exploration coverage and data dependence — Reuse presumes sufficient exploration of failure states during training; strategies learned in one deployment may not generalize to other workloads or environments.

5. State-space and scalability concerns — For complex systems, the state/action space may grow large, making learning and reuse of compact strategies harder.

6. Human-in-the-loop dependency — The approach leverages corrective sequences provided by users/agents; reuse in fully automated pipelines may be constrained if human guidance is required.

D7: Reported developer-facing challenges (affecting reuse)

1. Reliance on predefined hooks and fixed recovery points — Anticipating failure points and embedding hooks prevents reuse of adaptation logic across unanticipated failure modes and different systems.

2. Coarse-grained recovery actions (component replacement) — Replacing whole components reduces the ability to reuse finer-grained adaptation logic or compose reusable recovery fragments.

3. Need for expressive, reusable monitors — Monitors must express relevant failure predicates; insufficiently expressive or hard-to-reuse monitors hinder transferring adaptation logic.

4. Action exposure and API design for learning — Atomic actions must be observable; poor or inconsistent action APIs impede reuse of learned strategies.

5. Generalization and portability of learned strategies — Learned strategies are tied to the specific state/action space and workload; developers face the challenge of producing reusable strategies across deployments.

6. Instrumentation and integration burden — Integrating the reactive runtime, COP layers, and learning components requires nontrivial engineering effort, creating a barrier to reuse as drop-in adaptation assets.

D8: Proposed solutions and reusable artifacts

1. Flexible declarative monitor templates — Reusable predicate-based monitor definitions that can be instantiated and refined at runtime to detect failure states across the system.

2. Learning model and reaction map — A reusable learning component (Q-learning with options) producing a reaction map: mapping detected fault states to learned recovery sequences. The reaction map is a portable artifact that can be inspected, exported, or adapted.

3. Extraction as COP variations — Learned sequences are packaged as Context-Oriented Programming variations (layers) dynamically activated; reusable building blocks for runtime adaptation in COP-enabled systems.

4. Variation manager and runtime glue — A reusable runtime module that manages activation of learned variations when monitors fire; adaptable to other reactive/COP platforms.

5. Prototype implementations and exemplar code — Prototype implementations (mouse-tracking app, DeltaIoT integration) and evaluation scripts serve as reusable reference implementations and templates.

D1: Authors
Omid Gheibi; Danny Weyns

D2: Year
2022

D3: Title
Lifelong Self-Adaptation: Self-Adaptation Meets Lifelong Machine Learning

D4: Venue
SEAMS 2022 (International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 10 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Learner-centric scope — Targets evolution of learning models only; does not provide reusable runtime code or architectural artifacts for adaptation logic across different managing systems without adapting the lifelong loop interfaces.

2. Knowledge representation coupling — Knowledge Manager and task labeling assume particular triplet formats and metadata; reuse in other domains requires mapping or reengineering of knowledge schemas.

3. Tooling and integration effort — Integrating the lifelong loop with an existing MAPE-K system requires connectors for probes, learner state, and effectors; conceptual description exists, but reusable adapters are not delivered.

4. Learner dependence and portability — Instantiations rely on specific learners (SGD, HAT) and retraining strategies; porting to other learners or systems with different feature spaces requires adaptation.

5. Operational overhead and cadence — Loop periodicity and storage of knowledge triplets impose runtime and storage costs; reuse in resource-constrained systems may require tailoring.

6. Assumptions about non-evolving software — Assumes no runtime evolution of managing or managed software; reuse in contexts needing co-evolution of models and software is unsupported.

D7: Reported developer-facing challenges (affecting reuse)

1. Mapping and instrumentation burden — Developers must provide probes and interfaces so the Knowledge Manager can collect required triplets; reuse requires re-instrumenting target systems to expose similar metadata.

2. Task labeling and domain adaptation — Detecting and labeling new tasks depends on statistical change detection and domain thresholds; reuse across domains requires tuning and expertise.

3. Model evolution strategy selection — Choosing how to evolve models (retrain, switch learners, create new models) is domain-dependent; reusable policies must expose knobs and decision criteria.

4. Knowledge schema and interoperability — Knowledge triplet format and meta-knowledge assumptions create coupling that must be reconciled when reusing the architecture in different systems.

5. Human-in-the-loop and validation overhead — When stakeholder input is required, integrating human workflows reduces plug-and-play reuse.

6. Scalability and storage management — Large deployments require decisions about historical knowledge storage, pruning, summarization, or caching; reuse requires planning these strategies per deployment.

D8: Proposed solutions and reusable artifacts

1. Reusable architecture (lifelong learning loop) — Layered architecture (Knowledge Manager, Task Manager, Knowledge-based Learner, Task-based Knowledge Miner) that can be instantiated as a template for adding lifelong ML capabilities.

2. Knowledge triplet abstraction — Reusable data abstraction ((input, state, output)) and task labeling; serves as a template for instrumentation and storage.

3. Task detection and labeling workflow — Reusable pattern: periodically sample knowledge, detect distributional changes, assign task labels, and notify learner.

4. Model evolution strategies — Reusable decision points and options: retrain existing models, switch learners, or create new models; concrete instantiations (SGD ↔ HAT) can be adapted.

5. Integration pattern with MAPE-K — Shows placement of lifelong loop relative to probes, learner, and effectors; reusable for implementing data flows and integration.

6. Evaluation recipes — Guidance on trigger cadence, feature sets, and metrics (utility, packet loss) that can serve as starting points for other domains.

D1: Authors
Valeria Cardellini; Emiliano Casalicchio; Vincenzo Grassi; Stefano Iannucci; Francesco Lo Presti; Raffaela Mirandola

D2: Year
2012

D3: Title
MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems

D4: Venue
IEEE Transactions on Software Engineering

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Assumption of rich service metadata and SLAs — Requires detailed, up-to-date QoS parameters and operation descriptions; reuse without metadata requires service profiling.

2. Modeling and domain coupling — Adaptation models (composition grammar, QoS utility functions, LP formulation) are tailored to SOA/composite-service settings and specific QoS metrics; reuse in other domains requires re-expressing models and utility functions.

3. Single-authority assumption — MOSES assumes a single adaptation authority; reuse in federated/multi-stakeholder environments requires additional coordination protocols.

4. Stateful service handling complexity — Reuse for stateful services requires careful handling of state transfer and consistency; not fully automated.

5. Instrumentation and monitoring dependency — Effective reuse depends on runtime monitoring; systems without equivalent telemetry need added instrumentation.

6. Scalability of optimization — LP-based planner scales to moderate composition sizes; very large compositions or frequent adaptation cycles may require approximation or hierarchical planning.

D7: Reported developer-facing challenges (affecting reuse)

1. Need for accurate, standardized service descriptions and SLAs — Without precise QoS parameters and operation constraints, adaptation logic cannot be reused reliably.

2. Mapping abstract compositions to concrete services — Translating workflows into concrete bindings requires per-deployment configuration; reusable policies must expose mapping hooks and variability points.

3. Instrumentation and observability requirements — Reuse requires target systems to expose monitored parameters; adding probes and ensuring causal connection is nontrivial.

4. Handling stateful services during reconfiguration — Developers must manage state transfer, quiescence, and consistency when reusing adaptation actions that assume stateless operations.

5. Integration and deployment complexity of adaptation middleware — Embedding MOSES (MAPE-K components, model manager, solver) into existing SOA stacks requires adapting orchestration artifacts and connectors.

6. Scalability and performance tuning of planners — Developers must tune adaptation frequency, solver cadence, and model granularity; deployment-specific tuning hinders out-of-the-box reuse.

7. Domain specificity of utility and QoS models — Utility weights and QoS aggregation formulas are stakeholder-specific; reusing adaptation logic across domains requires re-eliciting and

configuring models.

D8: Proposed solutions and reusable artifacts

1. Executable methodology and prototype (MOSES tool) — Implements MAPE-K loop, model manager, monitor, analyzer, planner (LP solver), and executor; can be adapted and extended.

2. Problem-space taxonomy and composition grammar — Reusable conceptual artifacts guiding how to model target systems for adaptation.

3. SLA and QoS modeling templates — Reusable SLA representations and utility aggregation functions (weights, normalization) adaptable to new deployments.

