The Journal of Systems and Software 162 (2020) 110497

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Providentia: Using search-based heuristics to optimize satisficement and competing concerns between functional and non-functional objectives in self-adaptive systems

Kate M. Bowers[a,*], Erik M. Fredericks[a], Reihaneh H. Hariri[a], Betty H. C. Cheng[b]

[a] *Oakland University, Rochester, MI, USA*
[b] *Michigan State University, East Lansing, MI, USA*

A R T I C L E I N F O

Article history:
Received 1 May 2019
Revised 30 November 2019
Accepted 13 December 2019
Available online 18 December 2019

Keywords:
Search-based software engineering
Non-functional requirements
Self-adaptive systems
Evolutionary computation
Optimization

A B S T R A C T

In general, a system may be subject to a combination of functional requirements (FRs) that dictate behavior and non-functional requirements (NFRs) that characterize *how* FRs are to be satisfied. NFRs also introduce cross-cutting concerns that may be difficult to predict, where the degree of satisfaction (i.e., satisficement) of one NFR may be impacted by the satisficement of one or more FRs/NFRs. In particular, self-adaptive systems (SASs) can modify system configurations or behaviors at run time to continuously satisfy FRs and NFRs. This paper presents `Providentia`, a search-based technique to optimize the satisficement of NFRs in an SAS experiencing various sources of uncertainty. `Providentia` explores different combinations of weighted FRs to maximize NFR/FR satisficement. Experimental results suggest that `Providentia`-optimized goal models significantly improve the satisficement of an SAS when compared with manually- and randomly-generated weights and subgoals. Additionally, we apply a hyper-heuristic (`Providentia-SAW`) to balance the contribution of NFRs, FRs, and the number of adaptations and further improve the `Providentia` technique. We apply `Providentia` and `Providentia-SAW` to two case studies in different application domains involving a remote data mirroring network and a robotic vacuum controller, respectively.

1. Introduction

A self-adaptive system (SAS) provides adaptation strategies for performing reconfigurations at run time to address unexpected issues that arise as a result of uncertainty (e.g., adverse environmental conditions or unanticipated issues in the system itself) (Kephart and Chess, 2003; McKinley et al., 2004). For example, a smart vacuum can be modeled as an SAS, where a reconfiguration is modeled as updating the cleaning path or navigation strategy as the vacuum encounters an obstacle, such as a chair. Each reconfiguration performed by the system can incur a cost (e.g., computation time, memory resources, etc.) associated with initializing and performing the adaptation. The SAS will use these adaptation strategies to select an optimal configuration that enables requirements to be continuously satisficed (i.e., degree of satisfaction) (Chung et al., 2000). Generally, an SAS is governed by functional requirements (FRs) that focus on a specific function or feature of the system and can be mathematically quantified to monitor satisficement (van Lamsweerde, 2009). FRs in a smart vacuum may include maintaining battery power above 5% or avoiding obstacles detected by a sensor. Introducing non-functional requirements (NFRs) makes the adaptation selection process more difficult as NFRs specify properties and/or characteristics about system operations, tend to be qualitative, and may not be easily mathematically quantifiable (e.g., specifying resiliency and efficiency) (van Lamsweerde, 2009; Yrjönen and Merilinna, 2009). An example NFR for a smart vacuum SAS may be to optimize performance by cleaning as much dirt as quickly as possible, in contrast to an FR that mandates the vacuum to clean at least 50% of the room. Quantifying NFRs often relies on domain knowledge and may not be optimal given the changing environmental conditions that an SAS must address (Yrjönen and Merilinna, 2009). Therefore, this paper describes `Providentia` and `Providentia-SAW`, search-based techniques performed at design time that automatically determine an optimal set of FRs to support each NFR in an SAS.[1]

* Corresponding author.
E-mail address: kmlabell@oakland.edu (K.M. Bowers).

[1] A preliminary version of `Providentia` was presented at SSBSE 2018 (Bowers et al., 2018).

https://doi.org/10.1016/j.jss.2019.110497
0164-1212/© 2019 Elsevier Inc. All rights reserved.

Current techniques to satisfy NFRs in SASs do not offer concrete numerical values to be evaluated at run time. In the KAOS goal modeling framework, NFRs are incorporated as behavioral or soft goals (van Lamsweerde, 2009). A KAOS soft goal describes preferences of system behaviors that tend to be qualitative in nature, thereby making the determination of an optimal reconfiguration strategy more challenging (van Lamsweerde, 2009). In contrast, KAOS FRs can be quantified via utility functions and provide a concrete numerical basis for comparisons between reconfiguration strategies (Chung et al., 2000). NFRs in the iStar framework are also modeled as soft goals and use the $++/+$ or $--/-$ operators to respectively indicate that an NFR makes/helps or breaks/hurts an FR (Yu, 1997). The iStar operators are also qualitative and can be challenging to use in an SAS, as qualitative descriptions are not necessarily as easy to use when making an adaptation compared to quantitative descriptions. Similar to Providentia, the Analytic Hierarchy Process (AHP) decomposes NFRs into one or more weighted FRs using a prioritization schema (Salehie and Tahvildari, 2012). However, prioritizations in an SAS may change drastically or even be inapplicable at a given instance in time (e.g., most or all requirements have equal priority) as the system experiences various forms of uncertainty due to changing environmental and system conditions. In a smart vacuum SAS, environmental uncertainty can be described as the positioning of obstacles in a room, stairs or other changes in elevation, the amount of dirt to be cleaned, and the number of water puddles to avoid. Uncertainty with regard to system conditions relates to sensor failures, damage that occurs in an obstacle collision, and motor degradation in the wheels. Each of these conditions are subject to change and the vacuum must be able to satisfy its requirements for each combination of uncertainties.

This paper describes Providentia (Bowers et al., 2018) and Providentia-SAW (an extension), two techniques that address the challenges of quantifying and analyzing NFRs at run time in SASs. We introduce Providentia within the context of SASs to minimize the number of reconfigurations performed and achieve optimal weighted combinations that maximize requirement satisficement. Non-SASs can also apply Providentia to obtain optimal weighted combinations for each NFR given that the utility functions are adjusted accordingly. However, the effectiveness of Providentia is limited in non-SASs as the system configurations are static and cannot be reconfigured at run time. Therefore, we limit our scope to SASs. Providentia is a design-time technique that takes into account uncertainty from the environment and the system itself and optimizes FR/NFR relationships, where each relationship contributes to quantifying NFR objectives at run time. Each FR is associated with a utility function that specifies a mathematical expression of requirement satisficement (Chung et al., 2000). Each NFR comprises a combination of one or more FRs using a linear-weighted sum to indicate the relative impact that an FR has in contributing to the satisficement of the NFR's objectives (Salehie and Tahvildari, 2012). For example, the NFR to maximize performance uses four FRs (i.e., Achieve 50% clean, Achieve cleaning efficiency, Achieve cleaning effectiveness, and Maintain safety) to represent performance objectives that might otherwise be more difficult to quantify. Providentia explores different combinations of weights at design time to find an optimal linear-weighted expression that makes the system more robust to various forms of uncertainty at run time. Providentia-SAW, in contrast, is a hyper-heuristic (Burke et al., 2003) approach to adjust the weights of the linear-weighted sum to respond to changing environmental conditions.

Providentia is a search-based evolutionary technique that assesses the system's run-time behavior via an executable system specification that is subjected to randomly-generated sources of uncertainty. The search process identifies optimal goal model configurations, namely the set of FRs and their corresponding weights for a given NFR, to maximize FR/NFR satisficement. Providentia uses a genetic algorithm (Holland, 1992) as a search heuristic, where the search space is the weight of each FR set for a given NFR, and the output is a set of optimal weight assignments that results in the highest satisficement of the NFR when faced with uncertainty. The optimal weight assignments determined by Providentia are then applied to the SAS at run time. By evaluating traditionally soft goals with FR metrics during execution, the SAS is able to perform online reconfigurations in response to both NFR and FR objectives, where traditionally only FR objectives are mainly considered. Furthermore, the SAS can perform better at run time by optimizing the weighted contributions of FRs to each NFR, as a requirements engineer may not be able to foresee the impact of random sources of uncertainty when determining the weight assignments at design time.

Since it may be difficult to achieve an optimal weighting scheme between NFRs, FRs, and the number of adaptations, this paper extends Providentia Bowers et al. (2018) by introducing a stepwise adaptation of weights (SAW) hyper-heuristic (Craenen and Eiben, 2001; Eiben and van der Hauw, 1997; 1998) to optimize the overall fitness value of the SAS for a given set of environmental conditions. Rather than manually selecting weights for the overall fitness function of the system, either according to preference or empirical evidence, Providentia-SAW more accurately determines an optimal set of weights that better guide Providentia's search procedure. As Providentia determines the optimal set of FRs and the weighting scheme for each NFR, Providentia-SAW runs in tandem to determine the weights assigned to balance NFR fitness values (i.e., satisficement), FR fitness values, and the number of adaptations to yield an optimal overall fitness value. Note, the term *weights* for Providentia apply to the NFR utility functions, whereas Providentia-SAW *weights* apply to the fitness sub-functions that comprise the Providentia genetic algorithm.

We illustrate the effectiveness and domain independence of Providentia and Providentia-SAW with two case studies: a remote data mirroring (RDM) network and an intelligent robotic vacuum. The RDM is an industry-provided application that replicates and disseminates messages to each RDM within the network (Ji et al., 2003; Keeton et al., 2004). The RDM performs dynamic reconfigurations in response to uncertainty due to dropped or delayed messages, sensor noise, and unexpected server and network link failures. Results from our preliminary work (Bowers et al., 2018) have shown that Providentia-optimized goal models result in significantly higher fitness values compared to goal models with manually- and randomly-assigned FR weights. Furthermore, results also indicated that the number of FR violations was significantly reduced when Providentia was used. The second case study of a smart vacuum system (SVS) demonstrates the application of Providentia in a different domain. The SVS is an autonomous robotic vacuum modeled as an SAS and tasked with cleaning a given environment, where adaptations are performed at run time to switch between different configuration modes (Bencomo et al., 2010; Bencomo and Belaggoun, 2013). Experimental results suggest that the SVS goal model optimized with Providentia performs better than SVS goal models with manually- and randomly-selected FR weights. Finally, results from both the RDM and SVS case studies indicate that the overall fitness can be further improved with Providentia-SAW.

Extensions. This paper extends an earlier description of the Providentia technique along several dimensions. First, we introduce Providentia-SAW that makes use of a second level of abstraction to explore the trade-offs between non-functional, functional, and adaptive requirements. Then we apply Providentia to a second case study, an autonomous robotic vacuum (SVS), to further demonstrate the technique's effectiveness in a different application domain. Providentia-SAW is also applied to this second case study. Finally, this paper includes additional details about the overall techniques and expands the related work discussion.

The remainder of this paper is organized as follows. Section 2 provides relevant background information on SASs, the RDM and SVS, goal modeling, NFRs, utility functions, and genetic algorithms. Section 3 presents the Providentia approach and introduces the integration with Providentia-SAW. Section 4 provides the experimental results of applying Providentia and Providentia-SAW to the RDM and SVS applications, respectively. Following, Section 5 presents a discussion of the results and threats to validity. Section 6 overviews the work related to Providentia and Providentia-SAW, and finally, Section 7 summarizes the findings and overviews future work.

2. Background

This section provides relevant background information on SASs, the case studies, NFRs, utility functions, and genetic algorithms.

2.1. Self-adaptive systems

The explosion of the number of possible combinations of system and environmental parameters often inhibits an engineer's ability to fully enumerate each combination (Cheng et al., 2009a; Whittle et al., 2009). System requirements and objectives may also change following deployment, potentially requiring numerous software updates or patches. An SAS provides an approach for enabling continuous requirements satisfaction by dynamically adapting the system's configuration and/or behavior at run time (McKinley et al., 2004; Oreizy et al., 1999; Neema et al., 1999). As such, the RDM and SVS applications has been modeled as SASs to address uncertainty in the environment and the system itself (Neema et al., 1999).

Uncertainty. Given the exponential number of system and environmental combinations (Cheng et al., 2009a; Whittle et al., 2009), coupled with the possibility that software requirements and/or models will change following deployment (potentially requiring new software or bug fixes), it is difficult to accurately predict or model all situations an SAS may face throughout its lifetime. An SAS provides an approach for continuous requirements satisfaction by enabling self-reconfiguration at run time to mitigate such issues (McKinley et al., 2004; Oreizy et al., 1999). SASs are generally guided by a run-time feedback loop such as MAPE-K, comprising *monitoring, analyzing, planning*, and *executing* components, linked together by common *knowledge* of the system and its elements (Kephart and Chess, 2003). This feedback loop enables an SAS to change its configuration and/or its behavior, at run time, to better mitigate current operating conditions or manifested uncertainties.

While many forms of uncertainty exist (Li et al., 2013), we focus on *known unknowns* and *emergent behaviors/feature interactions*. Known unknowns tend to deal with knowledge of the system's operating conditions, where data may change unexpectedly, be inaccurate, or be in an unanticipated state (Chua Chow and Sarin, 2002; Esfahani et al., 2011). Emergent behaviors/feature interactions occur when multiple subsystems interact, introducing unexpected or possibly dangerous new behaviors that were not explicitly considered at design time (Keck and Kuehn, 1998).

Requirements monitoring. While requirements monitoring is not specific to the SAS domain, monitoring does feature prominently in the MAPE-K loop for providing feedback to the SAS adaptation engine (Kephart and Chess, 2003). Specifically, an SAS that monitors requirements can self-reconfigure in the event that a requirement is violated or unsatisfied. To quantify requirements at run time, utility functions are often used to provide a mathematical quantification (Ramirez and Cheng, 2011) that indicates the degree to which requirements are satisfied. The Rainbow framework uses utility functions at an architectural level (Garlan et al., 2004) and was our initial inspiration for leveraging utility functions, however in this work, we apply them at the requirements level (Ramirez and Cheng, 2011).

2.2. Case studies

This section provides an overview of the two case studies used to demonstrate the Providentia technique: the RDM and the SVS.

2.2.1. Remote data mirroring

RDM is a technique to protect data by minimizing data loss and maximizing the availability of data Ji et al. (2003); Keeton et al. (2004). The RDM technique disseminates data replicates to other servers (i.e., data mirrors) in physically remote locations. Each network link between data mirrors is associated with an operational cost. Furthermore, each link has a throughput, latency, and loss rate to collectively measure the performance and reliability of the RDM as a whole. The RDM must optimize the number of links to send messages between data mirrors efficiently without exceeding the budget.

Requirements may become unsatisfied in the face of various forms of uncertainty, such as unexpectedly dropped or delayed messages, random network link or data mirror failures, and noise in the network links or data mirror sensors. An RDM can be modeled as an SAS (Ramirez et al., 2009), where reconfigurations can change the network topology (e.g., a minimum spanning tree or a redundant topology) as well as the manner of data propagation among nodes to ensure that requirements are continuously satisfied. The reconfiguration strategies involve modifying the status of data mirrors impacted by uncertainty. The status of a data mirror can be active (i.e., can send and receive messages), passive (i.e., cannot send messages but can receive messages), or quiescent (i.e., cannot send or receive messages).

2.2.2. Smart vacuum system

Smart vacuums such as iRobot's Roomba[2] are available in the consumer market to clean dirt by navigating across a room and around obstacles autonomously without guidance from the user. The SVS is an open-source simulation of a Roomba. The SVS operates by using input from sensors (e.g., bumper sensors and motor sensors) to plan a path in a given area and follow the path to clean its local environment. For the purposes of our simulation, the SVS contains bumper sensors to detect when the robot collides with an item (e.g., a wall or leg of a chair), cliff sensors to prevent the robot from damaging itself by falling down a flight of stairs, and motor sensors to provide feedback on the movement of the robot (e.g., velocities of the wheels and power modes for the suction). A controller uses the sensor data to plan an optimal cleaning path and minimize battery consumption.

The SVS can be modeled as an SAS to reconfigure multiple modes during run time when faced with various forms of uncertainty (e.g., noisy sensor data, amount and location of dirt within a room, obstacle encounters, etc.) (Bencomo et al., 2010; Bencomo and Belaggoun, 2013; Neema et al., 1999). For example, the SVS can change the pathfinding algorithm, power modes with regard to movement and suction, and obstacle avoidance measures at run time to maintain requirements satisfaction.

[2] See http://www.irobot.com.

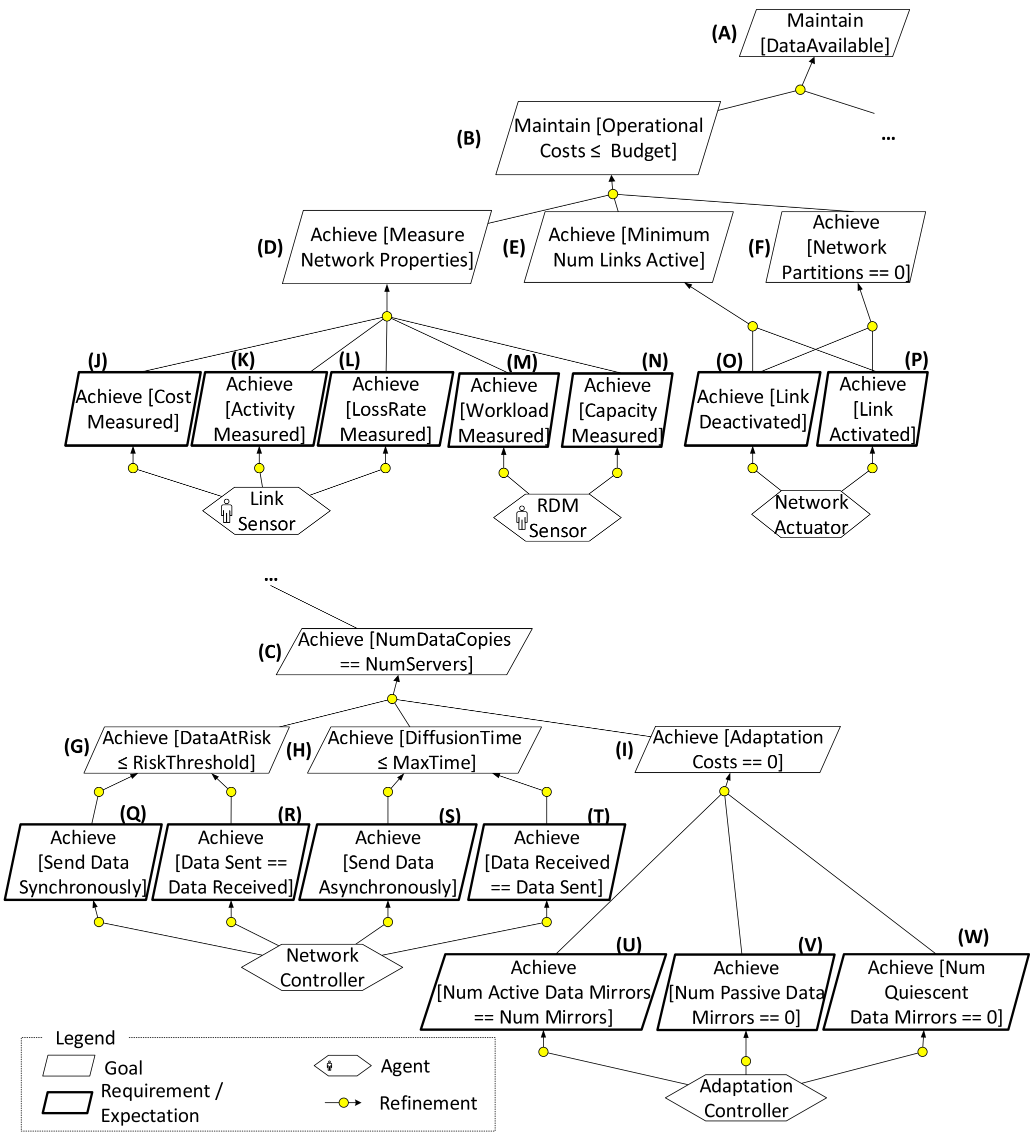

(A) Maintain [DataAvailable]
(B) Maintain [Operational Costs ≤ Budget]
...
(D) Achieve [Measure Network Properties]
(E) Achieve [Minimum Num Links Active]
(F) Achieve [Network Partitions == 0]
(J) Achieve [Cost Measured]
(K) Achieve [Activity Measured]
(L) Achieve [LossRate Measured]
(M) Achieve [Workload Measured]
(N) Achieve [Capacity Measured]
(O) Achieve [Link Deactivated]
(P) Achieve [Link Activated]
Link Sensor
RDM Sensor
Network Actuator
(C) Achieve [NumDataCopies == NumServers]
(G) Achieve [DataAtRisk ≤ RiskThreshold]
(H) Achieve [DiffusionTime ≤ MaxTime]
(I) Achieve [Adaptation Costs == 0]
(Q) Achieve [Send Data Synchronously]
(R) Achieve [Data Sent == Data Received]
(S) Achieve [Send Data Asynchronously]
(T) Achieve [Data Received == Data Sent]
Network Controller
(U) Achieve [Num Active Data Mirrors == Num Mirrors]
(V) Achieve [Num Passive Data Mirrors == 0]
(W) Achieve [Num Quiescent Data Mirrors == 0]
Adaptation Controller
Legend
Goal
Requirement / Expectation
Agent
Refinement

Fig. 1. RDM goal model.

2.3. Goal-oriented requirements modeling

Goal-oriented requirements modeling (GORE) is an approach that uses goals to model the behaviors of the system (van Lamsweerde, 2009). A goal is a desired system behavior achieved through interactions with agents, where an agent is a system component that performs specific actions according to goals. A requirement is a goal that interfaces with a single agent. An expectation is a requirement where the agent is in the environment, compared to agents within the system itself. Requirements and goals can be further classified as functional and non-functional. FRs specify *what* services are to be provided while NFRs specify *how* the FRs are to be satisfied.

GORE uses a directed acyclic graph, where each node represents a goal or requirement and each edge represents a goal/requirement refinement (van Lamsweerde, 2009). KAOS and iStar extend GORE by adding additional goal refinements (van Lamsweerde, 2009; Yu, 1997; Dardenne et al., 1993). Fig. 1 presents a KAOS goal model of the RDM application, and Fig. 2 presents a KAOS goal model of the SVS application.[3] KAOS uses AND- and OR-refinements, where an AND-refined goal is satisfied only when all of its subgoals are satisfied (e.g., Goal (A) in Fig. 1 is satisfied only if both Goals (B) and (C) are also satisfied) and an OR-refined goal is satisfied when at least one of its subgoals is satisfied (e.g., Goal (G) in Fig. 1 is satisfied if either Requirements (Q) or (R) are satisfied). Furthermore, KAOS FRs can be classified as invariant or non-invariant. An invariant goal, denoted by the keywords “Maintain” or “Avoid,” must always be satisfied. If any invariant goal is unsatisfied then the system fails. For example, the safety-related goals in Fig. 2 are denoted as invariant goals to emphasize the safety of the SVS. A non-invariant goal, denoted by the keyword “Achieve,” may be temporarily unsatisfied due to uncertainty. For example, the path planning goals and requirements of the SVS in Fig. 2 (e.g., Goals (C), (H), (I), (O), (P), and (Q)) are non-invariant goals to demonstrate that the SVS may temporarily have a less-than-optimal path to clean a room.

2.4. Non-functional requirements

NFRs specify quality constraints on a system, such as performance or reliability (Chung et al., 2012), and are often difficult to quantify given their subjective nature. Furthermore, NFRs may introduce cross-cutting concerns given the broad impact on the

[3] This work does not use the KAOS formal refinement infrastructure.