4. LP-based planner integrating service selection and coordination patterns — Reusable planning formulation and solver integration pattern; can be reused or replaced with approximations.

5. Adaptation action catalog — Patterns for service selection, coordination pattern selection, and service tuning; reusable adaptation primitives.

6. Prototype integration patterns — Concrete examples and code for monitoring instrumentation, planner orchestration connection, and execution; reusable templates.

7. Evaluation artifacts and benchmarks — Experimental setups and metrics that can be reused to validate adaptation behavior in other systems.

D1: Authors
Cody Kinneer; Zack Coker; Jiacheng Wang; David Garlan; Claire Le Goues

D2: Year / Venue
2018

D3: Title
Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search

D4: Venue
SEAMS ’18 (13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Naïve reuse can reduce utility — Simply seeding the GP population with full prior plans may perform worse than replanning due to evaluation costs and search dynamics.

2. Exponential evaluation cost with plan size — Fitness evaluation grows exponentially with plan size (state-tree expansion), making long reused plans expensive.

3. Dependence on parallel hardware — Mitigations (early termination, kill_ratio) rely on multi-core resources; reuse may be impractical without them.

4. Parameter sensitivity — Effective reuse requires tuning scratch_ratio, kill_ratio, and trimming heuristics; these are problem- and hardware-dependent.

5. Plan expressiveness and applicability gaps — Missing guards and applicability checks limit safe reuse in new contexts.

6. Domain specificity of plans and tactics — Plan templates and tactics are tailored to cloud/web scenarios; porting to other domains requires redesign.

7. Invalid actions and penalization — Plans may generate semantically invalid actions; additional validation is needed for safe reuse.

8. Scalability and multi-actor settings not fully explored — Reuse behavior in large, multi-tenant, or multi-agent systems is not demonstrated.

D7: Developer-facing challenges affecting reuse

1. High evaluation cost of reused (long) plans — Long plans are expensive to evaluate due to probabilistic state-tree expansion, which can negate reuse benefits.

2. Naïve seeding harms search — Copying prior plans into GP population may reduce utility because costly long plans block exploration.

3. Need for parallel evaluation resources — Early termination and kill_ratio strategies rely on multi-core evaluation; without it, reuse efficiency suffers.

4. Tuning and configuration burden for reuse parameters — Scratch_ratio, kill thresholds, and trimming heuristics require careful tuning per domain/hardware.

5. Plan applicability and missing guards — Omitted applicability checks make reused plan fragments potentially unsafe.

6. Handling invalid or out-of-date tactics — Reused plans may rely on outdated tactic semantics; repair or trimming is needed.

7. Bias introduced by starting plan objectives — Reused plans can bias GP search toward similar objectives, limiting diversity and reuse effectiveness.

8. Plan size and verbosity management — Large plans increase evaluation cost; developers must manage verbosity via penalties and parsimony measures.

9. Model synchronization responsibility — Reuse assumes the system model is synchronized with reality; ensuring this synchronization is a practical challenge.

D8: Proposed solutions and reusable artifacts

1. Seeding strategy with mixed population (scratch_ratio) — Initialize only a fraction of population with reused plan fragments; the rest are random; reusable pattern for controlled plan reuse.

2. Early termination / kill_ratio for long evaluations — Stop evaluation after a configurable fraction; practical, parallel-friendly mechanism for bounding evaluation time.

3. Plan trimmings (random subtree extraction) — Seed smaller plan subtrees instead of full plans; produces compact, high-utility reusable fragments.

4. Fitness heuristics and penalties — Apply invalid-action, verbosity, and parsimony penalties; reusable for safer, efficient plan reuse.

5. Parallelized GP evaluation (implementation pattern) — Multi-core evaluation to amortize expensive fitness computations; reusable engineering pattern.

6. Plan representation and grammar — Concrete plan grammar (tactics, try/catch, loops) and tree representation; reusable artifact for other planners.

7. Search configuration and SPEA2 multi-objective setup — Reproducible configuration for elite set sizes, horizon/depth limits; reusable in other GP planners.

8. Empirical evaluation artifacts — Benchmarks, PRISM comparison, Wilcoxon testing, metrics (fitness, time-to-repair, diversity); reusable for validating reuse strategies.

D1: Authors
Rafael R. Aschoff; Andrea Zisman

D2: Year
2012

D3: Title
Proactive Adaptation of Service Composition

D4: Venue

SEAMS (International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 9 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Stateless-only assumption — The approach only supports stateless services; adaptation logic for stateful services or state transfer cannot be reused.

2. Signature dependency complexity — Resolving signature dependencies may cascade changes recursively, complicating reuse of replacement fragments.

3. Prototype monitoring & prediction simplicity — Monitoring is limited to intercepting calls and simple EWMA QoS prediction; limits reliable reuse of precomputed replacement strategies.

4. Service discovery assumed, not fully integrated — Reuse depends on external service discovery; heterogeneous registries or metadata standards may reduce applicability.

5. Limited handling of signature mismatches and data mediation — Preference for exact matches; when mediation is required, reuse of fragments is limited and not fully automated.

6. Evaluation scope limited — Prototype experiments do not cover large-scale scenarios; reuse applicability in complex dependency graphs is untested.

D7: Developer-facing challenges affecting reuse

1. Signature dependency and data compatibility — Replacing an operation can break dependent downstream operations; ensuring compatibility is required for reusable fragments.

2. Finding candidate replacements that satisfy functional and non-functional contracts (SLA/QoS) — Reusable artifacts must meet both interface and QoS requirements; balancing these constraints is challenging.

3. Managing grouped operations (1-n, n-1, n-m replacements) — Composing reusable fragments dynamically while preserving semantics and QoS is complex.

4. Spatial correlation and provider-level dependencies — Reusable replacement logic must consider correlated failures and provider-level constraints.

5. Limited metadata and discovery support for reliable reuse — Without rich, standardized metadata, automated discovery and reuse of replacement fragments is fragile.

6. Trade-offs between signature compatibility and QoS compensation — Reusable adaptation strategies must encode how to compensate for QoS changes.

7. Recursive and cascading adaptation effects — Reusable artifacts must be applied safely without causing unpredictable chains of replacements.

D8: Proposed solutions and reusable artifacts

1. Signature‑first matching policy — Prefer candidate replacements that exactly match operation signatures to reduce cascading changes.

2. Dependency analysis (signature and spatial correlation) — Analyze dependencies to determine safe scope for reusable fragments.

3. Support for grouped replacements (1‑n, n‑1, n‑m) — Enable reuse of composite fragments when single replacements are insufficient.

4. Execution model instances and templates — Maintain per-execution model instances and default candidate combinations (STj) to speed reuse of validated replacements.

5. Use of QoS prediction and aggregation — EWMA prediction and aggregation help decide when to reuse a fragment versus compensating elsewhere.

6. Prototype integration with service discovery — Ranked candidate lists allow reusable replacements to be found at runtime, relying on external discovery mechanisms.

D1: Authors
Vincenzo Riccio; Giancarlo Sorrentino; Ettore Zamponi; Matteo Camilli; Raffaela Mirandola; Patrizia Scandurra

D2: Year
2024

D3: Title
RAMSES: an Artifact Exemplar for Engineering Self‑Adaptive Microservice Applications

D4: Venue
SEAMS ’24 (19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems)

D5: Quality Assessment
Total Quality Score: 9 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Stateless-service assumption — RAMSES is designed for stateless microservices; limits reuse of adaptation logic in stateful contexts requiring state transfer or quiescence.

2. Dependence on probing/actuating contract conformance — Reuse requires managed services to implement REST probing/actuating APIs; systems that do not comply need adapters.

3. Prototype evaluation scope — Experiments are preliminary and limited in scale and scenarios; constrains evidence of reuse effectiveness across diverse deployments.

4. Platform and framework coupling — Implementation relies on Spring Boot/Cloud and Docker; reuse on different platforms may require integration effort.

D7: Developer-facing challenges affecting reuse

1. Contract and API conformance burden — Developers must implement Contract-First probing/actuating APIs to enable reuse.

2. Separation of concerns vs. integration friction — Decoupling adaptation logic into MAPE-K microservices aids reuse, but integrating with existing microservice stacks requires careful alignment.