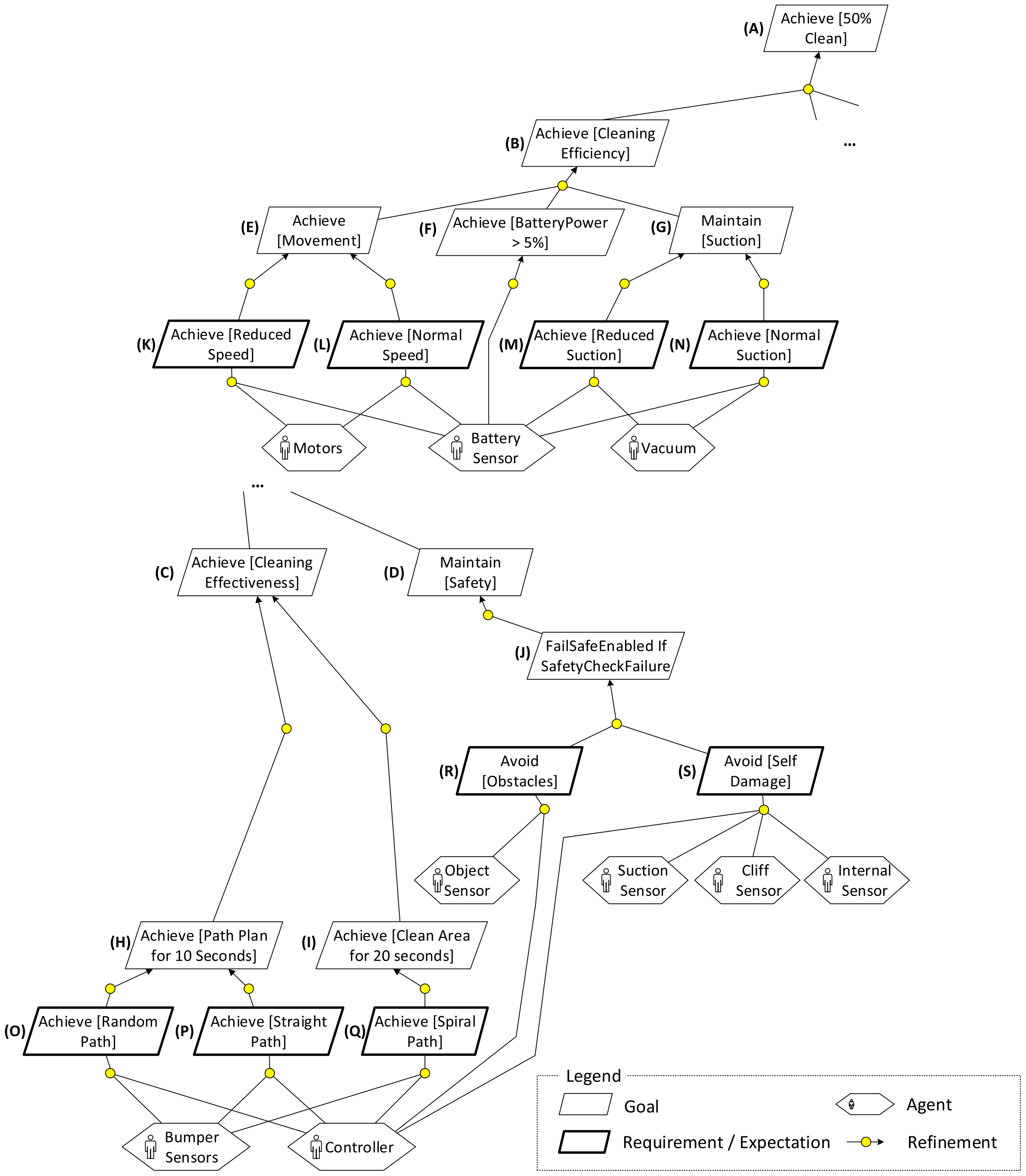

(A) Achieve [50% Clean]
(B) Achieve [Cleaning Efficiency]
...
(E) Achieve [Movement]
(F) Achieve [BatteryPower > 5%]
(G) Maintain [Suction]
(K) Achieve [Reduced Speed]
(L) Achieve [Normal Speed]
(M) Achieve [Reduced Suction]
(N) Achieve [Normal Suction]
Motors
Battery Sensor
Vacuum
...
(C) Achieve [Cleaning Effectiveness]
(D) Maintain [Safety]
(J) FailSafeEnabled If SafetyCheckFailure
(R) Avoid [Obstacles]
(S) Avoid [Self Damage]
Object Sensor
Suction Sensor
Cliff Sensor
Internal Sensor
(H) Achieve [Path Plan for 10 Seconds]
(I) Achieve [Clean Area for 20 seconds]
(O) Achieve [Random Path]
(P) Achieve [Straight Path]
(Q) Achieve [Spiral Path]
Bumper Sensors
Controller
Legend
Goal
Agent
Requirement / Expectation
Refinement

Fig. 2. SVS goal model.

overall system (Chung et al., 2012). For example, the RDM may have one NFR to maximize performance that keeps as many data mirrors in an active state as long as possible. However, there may be a second NFR to minimize power consumption that puts data mirrors in a quiescent state, directly contradicting the performance NFR. Such cross-cutting concerns introduce further complexity in measuring the satisfaction of NFRs. Although other approaches to quantify requirement satisficement have been introduced (Ramirez and Cheng, 2011; Garlan et al., 2004), such models often require detailed knowledge of both the system and its environment that may not always be possible with the wide impact of NFRs.

Therefore, `Providentia` uses FRs already defined as part of the system to quantify NFR objectives. Fig. 3 shows a sample NFR for the RDM application to `Minimize [Power]`, where many factors may impact power consumption (e.g., Goals (A), (E), (I), (V), and (W) from Fig. 1). For the purposes of this paper, Fig. 3 is shown separately from Fig. 1 but is intended to be an extension of the FR goal model rather than a separate NFR goal model. We use NFRs

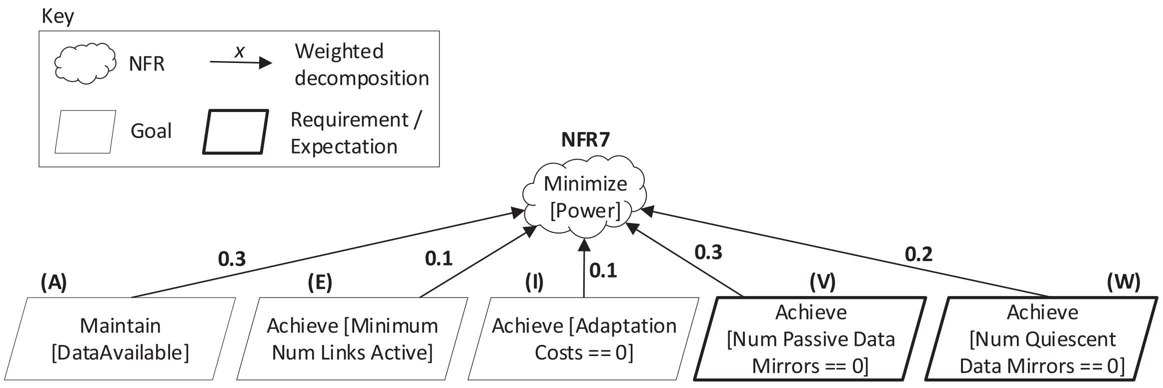

Key
NFR
x
Weighted decomposition
Goal
Requirement / Expectation
NFR7
Minimize [Power]
0.3
0.1
0.1
0.3
0.2
(A)
(E)
(I)
(V)
(W)
Maintain [DataAvailable]
Achieve [Minimum Num Links Active]
Achieve [Adaptation Costs == 0]
Achieve [Num Passive Data Mirrors == 0]
Achieve [Num Quiescent Data Mirrors == 0]

Fig. 3. RDM NFR7: Minimize [Power].

to represent all non-functional goals and requirements and FRs to represent all functional goals and requirements. The cloud node in Fig. 3 represents a single NFR. Each parallelogram node represents a goal or requirement/expectation that a requirements engineer designates to have an impact in the satisficement of NFR7. Each edge depicts the relative contribution, or weighted sum value, of each FR to NFR7. Note that the sum of the weights is 1.0.

Similarly, Fig. 4 shows a sample NFR for the SVS application to `Minimize [Cost]`, where cost refers to avoiding damage to obstacles (e.g., damaging the legs of a dining room chair) and avoiding damage to the smart vacuum itself (e.g., hitting an obstacle that damages a sensor or falling off a ledge).

2.5. Utility functions

A utility function can be used to calculate the satisficement of FRs in SASs (Ramirez and Cheng, 2011; deGrandis and Valetto, 2009; Walsh et al., 2004). The utility function for a specific, single requirement is evaluated and returns a utility value. A utility value of 1.0 indicates the highest degree of satisfaction and a utility value of 0.0 indicates the lowest degree of satisfaction. Utility values between 0.0 and 1.0 indicate the degree of satisficement for a given requirement (Chung et al., 2000). Eq. (1) shows a utility function associated with Goal (V) from the RDM application in Fig. 1, where n indicates the number of passive data mirrors.

$$util(goal_V) = \begin{cases} 1.0 & \text{if } n == 0 \\ f(x) & \text{if } 0 < n < 20\% \text{ of total nodes} \\ 0.0 & \text{if } n \geq 20\% \text{ of total nodes} \end{cases} \quad (1)$$

Goal (V) is considered to be completely satisfied if no data mirrors are in a passive state (i.e., can receive but not send messages) and evaluates to a utility value of 1.0. Goal (V) completely fails if more than 20% of all data mirrors are in a passive state. Otherwise, if the number of passive data mirrors is greater than 0% but less than 20%, then the *f(x)* value is linearly determined. For example, if 10% of the data mirrors are in a passive state, then Goal (V) will have a utility value of 0.5.

2.6. Genetic algorithms

A genetic algorithm is a heuristic used to search a space of solutions to find an optimal result (Holland, 1992). The evolutionary process of a genetic algorithm generates a population of candidate solutions, performs crossover and mutation operations, and evaluates the fitness of each solution within the population. The evolutionary process repeats until the specific number of generations, or iterations, is reached. We next describe each of these activities.

2.6.1. Population generation

A genetic algorithm may start with a randomly-generated set, or population, of candidate solutions in the first generation. The population of individuals represents the number of candidate solutions to be evaluated through the evolutionary process. For each generation, evolutionary operators such as crossover, mutation, and selection are applied to each individual to generate new members of the population and evaluate fitness. Ideally, the most fit individuals are preserved during the evolutionary process, with the best performing individual being considered an optimal result.

2.6.2. Crossover and mutation

Crossover and mutation are evolutionary operators that generate new individuals (i.e., children) during the evolutionary process. While there are many different types of crossover and mutation, we use two-point crossover and single-point mutation. Two-point

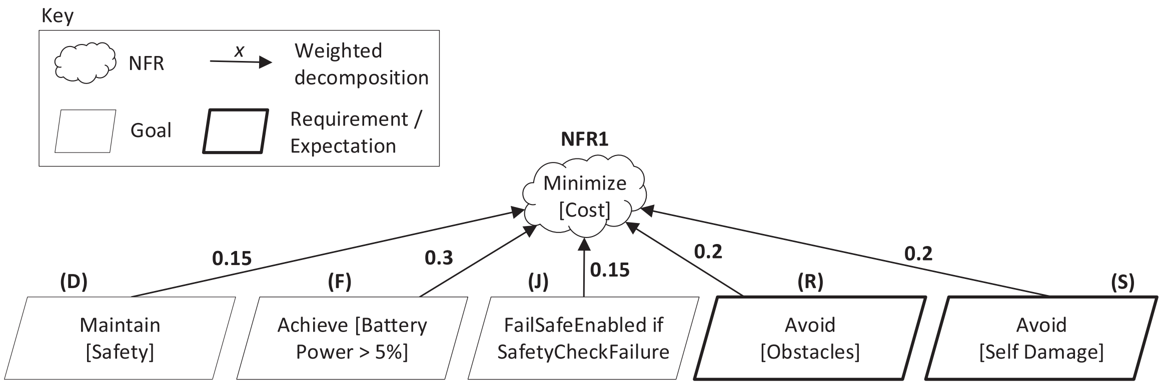


Fig. 4. SVS NFR1: Minimize [Cost].

crossover selects two individuals, either randomly or as a result of tournament selection (Holland, 1992), and then selects two random indices to denote cut points. Two new children are then generated by combining the genetic material from both parents, where genomes are swapped between the two cut points. Single-point mutation selects a single index from a candidate parent and then randomly mutates the gene at that location, creating a new child. Ideally, crossover preserves the best characteristics of its parents, and mutation introduces diversity to ensure that the search procedure does not become “stuck” at a local optimum.

2.6.3. Fitness evaluation

Populations are evaluated according to fitness functions, where each individual within a population is assigned a fitness value. Fitness values determine those individuals that performed better than others. Often, genetic algorithms use a form of tournament selection where a specific number of the highest performing individuals are used in the next generation. In this way, the genetic algorithm is guided toward an optimal value by carrying over the best solutions through each generation but explores different combinations via crossover and mutation to obtain a global optimal solution.

2.6.4. Hyper-heuristic algorithms

Hyper-heuristic algorithms are used to search a space of meta-heuristics Burke et al. (2003). Rather than exploring the problem space for a solution (i.e., a meta-heuristic), hyper-heuristics search among all the solutions to the problem space. Hyper-heuristics operate at an abstraction level above meta-heuristics (Kumari and Srinivas, 2013). For example, the SVS faces a variety of problems (e.g., cliff detection, object collision, water spots to avoid, etc.). `Providentia` is a meta-heuristic that searches for the most effective solution (i.e., combination of FR weights for each NFR) to satisfy the system requirements to the highest degree. However, satisfying the system requirements to the highest degree is a balance between satisfying FRs, NFRs, and minimizing the number of adaptations performed. Therefore, we apply a hyper-heuristic to search for a solution *among the solutions* that maximizes overall system satisficement. For the purposes of this paper, we classify `Providentia-SAW` as a hyper-heuristic that further improves the `Providentia` meta-heuristic genetic algorithm.

3. Approach

This section introduces `Providentia`, our technique that analyzes the weighted contributions of FRs to each NFR in the SAS requirements specification and/or goal model. `Providentia` determines an optimal combination of weights that yields the highest overall satisfaction of the entire goal model (i.e., including NFR and FR satisficement). The `Providentia` technique is intended to be an add-on feature to make a system more robust to uncertainty. `Providentia` assumes that the system is fully operational and contains models of FRs, NFRs, and mechanisms to measure how well those requirements are being satisfied.

The following sections outline the base technique using the RDM case study (Bowers et al., 2018) as a motivating example and extends previous work to include the SVS case study. First, we discuss `Providentia` with respect to expected inputs and outputs. Next, we describe the process of automatically optimizing the input goal model using a genetic algorithm. Finally, we introduce the `Providentia-SAW` technique to balance the satisficement of FRs, NFRs, and the number of adaptations.

3.1. Providentia: assumptions, inputs, and outputs

`Providentia` requires four inputs: (1) a goal model of the SAS, consisting primarily of FRs but may include NFRs defined by a requirements engineer,[4] (2) a list of NFRs with a set of FR suggestions for each, (3) a set of utility functions to measure the satisficement of FRs, and (4) an executable specification with defined sources of uncertainty (for the RDM and SVS case studies, we use environmental and system uncertainty). Each of these inputs are expected to be previously defined by one or more engineers. The sources of uncertainty are parameters (e.g., sensor values, number of dropped messages in a network, etc.) whose values change during runtime that may cause the input FRs and NFRs to become unsatisfied, and therefore trigger a reconfiguration. The `Providentia` technique explores interactions between requirements as well as system behavior while subjected to uncertainty that an engineer may not be able to foresee when designing the system. The output of `Providentia` is a goal model with optimized FR/NFR relationships. Note that the `Providentia` technique is only as good as the accuracy of the input data (i.e., we assume the set of FRs for a given NFR is accurate and that the executable specification is not missing any sources of uncertainty).

Goal model. A KAOS goal model provides `Providentia` with a specification of the FRs and NFRs in the SAS. For example, Figs. 1 and 2 demonstrate the RDM and SVS goal models, respectively.

Utility functions. Each FR in the input KAOS goal model shall have a corresponding utility function to evaluate SAS requirements at run time (deGrandis and Valetto, 2009; Walsh et al., 2004). A utility function maps the FR to a utility value within [0.0, 1.0] to represent the degree to which the FR is satisfied. A requirements engineer is expected to provide a set of utility functions that correspond to the input goal model. A sample utility function is demonstrated in Eq. (1).

Applicable set of FRs mapped to NFRs. A requirements engineer must provide, in addition to the goal model, a set of FRs that may have an impact in the satisficement of an NFR, with the set of NFRs comprising the non-functional properties intended for the system. For example, Fig. 3 for the RDM case study shows an example of an applicable set of FRs, namely Goals (A), (E), (I), (V), and (W), that seem most relevant in minimizing power consumption. However, the requirements engineer may consider expanding the set of five FRs to include Goals (B), (K), (M), (O), (P), and (U) from Fig. 1.

Executable specification. An executable specification or simulation is required in order for `Providentia` to evaluate the overall fitness value of the SAS and determine an optimal combination of FR/NFR weights. We use a simulation of both the RDM and SVS case studies. The RDM is an industry-provided simulation shared with the authors and the SVS is an ongoing part of the authors' research lab. The specification provides the utility values to measure how well the requirements were met at a given instance in time during the simulation so that `Providentia` can determine where to further improve requirement satisficement. The executable specification should also include known sources of uncertainty anticipated by the requirements engineer, both in the environment and the system itself (e.g., liquid spills, sensor noise, etc.). This information better guides the search process at design time and makes the system more robust to known and unknown sources of uncertainty at run time. Moreover, the executable specification must include adaptation mechanisms for the NFRs, should they be violated. The requirements engineer is expected to identify and implement the adaptation mechanisms. Providentia does not change any reconfigurations but rather tries to minimize the number that occur, as each system reconfiguration incurs a cost (e.g., computation time, memory used, etc.). For example, if an NFR for maximizing performance is violated, then a reconfiguration strategy (e.g.,

[4] For the purposes of this paper, the term “requirements engineer” refers to the authors who performed the manual choices for the NFRs.

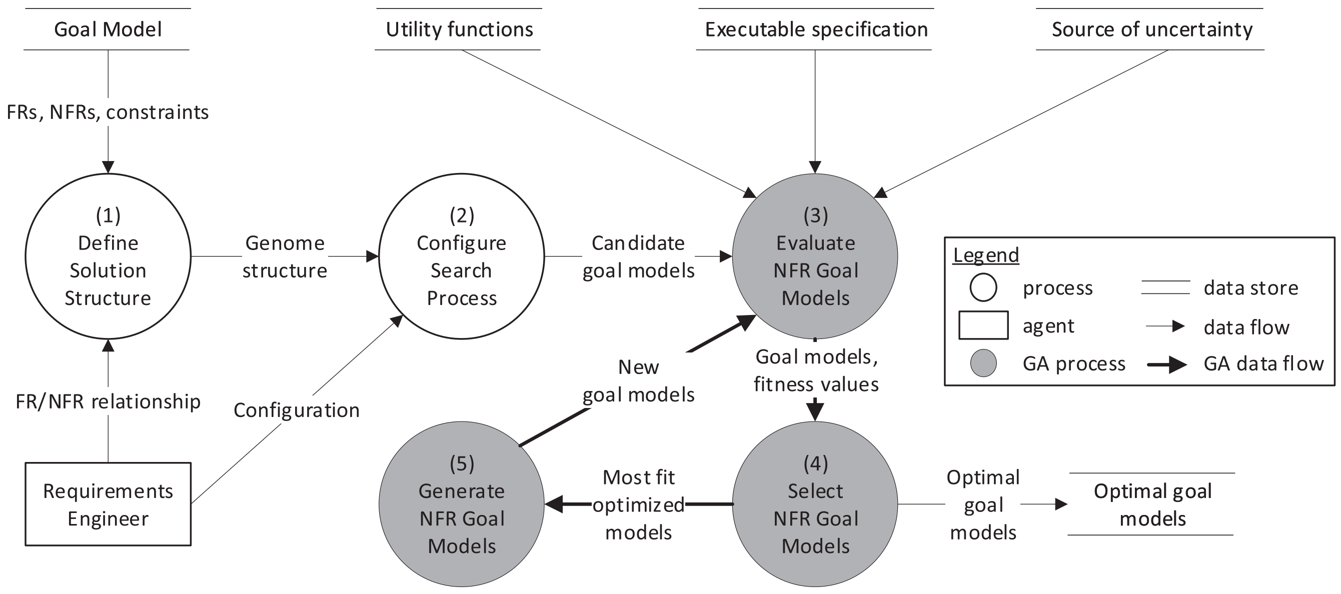


Fig. 5. Data flow diagram of Providentia technique (Bowers et al., 2018).

reconfiguring the RDM network overlay) must be defined in addition to those adaptations already defined for FR violations.

The executable specification acts as a simulation environment where the system is subjected to various randomized forms of uncertainty (e.g., where the RDM experiences a random number of dropped messages, or random connections are lost). The executable specification is not limited to any particular tool, programming language, or environment. Providentia observes how the system behaves and the degree to which FRs and NFRs are satisfied while experiencing adverse conditions. Providentia takes note of the weighted combinations that fulfill both FRs and NFRs to the highest degree and returns the weight assignments. The executable specification can be used as input to Providentia if the specification effectively simulates system behaviors with internal and environmental uncertainty.

Output. The output of Providentia is (1) an NFR goal model integrated with the input FR goal model, (2) for each NFR, a set of FRs that collectively contribute to the satisficement of the NFR, and (3) an optimal weight value assigned to each FR. Note that a weight value of 0.0 for an FR indicates that the FR did not at all contribute to the satisficement of the NFR. For example, the initial set of applicable FRs given for NFR7 from the RDM application in Fig. 1 were Goals (A), (B), (E), (I), (K), (M), (O), (P), (U), (V), and (W). Providentia determined the most optimal weights for each goal respectively to be as follows: B: 0.237144, E: 0.241000, K: 0.007185, M: 0.373794, O: 0.049442, U: 0.067218, V: 0.024216. The weight assignments to Goals (B), (E), and (M) indicate that maintaining costs below the budget, keeping the minimum number of links active, and achieving an accurate measure of the workload most significantly impact the overall power consumption of the RDM. Goals (K), (O), (U), and (V) minimally impacted the satisfaction of power requirements. Goals (A), (E), (I), (P), and (W) have weights 0.0 as they did not contribute to minimizing power consumption. The utility function for NFR7 is calculated by summing the products of the weight and FR utility value for each nonzero FR.

3.2. Providentia technique

This section describes the details of Providentia, comprising a genetic algorithm Holland (1992) to search for optimal FR weights that contribute to each NFR in an SAS. Fig. 5 illustrates a data flow diagram of the Providentia technique with each step further described in detail.

(1) Define solution structure. Each of Providentia's candidate solutions (i.e., the weights of the FRs associated with an NFR) is encoded in a genome shown in Fig. 6. The genome as a whole consists of all NFRs in the SAS. For example, the RDM case study has 7 NFRs. A single NFR is referred to as a sub-genome, denoted by the bold border in Fig. 6. A sub-genome is broken down into genes (i.e., corresponding weights for one NFR), where a single gene is a weight corresponding to a single FR. The sum of weights within one sub-genome (i.e., within one NFR) must be equal to 1.0.

(2) Configure search process. The search process of a genetic algorithm is controlled by the population size, number of generations, crossover rate, mutation rate, and selection rate. Based on empirical evidence on convergence rates Bowers et al. (2018), we specify the following configuration parameters: population size 20, 50 generations, 25% crossover rate, 50% mutation rate. With regard to the selection rate, we use the tournament selection approach Holland (1992) where the three individuals with the highest utility values become parents of the next generation. Larger values for population and generation sizes were considered (e.g., 25–50 population individuals and 50–100 generations) but optimal convergence was discovered on average at the specified values.

(3) Evaluate NFR models. Once the search process has been appropriately configured, each individual candidate solution is input to the executable specification and evaluated to compare against other individuals within the population. The overall fitness function for the SAS is shown in Eq. (2) and evaluates the satisfaction of FRs, NFRs, and the number of adaptations (NA) into a single fitness value.

$$FF = \begin{cases} \alpha_{NFR} * FF_{NFR} + \alpha_{FR} * FF_{FR} + \alpha_{NA} * FF_{NA} & \text{iff invariants true} \\ 0.0 & \text{otherwise} \end{cases} \quad (2)$$

Eq. (2) is a linear-weighted sum, where α_{NFR}, α_{FR}, and α_{NA} are further optimized by the Providentia-SAW technique (Providentia-SAW is described in Section 3.3). The weights must cumulatively sum to a value of 1.0. The terms FF_{NFR} and FF_{FR} represent fitness sub-functions. Note that if any invariant goals or requirements are unsatisfied, then the system is considered to be failed and evaluates its fitness value to be 0.0. Although many

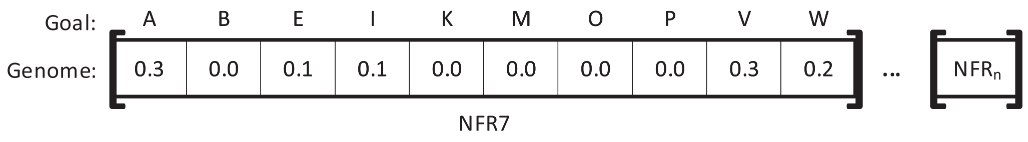


Fig. 6. `Providentia` sample genome.

other approaches exist to combine fitness sub-functions, we find that a linear-weighted sum balances competing concerns between requirement satisficement and adaptations adequately for this experiment.

The fitness sub-function to evaluate the satisficement of NFRs is shown in Eq. (3).

$$FF_{NFR} = \frac{\sum_{i=1}^{|NFRs|} utility\_value_{NFR_i}}{|NFRs| * timesteps} \tag{3}$$

The numerator in Eq. (3) represents the sum of utility values for all NFRs in the SAS. For example, the RDM application contains seven NFRs, resulting in a numerator evaluation of: $utility\_value_{NFR_1} + utility\_value_{NFR_2} + \cdots + utility\_value_{NFR_7}$. The denominator of Eq. (3) is the number of NFRs in the SAS (e.g., seven for the RDM) multiplied by the number of timesteps run in the executable specification.

The utility value calculation of each NFR from the numerator in Eq. (3) is shown in Eq. (4), where n refers to the number of the NFR (e.g., NFR_1, NFR_2, etc.).

$$utility\_value_{NFR_n} = \sum_{i=1}^{|FR_{NFR_n}|} utility\_value_{FR_i} * weight_{FR_i} \tag{4}$$

Eq. (4) calculates a utility value for one NFR by evaluating the utility function of each supporting FR (e.g., Eq. (1) supplied by the input utility functions), multiplying the utility value with its corresponding weight, and then summing the results for all supporting FRs for a given NFR. The term $weight_{FR_i}$ represents the weights present in `Providentia`'s genes and is the focus of the technique's optimization.

The fitness sub-function for the FF_{FR} term from Eq. (2) is shown in Eq. (5).

$$FF_{FR} = \frac{\sum_{i=1}^{|FRs|} utility\_value_{FR_i}}{|FRs| * timesteps} \tag{5}$$

Eq. (5) sums all of the utility values for the functional requirements and goals of the SAS in the numerator and divides by the product of the number of FRs and the number of timesteps in the executable specification.

Finally, the equation to minimize the number of adaptations performed by the SAS is shown in Eq. (6), where $|adaptations|$ refers to the total number of reconfigurations performed by the SAS and $|faults|$ reports the total number of adverse conditions introduced within the executable specification.

$$FF_{NA} = 1.0 - \frac{|adaptations|}{|faults|} \tag{6}$$

Eq. (6) defines the term FF_{NA} from Eq. (2). A minimized number of adaptations performed by an SAS reduces the overall disruption of the system.

(4) Select NFR models. `Providentia` uses tournament selection to select the genomes with the highest fitness value among k genomes to be used as a parent for the next generation. The remaining individuals not chosen in the population are removed from consideration.