3. Assumption of statelessness limits portability — Reusable adaptation actions (addInstance, changeConfiguration, selectNewImplementation) assume stateless services; stateful reuse is unsupported.

4. Tuning and configuration for different deployments — Reusable system exposes configurable parameters; per-deployment tuning is needed, creating engineering overhead.

5. Dependency on discovery and infrastructure semantics — RAMSES assumes specific discovery/load-balancing semantics (e.g., Eureka); reuse in other environments may require adapters.

D8: Solutions, reusable artifacts, and engineering guidance

1. Autonomic manager as reusable microservices — MAPE-K microservices (Monitor, Analyze, Plan, Execute, Knowledge) with documented REST probing/actuating APIs.

2. Contract-First API specifications — OpenAPI/PlanT UML diagrams for probing/actuating operations that can be adopted by other managed applications.

3. E-food managed microservice application and ScenarioRunner — Reusable benchmark and test harness for experiments and adaptation logic tuning.

4. HMI dashboard and configuration patterns — Exposes control variables and loop parameters for reuse and per-deployment tuning.

5. API-led, Contract-First integration — Standard probing/actuating contracts enable plug-and-play reuse of the autonomic manager.

6. Decoupling via RESTful contracts — Separates adaptation concerns, easing reuse across different managed applications adopting the same contract.

7. ScenarioRunner for reproducible tuning — Mechanism to exercise and tune RAMSES in target environments before production reuse.

8. Configurable control-loop parameters — Monitor periods, window sizes, thresholds exposed as runtime settings to adapt behavior per deployment.

D1: Authors
Martin Pfannemüller; Martin Breitbach; Christian Krupitzer; Markus Weckesser; Christian Becker; Bradley Schmerls; Andy Schürr

D2: Year
2020,

D3: Title
REACT: A Model-Based Runtime Environment for Adapting Communication Systems

D4: Venue
ACSOS 2020 (IEEE International Conference on Autonomic Computing and Self-Organizing Systems)

D5: Quality Assessment
Total Quality Score: 10 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Modeling requirement — Reuse depends on domain experts producing correct Clafer adaptation-options models and UML target-system specifications; lack of modeling skills can

hinder reuse.

2. Mapping effort — Translating problem models (Clafer) to solution models (UML) requires accurate target specifications; mismatches reduce portability of reusable adaptation options.

3. Integration adapters needed — Language-independent sensor/effector interfaces require per-system implementation, adding engineering cost for reuse.

4. Assumed runtime semantics — Planner and strategies assume specific runtime semantics; reuse in systems with different semantics may require rework.

5. Evaluation scope — Only two use cases evaluated; generalizability of reusable artifacts across heterogeneous real-world systems is not fully validated.

6. Stateful adaptation not emphasized — Handling complex stateful migrations or replacements is limited; reuse in stateful domains is constrained.

D7: Developer-facing challenges affecting reuse

1. Bridging heterogeneity with reusable interfaces — Developers must implement adapters for each target system to make MAPE components reusable.

2. Producing and maintaining reusable models (problem/solution separation) — Accurate Clafer/UML models are nontrivial to create; poor models reduce portability.

3. Mapping problem space to solution space reliably — Per-target mapping introduces friction; reusable fragments need templates or adapters.

4. Decentralized deployment and coordination complexity — Distributed deployment introduces coordination concerns; reusable MAPE components must include deployment policies.

5. Developer skill and process overhead — Domain experts must follow development processes and understand modeling choices; reuse is limited otherwise.

6. Assumptions about runtime semantics and action effects — Differing action semantics across platforms reduce safe reuse; artifacts need parameterization or adapters.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Language-independent sensor/effector interfaces (ISensor / IEffector) — Contracts enabling MAPE components to be reused across systems.

2. Problem/solution model separation (Clafer + UML) — Decouples adaptation logic from system specifics; reuse of decision logic is supported by changing only the mapping.

3. Model verification and static analysis (Clafer tooling) — Ensures adaptation options are consistent, increasing confidence in reuse.

4. Prepackaged microservice MAPE-K components — Ready-to-use Monitor/Analyze/Plan/Execute/Knowledge microservices deployable across systems once adapters are implemented.

5. Development process and templates — Provides examples and guided processes (ScenarioRunner, Clafer/UML models) to produce reusable models and mappings.

6. Configurable monitoring and planning strategies — Exposes window sizes, heuristics, and monitoring parameters to adapt reusable artifacts per operational context.

7. Open-source prototype and examples — Implementation and exemplar provide reference integrations for adaptation logic reuse without starting from scratch.

D1: Authors
Elvin Alberts; Ilias Gerostathopoulos; Vincenzo Stoico; Patricia Lago

D2: Year
2024

D3: Title
ReBeT: Architecture-based Self-adaptation of Robotic Systems through Behavior Trees

D4: Venue
2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The Architecture Adaptation Layer (AAL) is designed for reuse but faces integration limitations with existing ROS2 packages.

2. Only reparameterization works by default.

3. Addition/removal requires nodes to be "lifecycle nodes," and changing connections requires implementing specific event handlers in the nodes.

4. These constraints make it difficult to apply these reusable adaptation mechanisms to legacy or third-party ROS2 components without modifying them.

D7: Developer-facing challenges affecting reuse

1. Creating general, reusable mechanisms for architectural adaptation in robotics: Current mechanisms are mostly ad-hoc and not oriented toward reuse. The challenge is to design adaptation logic (like the AAL) that is abstracted from specific applications/missions to be reusable.

2. Integrating quality requirement (QR) consideration and architectural adaptation into existing, widely-adopted robotics practices (ROS2, Behavior Trees): Designing reusable extensions (QRDecorators, AdaptDecorators) that seamlessly plug into standard formalisms without requiring extensive system re-modeling.

3. Designing reusable adaptation artifacts that remain simple and understandable: For Behavior Trees to be reused, their structure should be simple to enhance understandability and reuse. ReBeT's extensions aim to add self-adaptation capability while preserving simplicity.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. QRDecorators: A reusable extension to Behavior Trees for specifying and monitoring QRs, decoupling QR logic from task logic to promote reuse across different missions or systems.

2. Architecture Adaptation Layer (AAL): A reusable ROS2 package that centralizes and standardizes the execution of architectural changes (reparameterization, add/remove, change connections), providing a generic interface for adaptation.

3. Adaptation Decorators (AdaptDecorators): A reusable extension to Behavior Trees that integrates architectural adaptation logic into task plans, allowing the reuse of adaptation strategies across different parts of a BT or across different BTs.

D1: Authors
Kevin Colson; Zhenjiang Hu; Robin Dupuis; Lionel Montrieux; Sebastian Uchitel; Pierre-Yves Schobbens

D2: Year
2016

D3: Title
Reusable Self-Adaptation through Bidirectional Programming

D4: Venue

2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment

Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The approach is focused on the adaptation of configuration files, which limits the kinds of system adaptations it can handle (e.g., architectural changes beyond parameters).

2. The approach requires writing a new bidirectional program (BX) for each target system implementation (and potentially version). While the adaptation logic is reused, the synchronization layer is not.

3. The paper assumes no concurrent modifications to the configuration file between parsing and rewriting, which is a limitation for production systems requiring a more sophisticated synchronizer.

D7: Developer-facing challenges affecting reuse

1. Creating adaptation logic that is independent of the target system's implementation to enable reuse across different implementations or versions: The adaptation layer (analyze and plan phases) must reason over an abstract model that generalizes across implementations. The challenge is defining this model and maintaining its consistency with the concrete, implementation-specific configuration.

2. Correctly synchronizing the concrete, implementation-dependent configuration with the reusable abstract model: Manual implementation of this synchronization is error-prone. Bugs in synchronization can cause drift between the abstract model and the real system, leading to counter-productive adaptation decisions.

3. Handling implementation-specific features (like implicit default values and context overriding in configuration files) within a general, reusable synchronization mechanism: These features complicate the synchronization logic. A reusable synchronization solution must correctly manage them to avoid information loss or corruption when updating the concrete system from the abstract model.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Putback-based bidirectional programming (BiGUL): The developer writes a single put function describing how to update the concrete model given a change in the abstract model. The

complementary get function is automatically derived, and the pair is guaranteed to form a well-behaved bidirectional transformation (BX).