(5) Generate NFR models. When the most fit individuals from the population are selected, `Providentia` then performs crossover and mutation to obtain new individuals for the following generation. `Providentia` uses two-point crossover that selects two points on a genome within an NFR (i.e., crossover does not occur between different NFRs) and swaps the genes with a second genome, creating two new candidate solutions. Furthermore, `Providentia` performs single-point mutation that selects a single gene (i.e., FR weight) and randomly generates a value within ± 20% of its original value. Note that although values are modified via crossover and mutation, the weights must still be normalized within a particular NFR/sub-genome.

Steps (3) - (5) are applied iteratively until the number of generations is reached. When the genetic loop concludes, `Providentia` returns a set of optimized weights of FRs for each NFR.

3.3. SAW integration

This paper extends the original `Providentia` technique (Bowers et al., 2018) with `SAW` optimization (Eiben and van der Hauw, 1998; van der, 1996) to explore a search space containing an optimal weighting scheme that balances the competing concerns between FR satisficement, NFR satisficement, and the number of SAS adaptations. Our preliminary work used an empirically chosen weighting scheme where the terms from Eq. (2) were set as follows: $\alpha_{NFR} = 0.375$, $\alpha_{FR} = 0.375$, and $\alpha_{NA} = 0.25$. However, the chosen weights may not have considered all possible values in the search space, given the uncertainty surrounding an SAS in terms of its environment and configured parameters. Therefore, we apply the hyper-heuristic `SAW` to gradually update the weighting scheme in tandem with `Providentia`, denoted as `Providentia-SAW`, to explore how and whether overall SAS fitness may be improved over the course of system execution.

`Providentia` uses the *online* `SAW` method of optimization described in the original `SAW` paper (van der, 1996). Step (3) in Fig. 5 is augmented with the `SAW` technique. Rather than randomly generating the weights, a requirements engineer can provide initial values (i.e., seed) for `Providentia-SAW` to use in the first generation. The fitness sub-functions are evaluated every *fifth* generation and the sub-function with the lowest fitness value (i.e., in comparison to the other fitness sub-functions in the equation) has its corresponding weight increased to guide the search process for an optimal combination of weights. By increasing the weight of the least fit sub-function, the search process ensures that the weighting scheme balances all concerns appropriately to achieve the highest overall fitness value. The generation number to evaluate the fitness sub-functions (i.e., five) was determined empirically. Other numbers, both higher and lower, were simulated, however optimal results were obtained when evaluating every fifth generation. This generation value balances the competing concerns of adapting too frequently, where the search space does not have enough time to consider optimal solutions, and adapting too infrequently, where more optimal solutions are never explored.

4. Experimental results

This section describes the experimental setup and results for both the RDM and the SVS case studies. The RDM is an industry-provided case example and the SVS was created by the authors' research lab. First, we introduce the experiment setup parameters that we used and the types of adaptations performed for

Table 1
RDM NFRs with sets of FRs and manually-derived weights.

NFR	Set of FRs
NFR1: Maximize [Reliability]	(A), (B), (C), (E), (F), (G), (H), (I), (L), 0.4, 0.2, 0.2, 0.0, 0.0, 0.1, 0.0, 0.1, 0.0, (M), (N), (R), (T), (U), (V), (W) 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
NFR2: Maximize [Throughput]	(A), (C), (D), (E), (F), (G), (H), (L), (M), 0.0, 0.6, 0.0, 0.0, 0.0, 0.1, 0.1, 0.0, 0.0, (N), (O), (P), (Q), (R), (S), (T), (U), (V), 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.1, 0.0, 0.0, (W) 0.0
NFR3: Maximize [Speed]	(C), (D), (H), (I), (K), (M), (Q), (S), (U), 0.2, 0.1, 0.3, 0.2, 0.0, 0.0, 0.0, 0.2, 0.0, (V), (W) 0.0, 0.0
NFR4: Maximize [System Security]	(A), (B), (D), (G), (H), (L) 0.4, 0.2, 0.2, 0.1, 0.1, 0.0
NFR5: Maximize [Secure Communication]	(C), (G), (H), (Q), (R), (S), (T), (W) 0.5, 0.3, 0.1, 0.0, 0.0, 0.0, 0.0, 0.1
NFR6: Maximize [Message Security]	(C), (D), (H), (R), (T) 0.2, 0.2, 0.0, 0.3, 0.3
NFR7: Minimize [Power]	(A), (B), (E), (I), (K), (M), (O), (P), (U), 0.3, 0.0, 0.1, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, (V), (W) 0.3, 0.2

each application. Next, we present our results for each application to compare the performance of randomly-generated, manually-selected, and Providentia-derived FR/NFR weights. We then describe how Providentia-SAW enabled further optimization towards the satisficement of SAS requirements.

4.1. RDM study

This section describes the experimental setup and results of the RDM application.

4.1.1. RDM experimental setup

The RDM application is modeled as a completely-connected graph. Each node of the graph represents an RDM. Each edge of the graph represents a network link. For each trial, system and environmental parameters were randomized based on a model previously presented by Ji et al. (2003) and Keeton et al. (2004). For example, the randomized system parameters include a random number of RDMs (i.e., within [15,30]) and a random number of valid messages (i.e., [100,200]) inserted into RDMs at random timesteps. Each message is required to be replicated to all other RDMs. The RDM simulation was performed over 300 timesteps. In addition to the 23 FRs presented in Fig. 1, we also examine seven NFRs specific to the RDM that are next presented in Table 1.

We compared and evaluated different combinations of FRs and their supporting weights for every NFR. The seven NFRs were derived using three different techniques: (1) FR weights generated by random search (Arcuri and Briand, 2011), (2) manually-selected weights assigned by a requirements engineer, and (3) Providentia-optimized weights. Note that although a requirements engineer initially selects a subset of FRs for each NFR, all three techniques may disable (but not add) one or more FRs by setting the corresponding weight to 0.0, effectively allowing limited flexibility in the selection of FRs as well as the weight. Table 1 shows the initial sets of FRs for each NFR chosen by a requirements engineer with the manually-derived weights listed below.

For instance, NFR6 uses the utility functions from Goals (C), (D), (H), (R), and (T) to calculate its own utility function in aggregate. Specifically, Eq. (7) demonstrates the utility function for NFR6:

$$
\begin{aligned}
util(NFR_6) = {} & \alpha_C * util(goal_C) + \alpha_D * util(goal_D) \\
& + \alpha_H * util(goal_H) + \alpha_R * util(goal_R) \\
& + \alpha_T * util(goal_T)
\end{aligned} \qquad (7)
$$

Each specified NFR has a similar utility function to determine its utility value at run time. Note that Providentia optimizes the α values on a per-NFR basis (i.e., each set of FR weights is optimized by Providentia and normalized to ensure they sum to 1.0 to ensure that the associated utility function is also normalized). To support NFR feedback within the SAS decision loop Kephart and Chess (2003), additional reconfiguration strategies were implemented for the RDM, where the new reconfiguration strategies were defined based on the requirements engineer's knowledge of the system. For instance, reconfiguration as a result of an NFR3 violation resulted in an internal search for a new network overlay. Finally, we configured the genetic algorithm as shown in Table 2.

Each of the parameters in the genetic algorithm were determined empirically. For example, population sizes of 10, 25, 50, etc. were run in the simulation as well as other values for the remaining parameters. Table 2 shows the values where the genetic algorithm converged. Note that the population size (i.e., the number of individuals) combined with the number of generations results in the number of evaluations *per experimental replicate*. Using the values presented in Table 2, each experimental replicate evaluates 1000 individuals. To ensure statistical

Table 2
Configuration of Providentia genetic algorithm.

Parameter	Value
Population size:	20
Number of generations:	50
Crossover rate:	25%
Crossover type:	*Two-point*
Mutation rate:	50%
Mutation type:	*Single-point*
Selection:	*Tournament selection*, $k = 3$
Stepwise adaptation of weights:	*Online, every 5th generation*

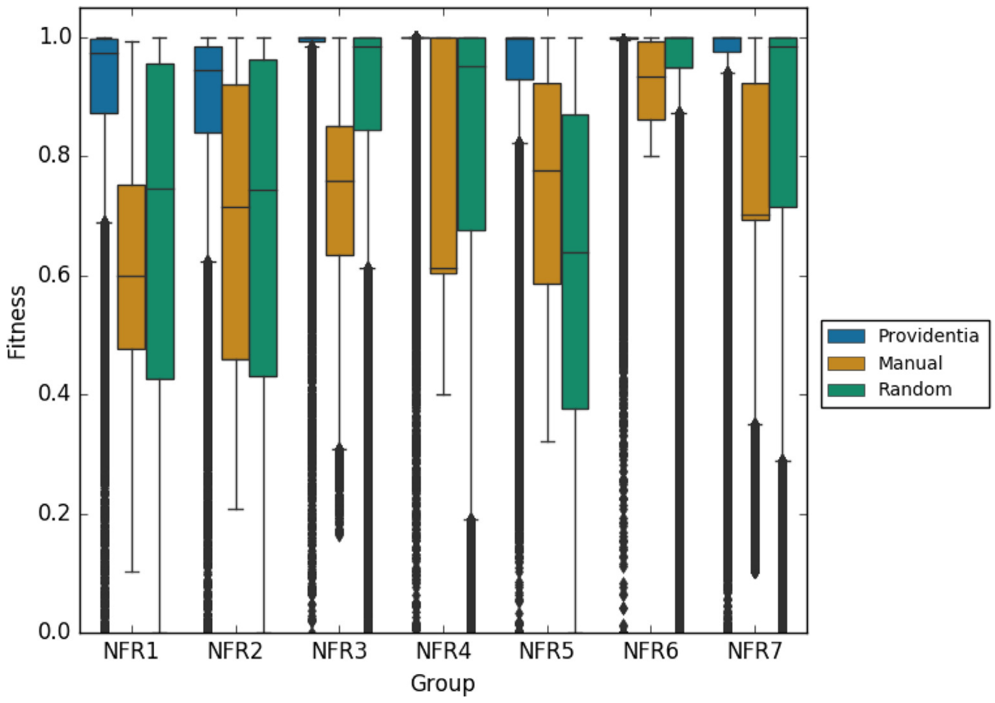


Fig. 7. NFR fitness experimental results Bowers et al. (2018).

significance of our results, we performed 50 experimental replicates, each of which was seeded differently. The p-value was calculated using the Wilcoxon-Mann-Whitney u-test by comparing the results of (1) manually-selected weights vs. randomly-generated weights, (2) Providentia-optimized weights vs. randomly-generated weights, and (3) Providentia-optimized weights vs. manually-selected weights. Manually selected weights indicate a single set of weights chosen by engineers to best represent the contribution of each FR to each NFR. The configurations for the genetic algorithm shown in Table 2 produced the experimental results used in evaluating `Providentia` and `Providentia-SAW`.

4.1.2. *RDM experimental results*

This section presents our results from investigating how NFRs impact an SAS. Specifically, we examine how a set of FRs can contribute to the satisfaction of NFRs that are then in turn incorporated into the SAS decision loop (Kephart and Chess, 2003) to support run-time reconfigurations. We compare and evaluate the impact of applying automatically-selected `Providentia` FR weights with manual and random selection, respectively. For this experiment, all FR weights associated with each NFR are normalized to 1.0. For manual weight selection, we apply the weights as shown in Table 1, where weights of 0.0 indicate that the related FR's utility function is not applied to the calculation of its associated NFR.

With respect to the fitness function in Eq. (2), we set $\alpha_{NFR} = 0.375$, $\alpha_{FR} = 0.375$, and $\alpha_{NA} = 0.25$, where these values were selected based on empirical evidence.

For this experiment, we define two null hypotheses. First, $H1_0$ states that "there is no difference in fitnesses achieved by a `Providentia`-optimized goal model and those that are unoptimized." Second, $H2_0$ states that "there is no difference in fitnesses achieved by a `Providentia`-optimized goal model and those manually optimized by a requirements engineer."

Fig. 7 shows three boxplots for each NFR in the RDM with average fitness values calculated from randomly-selected FR weights, manually-selected FR weights, and `Providentia`-optimized FR weight selection. Fig. 7 demonstrates that `Providentia` can significantly improve overall NFR fitness than those manually selected by a requirements engineer or selected at random ($p < .05$, Wilcoxon-Mann-Whitney u-test). Table 3 presents the average utility values (μ) and standard deviation (σ) for each NFR, with the optimal value highlighted in gray. These results suggest that `Providentia` can improve overall NFR fitness when an SAS is subject to uncertainty.

`Providentia` also enabled a significant *decrease* in the number of encountered FR violations in comparison to randomly- and manually-defined FR weights ($p < .05$, Wilcoxon-Mann-Whitney u-test), as shown in Fig. 8. As requirements violations tend to signify a significant problem with a system, a reduction in run-time violations is an ideal result for an optimization procedure.

Given the results presented in Fig. 7, Table 3, and Fig. 8, we can reject both $H1_0$ and $H2_0$ and accept our alternate hypotheses that `Providentia` provides a significant improvement over manual and random search.

We next describe the integration of `SAW` within `Providentia` to provide further points of optimization.

Table 3
NFR average utility values and standard deviations Bowers et al. (2018).