2. Correct-by-construction synchronization: Guarantees the synchronization is correct by construction, eliminating the risk of drift and enabling trust in the reusable adaptation logic operating on the abstract model.

3. Handling configuration file features: BiGUL programs can be written to correctly handle default values and context overriding, addressing key challenges for configuration file synchronization.

D1: Authors
Gustavo Rezende Silva; Juliane Passler; Jeroen Zwanepol; Elvin Alberts; S. Lizeth Tapia Tarifa; Ilias Gerostathopoulos; Einar Broch Johnsen; Carlos Hernández Corbato

D2: Year
2023

D3: Title
SUAVE: An Exemplar for Self-Adaptive Underwater Vehicles

D4: Venue
2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The Metacontrol framework promotes reuse by exploiting a model of the managed subsystem, but the monitor step is mocked up (water visibility and thruster failures are simulated, not probed from a real system), limiting the demonstration of a fully reusable, plug-and-play monitoring interface for real hardware.

2. The exemplar's reusable adaptation logic is tied to the specific functional hierarchy and quality attributes (e.g., generate_search_path, water_visibility) defined in its TOMASys model. Reuse in a fundamentally different domain would require redefining this model.

D7: Developer-facing challenges affecting reuse

1. Creating self-adaptation solutions that are not tied to a single application or robot platform: The separation between application and adaptation concerns allows the adaptation logic to be reused

in different applications, addressing the challenge of tight coupling.

2. Enabling the comparison and benchmarking of different self-adaptation strategies in a consistent, reusable testbed: SUAVE provides a standardized platform to allow fair comparison of adaptation strategies, solving the challenge of lacking reusable testbeds.

3. Integrating reusable adaptation frameworks (like Metacontrol) with established robotics frameworks (like ROS 2) without breaking the existing managed system: The exemplar runs on ROS 2 to ensure the reusable adaptation layer interfaces cleanly with standard robotics stacks.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. SUAVE exemplar: Provides a modular, two-layered architecture that separates the managed subsystem (application) from the managing subsystem (adaptation logic), enabling reuse of the adaptation layer.

2. Metacontrol framework and TOMASys metamodel: Explicitly models functions, design alternatives, and quality attributes, creating a reusable knowledge base for adaptation reasoning.

3. Clear ROS 2 interfaces: Defines /diagnostics topics and mode change services for the monitor and execute steps, allowing any managing subsystem adhering to these interfaces to be plugged in, promoting reuse and comparison.

D1: Authors
Tim Bolender; Gereon Bürvenich; Manuela Dalibor; Bernhard Rumpe; Andreas Wortmann

D2: Year
2021

D3: Title
Self-Adaptive Manufacturing with Digital Twins

D4: Venue
2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The framework's reusability across different CPPSs requires significant re-configuration. Transferring the Digital Twin (DT) to a new machine requires implementing new adapters for communication, updating the domain model, and specifying a new case base and similarity measures. It is not plug-and-play.

2. The modeling languages (CBL, CSL), while extensible, require familiarity with data types, model relations, and potentially PDDL. This can be a barrier for domain experts without technical support, potentially hindering the independent reuse and maintenance of the adaptation knowledge base.

D7: Developer-facing challenges affecting reuse

1. Capturing and formalizing implicit, empirical domain expertise (often based on "gut feeling") into a reusable, machine-processable form for self-adaptation: The challenge is to reify sparse, context-dependent human expertise into explicit cases and similarity rules so it can be reused by the Digital Twin across different production runs and machines.

2. Creating a reusable self-adaptation framework that can be customized for different CPPSs and domains without requiring deep software engineering expertise from the domain expert: The challenge is designing a framework where the core adaptation logic (CBR cycle) is reusable, while the domain-specific knowledge (models) can be supplied by non-programmers to instantiate it for a new system.

3. Balancing autonomy with safety and explainability in a reusable adaptation framework: A reusable solution must be trusted in different operational contexts, acting autonomously while providing comprehensible explanations for its decisions to build trust across scenarios and teams.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Model-driven framework using Domain-Specific Languages (DSLs) and Case-Based Reasoning (CBR): Formalizes tacit domain knowledge for reuse.

2. CBL (Case Base Language) & CSL (Case Similarity Language): Allow domain experts to encode expertise as structured models (cases and similarity metrics), separating reusable adaptation logic from domain-specific models.

3. Modular Digital Twin Architecture: Separates the reusable CBR engine from domain-specific models, allowing adaptation logic to be reused while only the models change for a new CPPS.

4. Model-driven generation: Generates parts of the DT (e.g., data access) from the models, reducing manual coding and promoting reuse of the generation toolchain.

D1: Authors
Liliana Rosa; Luis Rodrigues; Antónia Lopes; Matti Hiltunen; Richard Schlichting

D2: Year
2013

D3: Title
Self-Management of Adaptable Component-Based Applications

D4: Venue
IEEE Transactions on Software Engineering

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The approach assumes system-level Key Performance Indicators (KPIs) can be obtained by monotonically combining component-level KPIs, which may not hold for all system properties or interaction effects.

2. The accuracy of adaptation decisions depends heavily on the quality of the impact functions provided by component developers; these functions are approximations and may be context-dependent.

3. The method requires significant upfront effort from component developers to specify all possible adaptations, their impacts, conflicts, and dependencies.

4. Dynamic creation/destruction of component instances is supported, but if an adaptation's impact depends on the pre-evolution system configuration, multiple adaptation specifications may be needed, increasing complexity.

D7: Developer-facing challenges affecting reuse

1. Knowledge Reuse: The knowledge of a component's adaptations and their impacts is distributed among different developers/teams. The challenge is how to reuse and integrate this decentralized knowledge to make effective system-wide adaptation decisions.

2. Specifying Reusable Adaptation Logic: For reuse across different system contexts, adaptation impact functions must be general enough yet accurate. This is difficult because impacts can depend on dynamic factors like workload and component configuration.

3. Composing Reused Adaptations: Determining how to combine adaptations from different components to achieve a system-wide goal is non-trivial due to potential conflicts and

dependencies. The framework must manage these interactions effectively.

4. Managing Trade-offs for Reuse: When reusing adaptations with known impacts on multiple KPIs, the adaptation manager must solve a multi-objective optimization problem to balance trade-offs according to high-level goals.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Adaptations Specification Language: A structured, declarative language for component developers to describe adaptation actions, requirements, impacts on KPIs, and stabilization periods. This specification becomes the reusable artifact.

2. Automated Two-Phase Rule Generation and Evaluation:

    1. Offline Phase: Analyze adaptation specifications and high-level goal policies to generate candidate adaptation rules, filtering out known conflicts.

    2. Online Phase: Evaluate pre-computed rules against the current system state to select the best set of adaptations while respecting goal rankings and trade-offs.

3. Rank-Eager Evaluation Algorithm: Prioritizes satisfying higher-ranked goals first, enabling graceful degradation and continuous optimization for reused adaptations.

D1: Authors
Wei-Chih Huang; William J. Knottenbelt

D2: Year
2015

D3: Title
Self-Adaptive Containers: Building Resource-Efficient Applications with Low Programmer Overhead

D4: Venue
IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)

D5: Quality Assessment
Total Quality Score: 9 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The reuse method (container library) requires the programmer to explicitly specify Service Level Objectives (SLOs) and operation descriptors for each container instance. This configuration

overhead is a prerequisite for reuse.

2. The adaptation frequency parameter requires careful tuning; too low causes unnecessary adaptation overhead, too high delays adaptations, impacting performance.

3. The library cannot guarantee that all SLOs will be satisfied, especially when they conflict (e.g., memory optimization may violate response time). Developers must manually set appropriate priorities.

4. The approach focuses on a specific domain (data structures); generalizability to other system components is not explored.

D7: Developer-facing challenges affecting reuse

1. High Programmer Overhead in Reuse across Environments: Significant effort is required to manually refactor or adapt software for reuse in each distinct execution environment with its own resource constraints and performance requirements.

2. Managing Conflicting Objectives in Reusable Adaptive Components: Designing reusable components requires mechanisms to resolve trade-offs between multiple non-functional objectives (e.g., response time vs. memory vs. reliability).

3. Specifying Reusable Adaptation Policies: Defining a reusable, declarative, standard-compliant way to specify adaptation goals (SLOs) that can be interpreted by a generic adaptation engine is challenging.

4. Granularity in Functional Reuse: Developers need to specify the exact subset of functionality required from a reusable component, enabling efficient selection of underlying implementations.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Library of Reusable Self-Adaptive Containers: Provides classes (`ICollection`, `IKeyValue`) encapsulating adaptation logic. Reuse in a new environment requires only updating the SLO configuration file.

2. Priority-Based SLO Satisfaction Algorithm: Implements a priority order in the Analyzer to resolve conflicting objectives according to SLO priorities, specified in WSLA format.

3. Operation Descriptors for Granular Reuse: Allow programmers to declare the precise subset of operations required, enabling selection of optimal underlying implementations (e.g., probabilistic data structures).

D1: Authors
Antonio Filieri; Giordano Tamburrelli; Carlo Ghezzi

D2: Year
2016

D3: Title
Supporting Self-Adaptation via Quantitative Verification and Sensitivity Analysis at Run Time

D4: Venue
IEEE Transactions on Software Engineering

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The approach assumes that the variable parameters (used in the precomputed expressions) can be anticipated at design time. (Sections 2, 4)

2. The number of variable parameters must remain a small fraction of total model parameters; otherwise, the approach becomes impractical for runtime reuse. (Sections 2, 4)

3. Nested PCTL formulae require additional parameters, which increases design-time computation complexity and the size of reusable symbolic expressions, limiting applicability to relatively small models. (Sections 5.2.4, 10.5)

D7: Developer-facing challenges affecting reuse

1. High Computational Cost of Reusing Traditional Verification at Runtime: Standard model checking tools are unsuitable for runtime adaptation because they were designed for offline use and cannot meet strict time/memory constraints in SAS. (Sections 1, 2)

2. Balancing Precomputation and Flexibility in Reusable Artifacts: Precomputing symbolic expressions shifts computation to design time, but requires accurate anticipation of which parameters will vary at runtime; incorrect anticipation reduces the system's adaptability. (Sections 2, 4)

3. Managing Complexity of Reusable Models for Full Property Expressiveness: Adapting a single runtime model to verify complex, nested properties introduces complexity, requiring many additional parameters and producing large, expensive-to-evaluate reusable artifacts. (Sections 5.2.4, 10.5)

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Partial Evaluation (Precomputation): Statically analyze system models and requirements at design time to generate symbolic verification conditions, which serve as reusable artifacts for runtime verification. (Sections 2, 5, 6)

2. Efficient Runtime Binding: At runtime, bind monitored values to symbolic parameters in the precomputed expressions and evaluate them, which is computationally inexpensive. (Sections 2, 10.4)

3. Dual-Method Strategy for Precomputation: Provide both matrix-based and equation-based algorithms for generating reusable expressions, allowing selection based on model characteristics and available computational resources. (Sections 5.1.1, 5.1.2, 10.5)

D1: Authors
Michael Stein; Alexander Frömmgen; Roland Kluge; Frank Löffler; Andy Schürr; Alejandro Buchmann; Max Mühlhäuser

D2: Year
2016

D3: Title
TARL: Modeling Topology Adaptations for Networking Applications

D4: Venue
IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. The TARL language cannot express the LMST topology adaptation algorithm because it requires matching paths of arbitrary length, which TARL does not support.

2. The interpreter-based runtime environment (a reusable framework) is infeasible for resource-constrained Wireless Sensor Network (WSN) devices, necessitating a compiler to generate optimized code.

D7: Developer-facing challenges affecting reuse

1. Tight Integration Hindering Reuse: Adaptation logic for network topology is often tightly integrated into application code, making it difficult to understand, compare, and reuse existing

solutions. (Introduction)

2. Lack of a Reusable Abstraction Model: The absence of a well-defined, general model for topology adaptations complicates development and hinders reuse, as there is no common conceptual framework. (Introduction)

3. Generic Rule Languages Lacking Domain-Specific Expressiveness: Existing reusable adaptation languages (e.g., Stitch) do not support graph-based operations needed for topology adaptations, limiting their reusability in networking contexts. (Section 6)

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. General, Reusable Adaptation Model: Decompose topology adaptation into monitoring, decision (graph-pattern-based), and execution steps aligned with MAPE-K, providing a reusable conceptual framework.

2. Domain-Specific Rule Language (TARL): Develop TARL, a reusable, declarative language tailored for expressing graph-based adaptation logic, including pattern matching, joins, and filters.

3. Reusable Runtime Framework: Implement a reusable runtime environment to interpret TARL rules, manage topology providers, and execute adaptations, simplifying integration of adaptation logic into new applications.

D1: Authors
Jürgen Dobaj; Andreas Riel; Thomas Krug; Matthias Seidl; Georg Macher; Markus Egretzberger

D2: Year
2022

D3: Title
Towards Digital Twin-enabled DevOps for CPS providing Architecture-Based Service Adaptation & Verification at Runtime

D4: Venue
International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment
Total Quality Score: 10 / 12

D6: Limitations relevant to reuse of adaptation logic and artifacts

1. The proposed self-adaptive models and DT-enabled approach introduce higher implementation complexity and require more IPC resources (processing, memory, bandwidth) compared to

traditional static CPS designs.

2. Future evaluation is needed to analyze detailed adaptation behavior under uncertain CPS conditions, limiting immediate generalizability and reuse confidence.

D7: Developer-facing challenges affecting reuse

1. Architectural Interoperability for Reusable Components: Designing reusable services and Digital Twin (DT) instances is challenged by high heterogeneity and frequent incompatibility of communication protocols and technologies across different levels of the Automation Pyramid (AP), complicating reusable service development.

2. Designing for Autonomous Adaptation & Deployment: Enabling reuse of adaptation logic (e.g., CI/CD pipelines) requires CPS to autonomously adapt to future changes and deploy to operations space, which is challenging in safety-critical OT environments.

3. Separation of Concerns for Independent Evolution: Independent development and reuse of management vs. operation services requires strict separation of traffic and data flows, with well-controlled interfaces for modular evolution and reuse.

4. Adapting IT Reuse Patterns to OT Constraints: Reusing IT deployment strategies (e.g., A/B testing, Blue/Green) is challenged by OT real-time constraints, interaction with physical assets, hierarchical networks, and limited shared IPC resources.

D8: Solutions, reusable artifacts, and mitigation strategies proposed

1. Use Digital Twins (DTs) as a reusable knowledge base across the DevOps lifecycle; layered Dev and Ops DT instances with adaptive fidelity facilitate context-awareness and coordinated adaptation.

2. Adopt a modular, microservice-based architecture for Industrial IoT to support independent service development, deployment, and reuse, aligning with separation of concerns.

3. Apply hierarchical control and coordination patterns from decentralized self-adaptive systems to manage adaptation of reusable services and DTs across heterogeneous AP levels.

4. Introduce a generic, layered device design model separating OT and IT domains with a mediator, providing a blueprint for reusable probes, effectors, and DT instances tailored to specific CPS devices.

D1: Authors
Narges Khakpour; Charilaos Skandylas; Goran Saman Nariman; Danny Weyns

D2: Year
2019

D3: Title
Towards Secure Architecture-based Adaptations

D4: Venue
IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and artifacts

1. Computational overhead: Generating attack graphs and calculating security metrics for each potential configuration (including transients) has complexity O(nm2logm)O(n m^2 \log m)O(nm2logm), impacting real-time adaptation decisions in large or complex systems.

2. Dependency on vulnerability specifications: The approach requires augmenting the architectural model with known vulnerabilities. For external or black-box components, this relies on external scanners (e.g., OpenVAS), which may not be complete or accurate, introducing uncertainty.

D7: Developer-facing challenges affecting reuse

1. Integrating security analysis into reusable frameworks: Extending a reusable architecture-based adaptation framework (like Rainbow) to securely reuse adaptation tactics and strategies is challenging because each composition of adaptation blocks can introduce unforeseen vulnerabilities.

2. Assessing transient configurations: Reusing adaptation logic often moves the system through intermediate architectural states. Evaluating security risk for the entire adaptation path—not just the final configuration—is complex, as vulnerabilities may be exposed during transitions.