NFR	Random	Manual	Providentia
`NFR1: Maximize`	μ: 0.654	μ: 0.615	μ: 0.905
`[Reliability]`	σ: 0.325	σ: 0.191	σ: 0.149
`NFR2: Maximize`	μ: 0.655	μ: 0.666	μ: 0.882
`[Throughput]`	σ: 0.325	σ: 0.262	σ: 0.153
`NFR3: Maximize`	μ: 0.875	μ: 0.743	μ: 0.975
`[Speed]`	σ: 0.207	σ: 0.148	σ: 0.085
`NFR4: Maximize`	μ: 0.802	μ: 0.736	μ: 0.979
`[System Security]`	σ: 0.273	σ: 0.177	σ: 0.085
`NFR5: Maximize`	μ: 0.621	μ: 0.742	μ: 0.925
`[Secure Communication]`	σ: 0.273	σ: 0.191	σ: 0.146
`NFR6: Maximize`	μ: 0.921	μ: 0.919	μ: 0.980
`[Message Security]`	σ: 0.181	σ: 0.072	σ: 0.069
`NFR7: Minimize`	μ: 0.821	μ: 0.758	μ: 0.926
`[Power]`	σ: 0.270	σ: 0.172	σ: 0.188

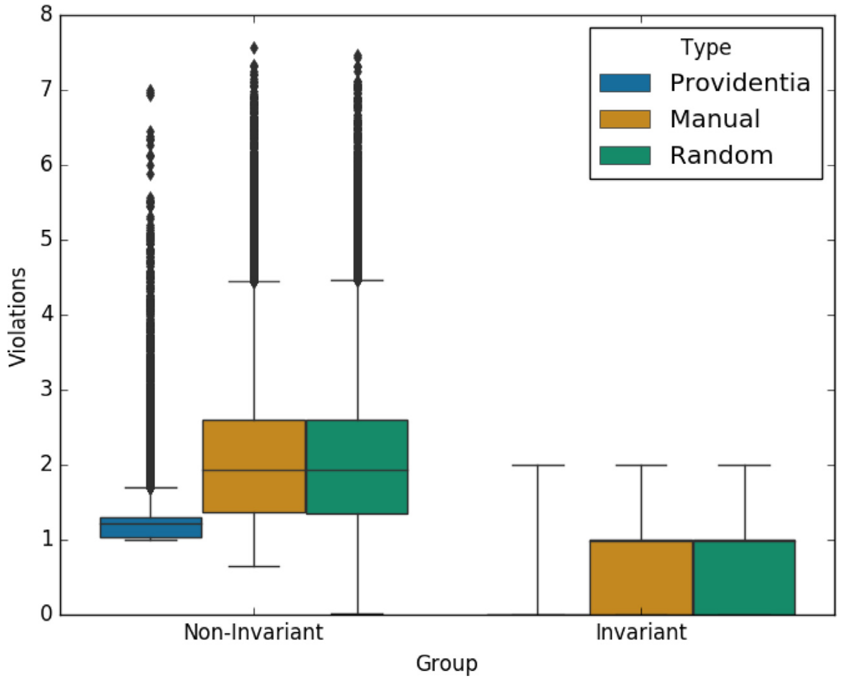


Fig. 8. FR violation experimental results Bowers et al. (2018).

4.1.3. *RDM results with SAW integration*

We now examine how the fitness function weights (α_{NFR}, α_{FR}, and α_{NA}; c.f., Eq. (2)) impact overall fitness, as the weights were initially selected based on empirical evidence and domain knowledge Bowers et al. (2018). We apply `SAW` to dynamically adjust the weights during execution (c.f., Section 3.3). For this experiment, we executed `Providentia` with and without `SAW` applied. We reuse the configuration of the genetic algorithm as presented in Table 2. Moreover, `SAW` dynamically updates the weights of the fitness function every fifth generation, where the poorest-performing fitness subfunction's weight is increased and the remaining subfunction weights are normalized to sum to 1.0.

We define an additional null hypotheses for this experiment. $H3_0$ states that "there is no difference between a `Providentia`-optimized goal model with static fitness subfunction weights and a goal model with dynamically-optimized fitness subfunction weights." For this experiment, we reuse the static weights defined in the previous section (i.e., $\alpha_{NFR} = 0.375$, $\alpha_{FR} = 0.375$, and $\alpha_{NA} = 0.25$) as both our static weights and seed weights for `Providentia-SAW`.

Fig. 9 presents two boxplots that show the fitness values obtained from goal models optimized with `Providentia` and `Providentia-SAW`, respectively. As this figure demonstrates, applying `SAW` to `Providentia` results in higher fitness values ($p < 0.05$, Wilcoxon-Mann-Whitney u-test), suggesting that weights that are dynamically adjusted better reflect the environment and/or configuration of the system. Moreover, these results enable us to reject $H3_0$ and conclude that the fitness function weighting scheme directly impacts overall fitness resulting from monitoring the system.

Next, Fig. 10 presents a set of grouped boxplots that demonstrate the average utility value (calculated by the aggregate utility values of each associated FR) for each NFR. As can be seen by this figure, there is no statistical difference that exists between the average utility values of each NFR that were optimized by `Providentia` and `Providentia-SAW`, respectively ($p > .05$, Wilcoxon-Mann-Whitney u-test). This result is interesting in that overall FR fitness was significantly improved (c.f., Fig. 9), however there is no improvement for the NFR values. This result suggests that the NFR utility values may be independent of the overall RDM fitness calculation, however reconfigurations that are performed at a coarser-grain (NFRs) can significantly impact and improve the performance of FR utility functions.

Fig. 11 shows the average number of *non-invariant* and *invariant* requirement violations that occurred during execution, respectively. Again, there is no significant difference ($p > .05$, Wilcoxon-Mann-Whitney u-test) in the number of violations between `Providentia` and `Providentia-SAW`, suggesting that adjusting the fitness function weights does not significantly impact the SAS reconfiguration engine for the RDM application.

Lastly, Fig. 12 provides a comparison of the starting and ending fitness sub-function weights attained with `Providentia-SAW`. As can be seen from this figure, α_{FR} is maximized and α_{NFR} and α_{NA} tend to be minimized, thereby suggesting that the satisficement of FRs is considerably more important to the performance of the RDM application. This result correlates with those found in Figs. 10 and 11 in that `SAW` seems to have a minimal impact on NFR satisficement and violation reduction. However, an improvement in overall performance of the RDM application still presents a significant finding.

We next repeat our experiments on the SVS application to demonstrate the domain independence of `Providentia` and `Providentia-SAW`.

4.1.4. *SVS experimental setup*

The SVS (c.f., Section 2.2.2) comprises an autonomous vacuum system tasked with cleaning a desired space *safely*. As such, the SVS must balance competing concerns to satisfy its requirements, including maximizing cleaning efficiency, minimizing needless power consumption, and ensuring the safe operation of the vacuum.

For this experiment, the SVS was simulated in the Open Dynamics Engine,[5] where its physical appearance (c.f., Fig. 13) and

[5] See http://www.ode.org/.

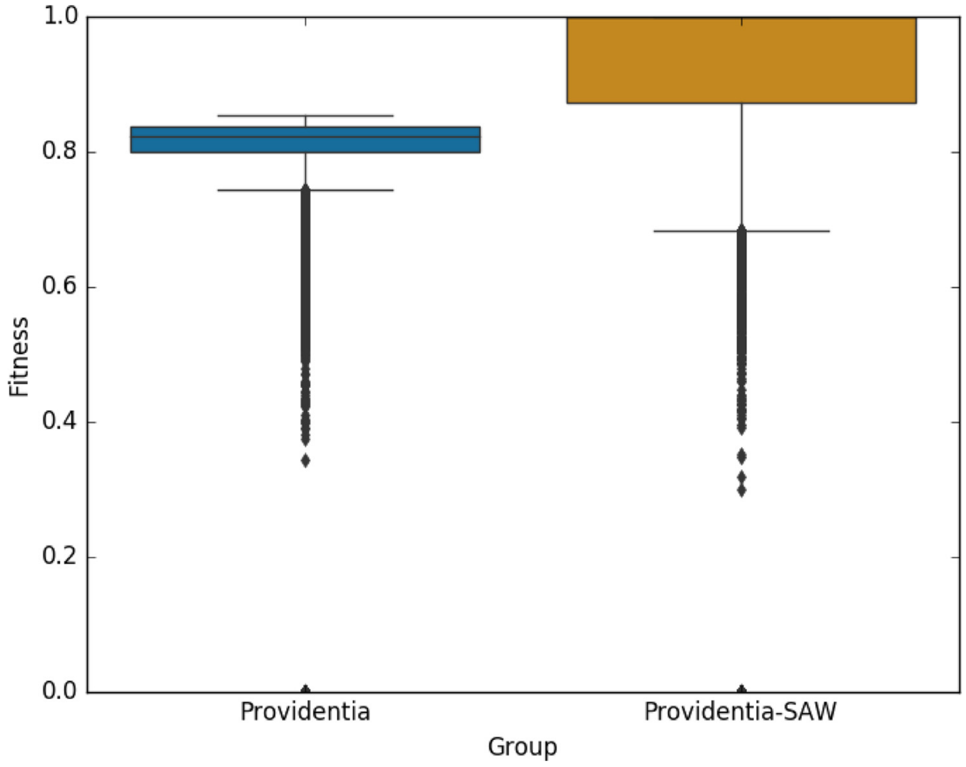


Fig. 9. Comparison of fitness values between `Providentia` and `Providentia-SAW` experiments for the RDM application.

behaviors were modeled on observations from the iRobot Roomba vacuum system. The SVS comprises two wheels, a circular body, seven touch sensors for collision detection, two wheel velocity sensors to monitor the status of each individual wheel speed, and a vacuum sensor that monitors the suction capabilities of the robot.

To motivate the need for run-time adaptation, uncertainty was configured in terms of system and environment-based uncertainty. System uncertainty comprised random sensor noise, sensor failures induced at random during execution, and fluctuations in the main controller timing logic (e.g., variations in the amount of time

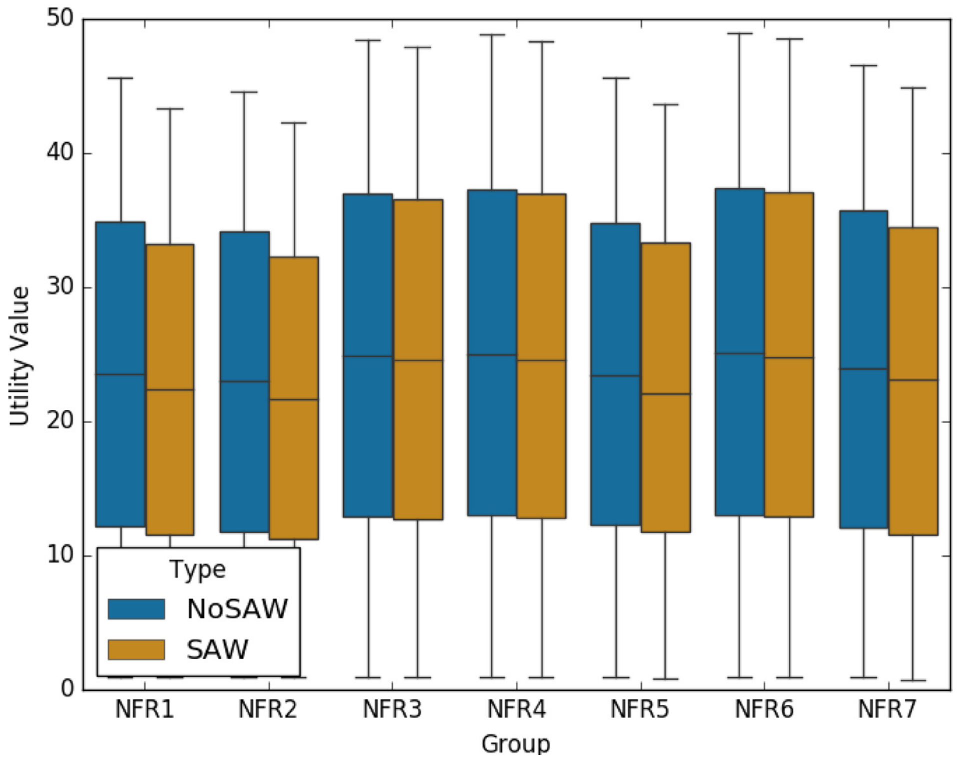


Fig. 10. Comparison of NFR fitness values between `Providentia` and `Providentia-SAW` experiments for the RDM application.