3. Defining reusable security metrics: To reuse utility-driven adaptation strategies safely, one must define quantifiable, reusable security metrics applicable to any configuration generated by the reusable adaptation logic. Translating attack graph properties into a consistent "security utility" score is non-trivial.

4. Automating vulnerability modeling for reusable components: Capturing vulnerabilities for both custom and third-party components in a reusable architectural knowledge base is challenging,

requiring automated, generic model transformations.

D8: Suggested Solutions

1. Extend the MAPE-K loop with a security analysis phase, creating a reusable pattern to integrate security assessment into any architecture-based SAS.

2. Introduce formal, graph-based security metrics (e.g., PMPL, PAGP) that aggregate risk across all configurations along an adaptation path, applicable for evaluating reusable adaptation strategies.

3. Implement a model transformer that converts runtime architectural models and vulnerability specifications into a formal attack model (Horn clauses), automating security analysis for reusable components.

4. Integrate security analysis directly into Rainbow as a new "security dimension" for utility calculation, demonstrating how a reusable SAS framework can be retrofitted to include reusable security-aware adaptation.

D1: Authors
Christian Krupitzer; Felix Maximilian Roth; Sebastian VanSyckel; Christian Becker

D2: Year
2015

D3: Title
Towards Reusability in Autonomic Computing

D4: Venue
IEEE 12th International Conference on Autonomic Computing (ICAC)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and artifacts

1. Loading the functional logic into reusable components is initially hard-coded using if/else structures, limiting dynamic replacement.

2. Integration of self-* property patterns is restricted to communication/distribution patterns in configuration files; more sophisticated pattern support is needed.

3. The generic context manager requires minor customization for each use case.

4. The reusable component template and communication mechanism are evaluated in a specific ecosystem (BASE middleware, Java), which may limit direct reuse in other technology stacks.

D7: Developer-facing challenges affecting reuse

1. Lack of fine-granular reusability in existing frameworks: Existing frameworks (e.g., Rainbow, SASSY) reuse components only at the architectural level, neglecting internal component logic, forcing repeated low-level implementation.

2. Difficulty in exchanging component behavior without full replacement: Swapping core functional logic often requires replacing the entire component, preventing reuse of stable infrastructure.

3. Tight coupling between functional logic and component infrastructure: Communication, data handling, and MAPE-K integration are tightly coupled to custom logic, impeding independent reuse and evolution.

4. Designing for decentralized communication and coordination: Building a reusable communication architecture that decouples components across heterogeneous, distributed SASs is complex.

D8: Suggested Solutions

1. Reusable Adaptation Logic (AL) Template: Based on MAPE-K and enhanced with a context manager to abstract low-level sensor/effector data, increasing reusability across SAS use cases.

2. Reusable Component Template: Uses the Template Method pattern to separate custom functional logic (exchangeable) from generic infrastructure logic (communication/data handling), enabling independent swapping.

3. Reusable topic-based Publish/Subscribe architecture: Decentralized, standardized information categories/types (e.g., MONITORING, ANALYZING) support reusable subscription and communication patterns.

4. XML-based configuration files: Define deployment, component distribution, and communication structures, largely reusable with minor adjustments for functional logic identifiers and data types.

D1: Authors

Thomas Vogel

D2: Year
2018

D3: Title
mRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization

D4: Venue
ACM/IEEE 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and artifacts

1. None explicitly stated for the mRUBiS exemplar itself. The paper positions it as a solution to limitations in other reuse methods.

D7: Developer-facing challenges affecting reuse

1. High cost and complexity of implementing runtime models: Developing a self-adaptive system with a model-based approach requires implementing and maintaining an architectural runtime model and its causal connection to the adaptable software. This involves addressing the abstraction gap, ensuring model fidelity, and achieving efficient synchronization, which is complex, error-prone, and costly, diverting effort from core adaptation logic.

2. Lack of reusable exemplars with pre-built models: Existing exemplars and frameworks for self-adaptive systems typically provide only APIs, not reusable runtime models. This forces developers to reimplement foundational models and synchronization infrastructure for each new system, severely hindering reuse and rapid development of model-based adaptation engines.

D8: Suggested Solutions

1. Provide a reusable exemplar (mRUBiS) that includes a simulator and a maintained architectural runtime model, eliminating the need for developers to build the model and causal connection from scratch.

2. Introduce a generic modeling language (CompArch) for representing component-based architectures and runtime properties, usable as a standard interface for adaptation engines.

3. Design a simulation framework that injects issues, validates adaptations, computes utility, and measures performance, providing a reusable testbed for developing and evaluating adaptation logic.

Scopus Articles

D1: Authors
Shengpu Liu; Liang Cheng

D2: Year
2011

D3: Title
A context-aware reflective middleware framework for distributed real-time and embedded systems

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations of Reuse Method

1. The component model currently only supports COM components and .NET assemblies. Supporting other component types (e.g., CORBA, Java Beans) requires the development of special component wrappers.

2. The framework and its synchronization protocol are designed for stateless applications. Supporting stateful applications requires integration with additional techniques like state-machine or model-based reconfiguration.

D7: Developer-facing challenges affecting reuse

1. Performance-related challenge rather than reuse-specific: The paper does not explicitly report challenges related to designing or developing self-adaptive systems with reusable components. The primary challenge addressed is performance: the long reconfiguration time (seconds) of existing adaptive middleware frameworks is inadequate for real-time systems.

2. Implicit reuse considerations: While the framework uses reusable components (marchlets, marchtools), the paper does not discuss challenges in achieving that reuse, such as component discovery, integration, or variability management.

D8: Suggested Solutions

1. Proposes the MARCHES framework using a multiple component-chain architecture to enable fast local behavior changes by switching between pre-built chains instead of modifying a single chain.

2. Introduces an active-message-based asynchronous synchronization protocol to coordinate distributed behavior changes without blocking operations, eliminating buffer clearance delays.

3. Provides a reflective middleware model with component-level and system-level reflection to incorporate standard components and reconfigure connections at runtime.

4. Uses an XML-based script language to declaratively describe and manage adaptation policies and system composition.

D1: Authors
Jaegoon Lee; Dirk Muthig; Matthias Naab

D2: Year
2010

D3: Title
A feature-oriented approach for developing reusable product line assets of service-based systems

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations of Reuse Method

1. The granularity of reusable services ("molecular services") is domain-specific and requires professional judgment to identify correctly.

2. The approach assumes the developed assets are under the control of a single organization, not third-party service providers.

3. The method and architecture focus on a specific class of systems (those whose variations are best described by workflow variations).

D7: Developer-facing challenges affecting reuse

1. Identifying reusable services at the right granularity: Service Orientation facilitates easy composition, but this leads to a proliferation of similar, fine-grained services, making it challenging to determine which groupings of functionality constitute a reusable, self-contained

service unit.

2. Managing dynamic configuration based on context: Service configurations must be determined and bound at runtime based on each user's current needs, role, and available technical resources, requiring mechanisms to map contextual situations to specific service configurations.

3. Maintaining service validity after dynamic reconfiguration: Runtime reconfiguration necessitates verifying that the currently active services remain valid and correct for the new configuration and user context (e.g., checking pre-/post-conditions and invariants).

D8: Suggested Solutions

1. Proposes a feature-oriented product line engineering approach where feature binding analysis identifies candidate reusable service units ("feature binding units") and their binding times.

2. Introduces a service classification into *orchestrating services* (workflow/behavior) and *molecular services* (computational/task) to achieve the right reuse granularity. Molecular services become the core reusable assets.

3. Provides a context analysis and specification method to define contextual parameters and situations, enabling runtime mapping of context to specific service configurations and dynamic reconfiguration triggers.

4. Proposes the HEART (Heterogeneous style based ARchiTecture) model, a three-layer architecture combining Service-Oriented, Communicating Process, and C2 styles to support late binding, mobility, and dynamic reconfiguration of service bricks (orchestrating and molecular).

5. Extends workflow/service specifications with pre-/post-conditions and invariants (based on Design by Contract) to allow automatic checking of service validity during composition and reconfiguration.