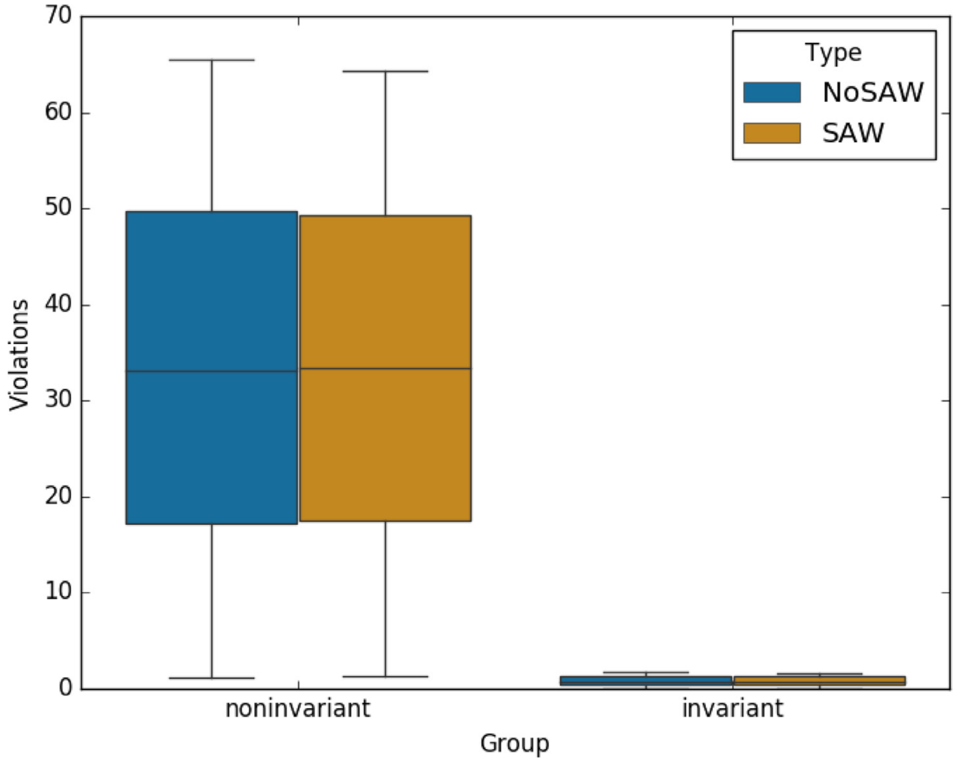


Fig. 11. Comparison of requirement violations between `Providentia` and `Providentia-SAW` experiments for the RDM application.

to spend in a particular path plan) to represent concerns that are found in real-time operating systems. Environment uncertainty includes the amount and distribution of dirt spread throughout the room, a downward step to avoid, and randomly-placed objects that may either hurt the SVS (e.g., a pole or liquid spill) or the object itself (e.g., a pet or child) that introduce safety concerns.

The SVS will reconfigure to minimize the impacts of uncertainty, thereby maximizing overall requirements satisficement. As with the RDM, the SVS performs run-time requirements monitoring via utility functions to quantify the performance of each separate requirement, where violations and/or unsatisfactory performance result in a reconfiguration. Possible reconfigurations

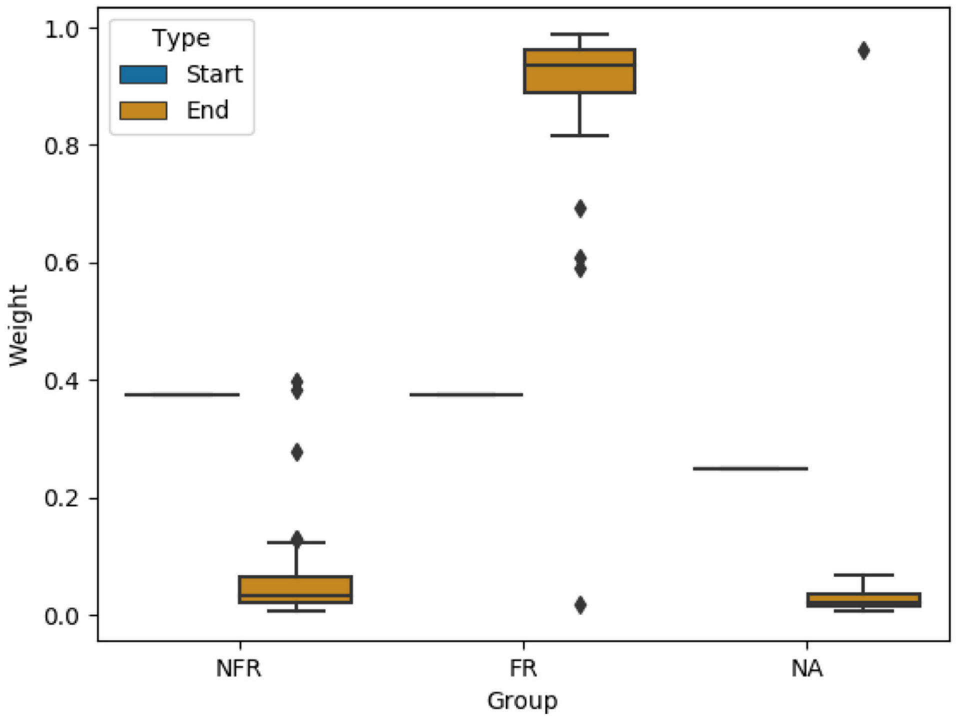


Fig. 12. Comparison of starting/ending fitness subfunction weights for the RDM application with and without `SAW` optimization.

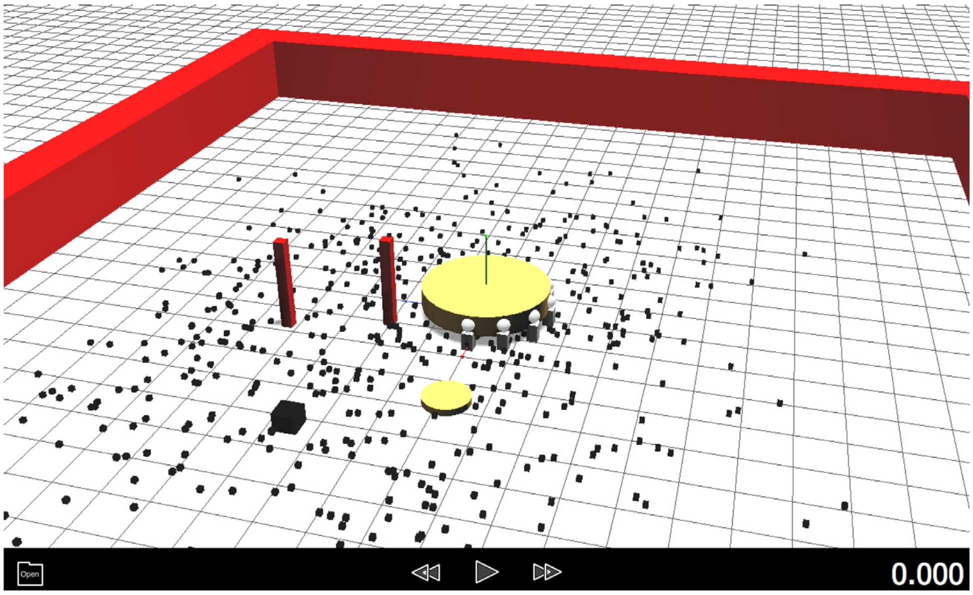


Fig. 13. Screenshot of SVS simulation environment in the Open Dynamics Engine.

include changing the current path plan (e.g., from a spiral to random search), updating the power moding strategy (e.g., from full power to reduced power), and instantiating emergency avoidance procedures to bypass a critical object.

For the SVS, we specify three NFRs to guide the system. Table 4 specifies the defined NFRs for the SVS, along with the sets of FRs and respective weights that comprise its aggregate utility function. Note that unlike the RDM application, Providentia is not provided FRs with weights of 0.0 to demonstrate its effectiveness in a smaller search space (i.e., fewer FRs) for each NFR.

4.1.5. SVS experimental results

For this experiment, we reuse the experimental setup for the RDM (c.f., Section 4.1.1). Specifically, we examine how Providentia-optimized FR weighting schemes compare with weighting schemes that were manually- and randomly-specified (note that, for presentation purposes, we condense our results section to include SAW as well as the replication of the Providentia experiment in the SVS application domain). As the fitness function introduced in Eq. (2) was specific to the RDM application, we now extend the fitness function to be specific to the SVS. The fitness function for the SVS is shown in Eq. (8):

$$FF = \begin{cases} \alpha_{NFR} * FF_{NFR} + \alpha_{FR} * FF_{FR} + \\ \alpha_{adaptations} * FF_{adaptations} & \text{iff invariants true} \\ \\ 0.0 & \text{otherwise} \end{cases} \tag{8}$$

First, we combine these fitness subfunctions with those fitness subfunctions specific to Providentia, as previously defined in Eqs. (3) and (5). As with the RDM, we set $\alpha_{NFR} = 0.375$, $\alpha_{FR} = 0.375$, and $\alpha_{adaptations} = 0.25$. We substitute Eq. (6) that is specific to the RDM with Eq. (9), specific to the SVS, to minimize the number of adaptations that the SVS experiences at run time:

$$FF_{adaptations} = \frac{1.0}{|adaptations|} \tag{9}$$

Fig. 14 presents boxplots that show the fitness values between goal models that were optimized with Providentia-SAW and Providentia, those whose FR weights were manually-selected (Manual), and those whose weights were randomly selected (Random). As can be seen from the plots, optimizing with Providentia significantly improves overall fitness of the SVS ($p < .05$, Wilcoxon-Mann-Whitney u-test). Moreover, introducing SAW to optimize the fitness sub-functions that guide Providentia (c.f., Eq. (8), α_{NFR}, α_{FR}, $\alpha_{adaptations}$) further significantly improves fitness ($p < .05$, Wilcoxon-Mann-Whitney u-test).

Similar to the results presented for the RDM application, Fig. 15 demonstrates how the SVS attains significant NFR fitness improvements as Providentia and then Providentia-SAW are applied ($p < 0.05$, Wilcoxon-Mann-Whitney u-test).

Table 5 and Fig. 16 demonstrate that although Providentia-SAW provides higher fitness than Providentia, the number of invariant and noninvariant violations are not significantly increased, therefore suggesting a positive impact to the Providentia technique with minimal negative repercussions.

Table 6 presents the average utility values (μ) and standard deviation (σ) for each of the SVS NFRs. As with the RDM, we see a significant improvement in average utility for most NFRs ($p < .05$, Wilcoxon-Mann-Whitney u-test), further suggesting that Providentia and Providentia-SAW significantly improve system performance while experiencing uncertainty.

Table 4
SVS NFRs with sets of FRs and manually-derived weights.

NFR	Set of FRs
NFR1: Minimize [Cost]	(D), (F), (J), (R), (S) 0.15, 0.3, 0.15, 0.2, 0.2
NFR2: Minimize [Time]	(A), (B), (C), (E), (G) 0.3, 0.2, 0.2, 0.15, 0.15
NFR3: Maximize [Performance]	(A), (B), (C), (D) 0.3, 0.25, 0.25, 0.2

Table 5
Invariant and noninvariant average violations and standard deviations for SVS application.

	Providentia	SAW
Invariant violations	μ: 3.592 σ: 25.042	μ: 3.418 σ: 25.292
Noninvariant violations	μ: 732.765 σ: 36.174	μ: 748.941 σ: 45.178

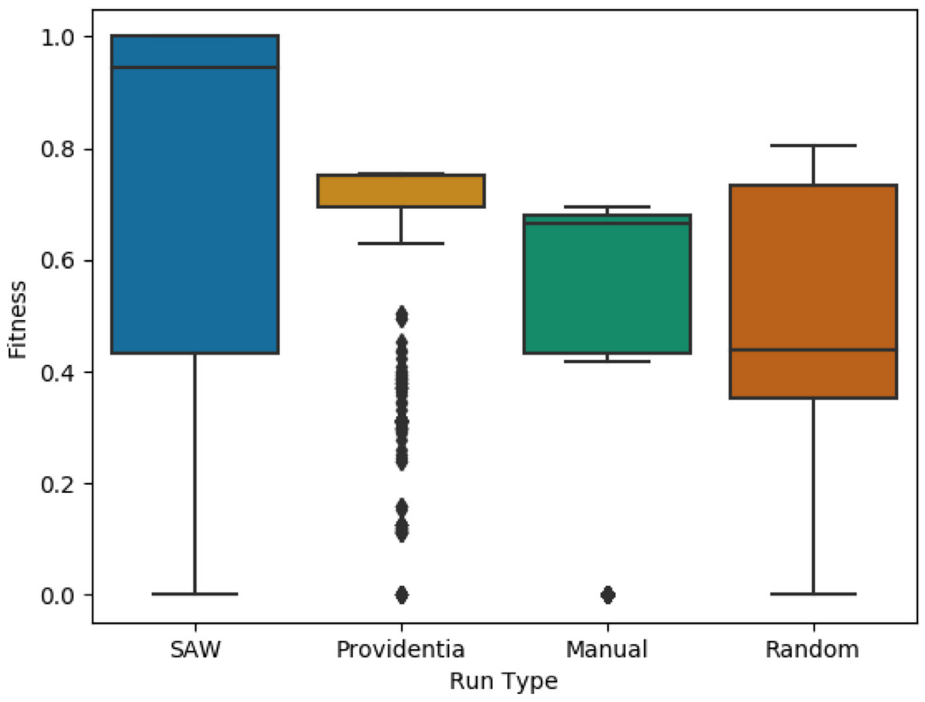


Fig. 14. Comparison of fitness values between all run types for the SVS application.