D1: Authors

Nadeem Abbas, Jesper Andersson, Danny Weyns

D2: Year

2020

D3: Title

ASPLe: A methodology to develop self-adaptive software systems with systematic reuse

D4: Venue

The Journal of Systems and Software

D5: Quality Assessment

Total Quality Score: 12 / 12

D6: Limitations (specifically regarding reuse of adaptation logic)

1. Internal adaptation mechanisms (mixing application and adaptation logic) lead to poor maintainability and reusability.

2. Existing external adaptation frameworks (e.g., Rainbow, MADAM) offer limited or no systematic process support for developing SASS with reuse.

3. Existing reuse strategies and practices are not designed to handle the uncertainty inherent in self-adaptive systems, arising from lack of knowledge about target domains and runtime variability.

D7: Challenges (developers + reuse)

1. Lack of systematic process support for reuse: There is a lack of documented, repeatable methodologies with defined roles, activities, and work-products to systematically reuse knowledge across SASS projects, hindering knowledge transfer and quality design.

2. Managing uncertainty in a reuse context: Combining software reuse with runtime adaptation creates complexity due to uncertainty, including:
   2.1. Lack of knowledge when developing domain-independent platforms.
   2.2. Runtime variability in managed systems that is hard to anticipate at design time.

3. Achieving effective separation of concerns for reusability: Applying the separation of managed and managing subsystems systematically to create reusable assets is challenging and lacks process guidance.

D8: Proposed Solutions

1. ASPLe methodology: Comprises three core processes:
   1.1. Domain Engineering – builds a horizontal, domain-independent platform for managing

subsystems.
1.2. Specialization – derives domain-specific platforms.
1.3. Integration – aligns and integrates managing and managed system platforms.

2. Extended Architectural Reasoning Framework (eARF): Provides reusable architectural knowledge (tactics, patterns like MAPE-K) and work-products (`dQAS`, `dRS`) for modeling requirements and design with explicit variability and uncertainty management.

3. Separation of concerns: Ensures independent development and reuse of adaptation logic.

4. Stepwise refinement and specialization: Gradually resolves uncertainty and tailors reusable assets to specific application domains.

D1: Authors
Frederico Alvares; Eric Rutten; Lionel Seinturier

D2: Year
2017

D3: Title
A domain-specific language for the control of self-adaptive component-based architecture

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. **Combinatorial complexity of Discrete Controller Synthesis (DCS):** State-space explosion limits scalability as the number of system configurations grows, restricting reuse in large or highly configurable architectures.

2. **Requirement for complete knowledge of configurations:** Off-line controller synthesis assumes that all possible system configurations are known at design time, limiting reuse in open, evolving, or unpredictable environments.

3. **Strong dependency on formal models and middleware:** The approach relies on formal behavioral models and integration with the FraSCAti platform and Heptagon/BZR toolchain, reducing portability and reuse across heterogeneous architectural frameworks.

D7: Reported developer-facing challenges (affecting reuse)

1. **Scalability of formal adaptation logic:** Applying reusable formally verified controllers becomes computationally expensive as architectural variability increases.

2. **Managing behavioral specification complexity:** Developers must author high-level behavioral programs that remain correct across many configurations, which is cognitively demanding and error-prone.

3. **Guarantee–flexibility trade-off:** Achieving strong correctness guarantees through off-line synthesis constrains runtime adaptability and limits reuse in systems with unforeseen configurations.

D8: Proposed solutions, artifacts, and reusable elements

1. **Off-line Discrete Controller Synthesis:** Correct-by-construction controllers are generated at design time, enabling lightweight and reactive runtime adaptation with guaranteed properties.

2. **Ctrl-F domain-specific language:** Provides reusable abstractions for expressing configurations, policies, and behavioral programs while hiding low-level reconfiguration actions.

3. **Modular DCS (future research direction):** Decomposes the global control problem into smaller units to reduce state-space explosion and improve scalability and reuse.

D1: Authors
Juliane Päßler; Maurice H. ter Beek; Ferruccio Damiani; Einar Broch Johnsen; S. Lizeth Tapia Tarifa

D2: Year
2025

D3: Title
Analyzing Self-Adaptive Systems as Software Product Lines

D4: Venue
The Journal of Systems & Software

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. **Bounded adaptivity assumption:** The DSPL/FTS approach supports only bounded adaptivity, where all relevant context variations must be anticipated at design time, limiting reuse in open or evolving environments.

2. **Probabilistic modelling requirement:** Reuse requires probabilistic modelling of environment behavior and uncertainty, which demands estimates or empirical data and may be difficult to transfer across domains.

3. **Abstraction limitations:** Finite-state and feature-based abstractions may fail to capture continuous, hybrid, or highly complex real-world dynamics, reducing fidelity when reused in different application contexts.

4. **Scalability constraints:** Family-based probabilistic models may suffer from state-space explosion as variability and feature combinations grow, limiting reuse for large-scale systems.

D7: Reported developer-facing challenges (affecting reuse)

1. **Managing runtime variability:** Managing runtime variability in SAS is explicitly identified as a difficult and still unresolved challenge, even when structured reuse approaches such as DSPLs are applied.

2. **Designing correct adaptation logic:** Developers must correctly implement and track all adaptation rules and valid configurations; omissions or oversights can easily introduce errors in reusable adaptation logic.

3. **Scalability of design and analysis:** Formal, family-based modelling and analysis techniques face scalability limitations as system complexity increases, restricting practical reuse.

4. **Design for unknown contexts:** Supporting open adaptivity, where context variations are not known at design time, remains a major challenge and is not addressed by the bounded DSPL approach.

5. **Integration of adaptation and application logic:** Cleanly separating and coordinating the managed subsystem (features/application logic) and managing subsystem (adaptation controller) within a single reusable formal model is non-trivial.

D8: Proposed solutions, artifacts, and reusable elements

1. **Modelling SAS as DSPLs:** The managed subsystem is modelled as a family of systems (e.g., Probabilistic Featured Transition Systems), while the managing subsystem is modelled as a feature controller, enabling structured reuse of adaptation logic.

2. **Family-based analysis:** Existing SPL analysis techniques and tools (e.g., ProFeat) are reused to analyze all configurations and reconfigurations in a single run, supporting verification of safety, rewards, and adaptation correctness.

3. **Separation of concerns via modelling layers:** A layered modelling framework enforces separation between feature models, behavioural models, and controllers, improving modularity and reuse of adaptation assets.

4. **Probabilistic reasoning under uncertainty:** Probabilistic modelling of environments and failures enables reusable analysis of adaptation logic under uncertainty.

D1: Authors
Kavan Sedighiani; Saeed Shokrollahi; Fereidoon Shams

D2: Year
2021

D3: Title
BASBA: A framework for Building Adaptable Service-Based Applications

D4: Venue
The Journal of Systems & Software

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. **Limited adaptation tactic coverage:** The reuse capability of BASBA is constrained by the currently available set of predefined adaptation tactics, limiting applicability to scenarios outside this library.

2. **Insufficient support for complex or domain-specific adaptations:** Adaptation needs involving complex business logic or strong domain specificity are not efficiently supported, reducing reuse potential in advanced service-based systems.

3. **Reduced usefulness for professional developers:** Empirical results show that professional developers derive less benefit from BASBA in highly complex scenarios, limiting reuse across developer profiles.

4. **Inefficiency in some adaptation scenarios:** In certain cases, defining adaptation behavior using the BASBA adaptive process model is less efficient than direct implementation, reducing incentives for reuse.

5. **Incomplete adaptation plan identification:** BASBA does not comprehensively identify all adaptation needs, as demonstrated by case studies where a significant portion of adaptation plans were specified without the framework.

D7: Reported developer-facing challenges (affecting reuse)

1. **Separation of concerns:** Adaptation logic is often interwoven with application logic, increasing complexity, reducing maintainability, and hindering systematic reuse.

2. **Lack of systematic reuse mechanisms:** Existing approaches fail to provide generic, repeatable, and reusable mechanisms to describe and perform adaptation at the service collaboration level.

3. **Limited adaptation expressiveness:** Many approaches support only a restricted set of adaptation patterns, which is insufficient for highly dynamic environments requiring frequent change.

4. **Difficulty in reusing adaptation knowledge:** Explicit management and reuse of adaptation knowledge (e.g., tactics and strategies) throughout the development lifecycle remains challenging.

D8: Proposed solutions, artifacts, and reusable elements

1. **BASBA framework:** A model-based framework that separates adaptation concerns by deriving runtime adaptation models from design-time specifications.

2. **Reusable adaptation tactics:** A library of predefined adaptation tactics categorized into process, activity, and communication variations, enabling reuse of common adaptation strategies.

3. **Metamodel and adaptation notation:** A dedicated metamodel and notation for specifying adaptive process models, quality requirements, and variation points in a reusable manner.

4. **Automated model-to-runtime transformation:** Transformation rules that automatically generate runtime adaptation artifacts and execute adaptation logic via a MAPE-K feedback loop.

D1: Authors
Javier Cámara; Pedro Correia; Rogério de Lemos; David Garlan; Pedro Gomes; Bradley Schmerl; Rafael Ventura

D2: Year
2016

D3: Title
Incorporating architecture-based self-adaptation into an adaptive industrial software system

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment

Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. **High upfront integration cost:** Integrating an architecture-based self-adaptation (ABSA) framework into a legacy industrial system requires significant initial effort, including removal or coexistence management of existing adaptation mechanisms and development of translation infrastructure.

2. **Static monitoring infrastructure:** Probes and gauges must be defined at development time, limiting the ability to reuse the framework for adaptations that would require dynamic changes to monitoring at runtime.

3. **Selective applicability of ABSA:** Not all adaptation concerns are suitable for management at the architectural level; some are better handled by low-level embedded mechanisms, limiting the scope of reusable adaptation logic.

4. **Dependency on source code access and refactoring:** Effective reuse of the ABSA framework requires access to legacy system source code and substantial refactoring to expose internal state and effectors when suitable interfaces are not already available.

D7: Reported developer-facing challenges (affecting reuse)

1. **High integration effort:** Introducing a reusable ABSA framework into an existing, complex industrial system requires considerable effort to disable scattered legacy adaptation logic and implement the necessary probes and effectors.

2. **Scattered legacy adaptation logic:** Adaptation mechanisms in legacy systems are often embedded across multiple components and intertwined with business logic, making extraction and replacement with reusable frameworks difficult.

3. **Limited access to global system information:** Embedded adaptation mechanisms rely on localized, low-level indicators; reusing an ABSA framework requires exposing and modelling higher-level, architectural system properties.

4. **Exposing system internals:** Developers must instrument and refactor legacy code to expose internal state and control points, which is challenging if the system was not designed for introspection.

5. **Determining adaptation scope:** A key challenge is deciding which adaptation concerns should be handled by the reusable ABSA layer and which should remain at lower levels to balance flexibility, complexity, and performance.

6. **Inflexible monitoring infrastructure:** Static monitoring limits the reuse of ABSA frameworks for adaptation scenarios that require runtime changes in what is observed.

D8: Proposed solutions, artifacts, and reusable elements

1. **Reusable ABSA framework:** Use frameworks such as Rainbow that provide reusable MAPE-K infrastructure with explicit customization points for architectural models, constraints, strategies, and effectors.

2. **Centralization of adaptation logic:** Decouple adaptation logic from application code by placing it in an external control layer, making it explicit, easier to evolve, and reusable across system variants.

3. **Runtime architectural models:** Employ architectural runtime models as a shared knowledge base, enabling adaptation decisions based on a global, system-level view rather than localized indicators.

4. **High-level adaptation languages:** Specify adaptation strategies and tactics in dedicated languages (e.g., Stitch) to improve clarity, maintainability, and reuse compared to code-level adaptations.

5. **Translation infrastructure investment:** Develop probes and effectors as a one-time integration investment to enable observation and control by the adaptation framework.

6. **Strategic separation of concerns:** Allocate adaptation responsibilities deliberately, using the ABSA layer for global, goal-driven adaptations and retaining low-level mechanisms for fast, localized responses.

D1: Authors
Arjan de Roo; Hasan Sözer; Lodewijk Bergmans; Mehmet Aksit

D2: Year
2013

D3: Title
MOO: An architectural framework for runtime optimization of multiple system objectives in embedded control software

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. **Static architectural view:** The MOO architectural style captures only a static representation of the optimization process, limiting reuse in systems requiring dynamic runtime adaptation.

2. **Lack of probabilistic constraint support:** Relationships between decision variables that are uncertain or probabilistic cannot currently be expressed, restricting the applicability and reuse of the adaptation logic in stochastic or partially known environments.

D7: Reported developer-facing challenges (affecting reuse)

1. **Implicit, non-modular implementation:** Optimization logic is typically tightly integrated with the core control software, making it difficult to extract, analyze, and reuse across different systems.

2. **Lack of architectural abstraction for reuse:** Decision variables, constraints, and objective functions are not explicitly represented in the architecture, hindering communication, analysis, and reuse of adaptation logic.

3. **Crosscutting concerns hindering reuse:** MOO logic spans multiple components, creating implicit dependencies that complicate modification and impede the reuse and evolution of both adaptation logic and core software.

4. **High cost of reuse and evolution:** Ad hoc, tightly coupled implementations increase development and maintenance costs, often discouraging the reuse of optimization capabilities.

D8: Proposed solutions, artifacts, and reusable elements

1. **Architectural framework for explicit modeling:** Introduce a MOO architectural style that separates adaptation concerns from core control logic, making decision variables, constraints, and objectives explicit and reusable.

2. **Automated generation and integration:** Use a toolchain to generate optimized adaptation logic (optimizer modules) from architectural models and integrate them into core software via aspect-oriented techniques, enhancing modularity and reuse.

3. **Separation of concerns with DSMLs:** Model physical and computational aspects using Domain-Specific Modeling Languages (e.g., SIDOPS+) separately from core implementation, allowing independent reuse and maintenance of each artifact.

D1: Authors
Andre Felipe Monteiro; Marcus Vinicius Azevedo; Alexandre Sztajnberg

D2: Year
2013

D3: Title
Virtualized Web server cluster self-configuration to optimize resource and power use

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment
Total Quality Score: 11 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. **Platform-specific API dependency:** The HVMM API relies on correcting and enhancing the underlying Xen API, creating integration effort and limiting straightforward reuse on other virtualization platforms.

2. **Fixed VM configuration:** The architecture assumes a fixed number of pre-configured Virtual Machines (VMs), which restricts reuse in scenarios requiring dynamic scaling or VM creation/deletion.

3. **Environment-specific tuning:** Policy parameters and neural network training are calibrated based on preliminary tests of the specific cluster, meaning adaptation logic must be reconfigured or retrained for different environments.

D7: Reported developer-facing challenges (affecting reuse)

1. **API dependency and heterogeneity:** Vendor-specific virtualization APIs create implementation difficulties in heterogeneous environments, complicating the development of reusable adaptation logic.

2. **Designing platform-independent abstractions:** Creating a comprehensive, reusable API abstraction that works across different virtualization platforms is challenging due to incomplete or inconsistent support in underlying vendor APIs.

D8: Proposed solutions, artifacts, and reusable elements

1. **HVMM high-level API:** A reusable API layer (Host and Virtual-Machine Manager) encapsulates vendor-specific operations, providing generic functions for the adaptation logic (Controller,

Actuator, Collector) and making the self-adaptive architecture largely platform-independent.

D1: Authors
Luxi Chen; Linpeng Huang; Chen Li; Xiwen Wu

D2: Year
2017

D3: Title
Self-adaptive architecture evolution with model checking: A software cybernetics approach

D4: Venue
The Journal of Systems and Software

D5: Quality Assessment
Total Quality Score: 12 / 12

D6: Limitations relevant to reuse of adaptation logic and assets

1. Not applicable — the paper does not propose or evaluate a software reuse method; its focus is on validating and evolving adaptation rules within an architectural model.

D7: Reported developer-facing challenges (affecting reuse)

1. No challenges related to the design and development of SAS with reuse are reported. The focus is on validating and correcting adaptation logic within a single architectural specification, rather than on reuse of components, patterns, or adaptation knowledge across systems.

D8: Proposed solutions, artifacts, and reusable elements

1. Not applicable — the suggested solutions (software cybernetics framework using model checking and a learning algorithm) address validation and evolution of adaptation rules for correctness, not reuse challenges.