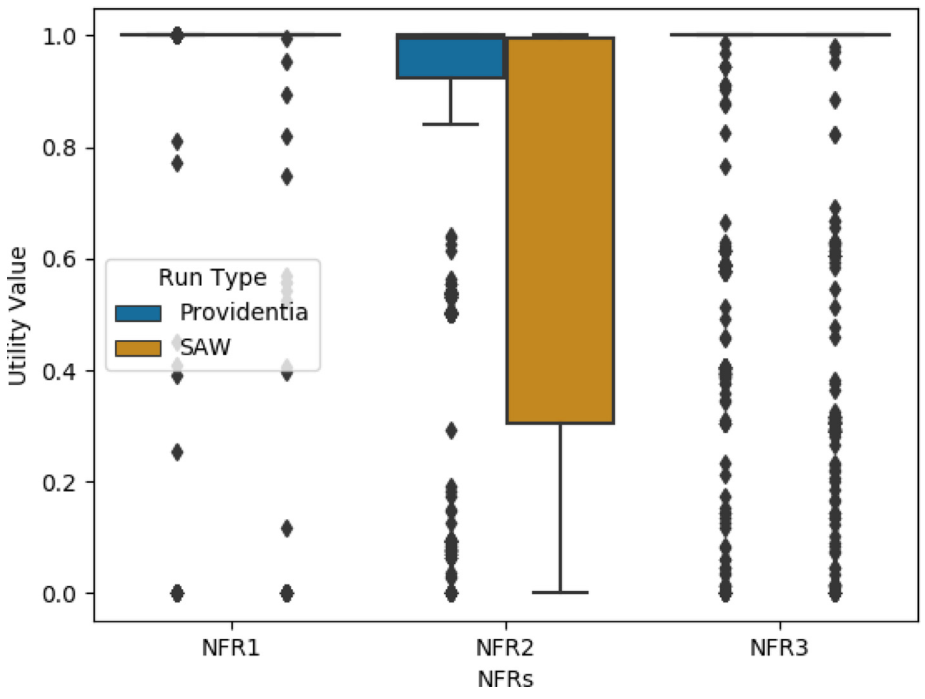


Fig. 15. Comparison of NFR fitness values between Providentia and Providentia-SAW experiments for the SVS application.

5. Discussion

This paper has described search-based techniques and their corresponding empirical studies that use quantitative NFR performance information in the SAS feedback loop to support online decision making. To enable this calculation, a set of FRs (either new or existing) are identified from a requirements specification to support quantification of an NFR, where each selected FR is assigned a weight to indicate its relative importance in the satisfaction of the NFR's objectives. `Providentia` was introduced to perform automated optimization of the FR selection and weight definition process, given the large search space that results from this problem. Furthermore, `SAW` was added to `Providentia` to further optimize the weights of the fitness subfunctions that guided the search procedure.

`Providentia` and `Providentia-SAW` determine optimal sets of weights to make the system more robust to uncertainty. For example, the uncertainty introduced in the RDM simulation include the percentage of dropped/delayed messages, the percent chance that a server goes down, of a network link severing, sensor fuzz, and more. Uncertainty introduced in the SVS include a randomized chance of each sensor failing and/or fuzzing, the

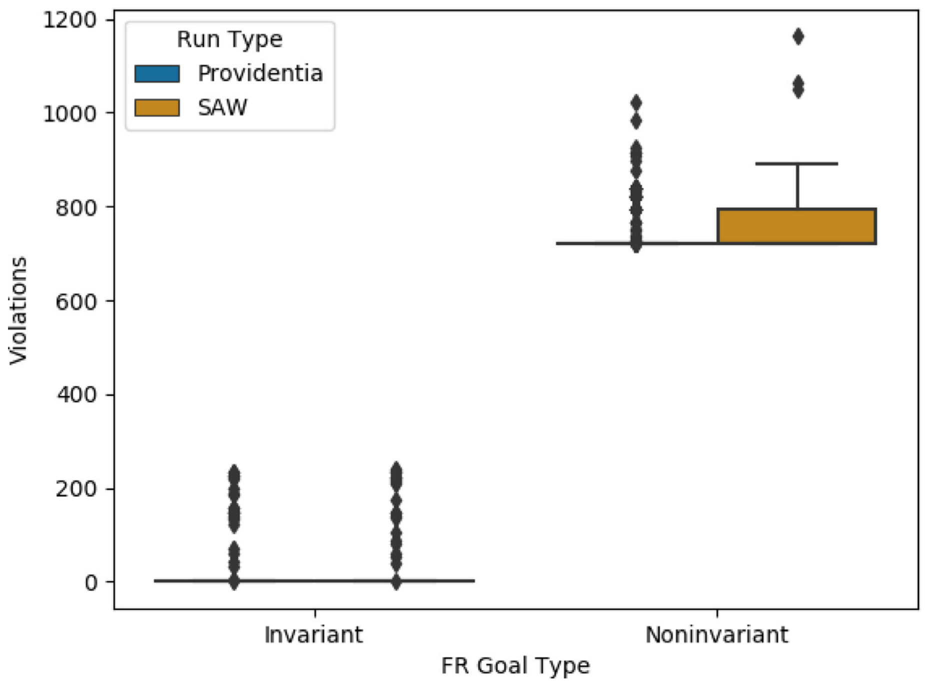


Fig. 16. Comparison of violations between Providentia and Providentia-SAW experiments for the SVS application.

distribution of dirt particles in the room, a downward step in the room, the random instantiation of hazardous objects to the SVS (e.g., liquid, large objects, etc.), and more. The different types of uncertainty faced by the two case studies show the effectiveness and domain independence of `Providentia` and `Providentia-SAW`, as the RDM is from an industrial collaborator and the SVS is based on a real-world system.

Experimental results suggest that the introduction of NFRs into the SAS feedback ecosystem, along with `Providentia`'s automated optimization procedure to determine the configuration of each NFR, can produce an SAS that performs better in terms of overall fitness while significantly reducing the number of requirements violations incurred during execution. These results demonstrate that non-functional objectives, while historically difficult to quantify, can be automatically reconfigured and tuned to enable an SAS to deliver optimal performance in the face of uncertainty.

Fig. 14 shows the comparison of fitness values (i.e., the values calculated in Eq. (8)) for each of the four runs. With the manually-specified weights for the fitness function, `Providentia` performs the best because the GA is able to explore the weights for NFRs *that are separate* from the weights of the fitness function. For manual, random, and `Providentia`, the alpha values in Eq. (8) are all equal. `Providentia-SAW` combines the results from `Providentia` (leading to a higher median fitness), but the search space for an optimal fitness value is explored as the overall fitness function is adjusted every fifth generation.

Table 6
NFR average utility values and standard deviations for SVS application.

NFR	Random	Manual	Providentia	SAW
`NFR1: Minimize [Cost]`	μ: 0.972 σ: 0.114	μ: 0.948 σ: 0.099	μ: 0.957 σ: 0.200	μ: 0.951 σ: 0.210
`NFR2: Minimize [Time]`	μ: 0.564 σ: 0.363	μ: 0.510 σ: 0.014	μ: 0.833 σ: 0.302	μ: 0.773 σ: 0.342
`NFR3: Maximize [Performance]`	μ: 0.489 σ: 0.399	μ: 0.426 σ: 0.072	μ: 0.875 σ: 0.281	μ: 0.886 σ: 0.279

`Providentia-SAW` has led to a greater degree of variance yet further improved the results from standalone `Providentia` (i.e., without `SAW` integration).

Additional results demonstrated the significance of adjusting the weights of the fitness subfunctions that guide `Providentia`'s search procedure. While there was no significant impact to the satisficement of NFR utility values, FR utility values were significantly improved when using `Providentia-SAW`, thereby leading to a significant improvement in overall fitness of both SAS applications that were studied. This result suggests that, as was previously discovered (Fredericks et al., 2014), a linear-weighted fitness function is subject to the uncertainties imposed by each individual environment and that the statistically-specified weights may not generate optimal solutions for all environments considered. As a result, running a hyper-heuristic optimizer can significantly improve fitness in uncertain environments.

In cyber-physical systems, modeling real-world uncertainty is an ongoing problem as simulations often cannot include unexpected issues that in reality a system may face. Such a system would benefit from `Providentia`-optimized NFRs as well as `Providentia-SAW` to balance NFR/FR/adaptation utility functions in unforeseen circumstances.

5.1. Threats to validity.

This paper has presented an extended proof of concept to demonstrate that quantifying NFRs at run time can support the reconfiguration engine in an SAS to improve overall requirements satisficement and minimize violations. As such, we have identified the following internal and external threats to validity for this research.

Internal. First, the derivation of the requirements for both the RDM and SVS applications was manually performed and may not be wholly inclusive of all possible requirements. Moreover, each application was simulated based on the executable specifications and may not exhaustively capture all of the detailed requirements. The manual selection of each FR set to support an NFR is another threat, as the selection may either be too limited or broad as it relies on domain knowledge, which is a common problem in general when identifying NFRs (Yu, 1997; Mylopoulos et al., 1992).

External. External threats include impacts to the system as a result of unanticipated environmental conditions, unexpected human interaction with systems under execution, and unplanned for changes to the system requirements, thereby invalidating prior optimizations discovered by `Providentia` and `Providentia-SAW`.

Construct. Construct threats include scalability and generalizability concerns. In terms of scalability, optimization heuristics traditionally suffer from a larger search space. For this paper, the size of the requirements specifications for both the RDM and SVS case studies may be considered small (i.e., 23 FRs and 7 NFRs for the RDM, and 19 FRs and 3 NFRs for the SVS), and as such, the results may not necessarily generalize to larger search spaces. In terms of generalizability, we have demonstrated `Providentia` and `Providentia-SAW` in two application domains: a networking application and a cyber-physical system, both of which are modeled as SASs. As such, it is possible that our techniques do not generalize to non-SAS domains and can be considered as future work for the authors.

We also only explored the genetic algorithm as a search heuristic. As such, other applicable search techniques, such as multi-objective optimization (Deb et al., 2002), could be used to discover ideal or more globally-optimal solutions. Therefore, an additional threat to validity lies in the search technique, and we plan to explore other such search heuristics in future work.

We next describe related work that span a number of complementary areas to further highlight our contributions.

6. Related work

This section presents related work with regard to self-adaptive systems, obstacle mitigation and requirement satisficement, goal modeling, and NFRs.

Self-adaptive systems. Self-adaptive capabilities are generally expensive to build, difficult to modify, and are usually specific to a given application (Garlan et al., 2004). The Rainbow framework generalizes an SAS such that it can be reused in different systems, separating the self-adaptive control infrastructure from the system itself (Garlan et al., 2004). This separation enables use in legacy systems, localization of problems in separate modules, and software reuse. Rainbow is an architecture-based modeling technique while `Providentia` extends GORE at a higher level. Cheng et al. (2009b) also presented the use of utility functions in SASs with quality NFRs such as performance, cost, and content fidelity. Rather than introducing new metrics to measure quality attributes in a separate architecture, `Providentia` extends the preexisting goal model of FRs and uses metrics already defined by the system to measure NFRs. The use of predefined metrics saves both time and space when performing reconfigurations at run time.

Aceituna and Do (2015) presented a model to determine if, based on a given requirements model, an SAS can be put into undesired states. Although this model can be useful in evaluating FRs, `Providentia` operates under the assumption that the FRs keep the system in an acceptable state. Moreover, `Providentia` focuses on guiding the SAS behavior based on FRs. Should the NFRs introduced with `Providentia` fail, the system will remain in a functionally-valid, yet less optimal, state.

Bencomo and Belaggoun (2014) use a Bayesian definition of surprise to measure the degrees of uncertainty that cause a self-adaptive system to deviate from expected behavior. Their approach uses dynamic decision networks that use probability to determine the satisficement of an NFR based on a system's decision. Initial probabilities are either estimated or derived based on past statistical performance. `Providentia` uses utility functions to determine requirement/goal satisficement and violations, rather than probability, to measure uncertainty.

A recent direction for SASs involves Complex Event Processing (CEP) systems (Weisenburger et al., 2017). A CEP system analyzes event streams and detects specific events or patterns.Weisenburger et al. (2017) address the difficulties in making CEP systems self-adaptive and the need to identify conditions that trigger adaptations. CEP systems can directly benefit from `Providentia` as CEP systems typically do not allow developers to specify metrics on quality attributes. `Providentia` provides quantifications on ambiguous goals and requirements that can be optimized in SASs and therefore can be used to evaluate CEP systems to determine the specific conditions that require an adaptation.

In addition to the numerous approaches for representing NFRs in goal models and SASs, Aspect-Oriented Requirements Engineering presents an approach to identify and specify cross-cutting concerns in separate modules, or aspects (Rashid et al., 2002). Yu et al. (2004) demonstrate that aspects can be identified in goal-oriented requirement analysis using both FRs and NFRs. Similar to `Providentia`, Gray *et al.* elevate cross-cutting concerns to be represented with FRs in a goal model, in contrast to `Providentia` that includes NFRs (Gray et al., 2003). As shown in the performance NFR of the RDM case study, cross-cutting concerns are prevalent in generic NFRs. DeVries and Cheng use evolutionary computation to automatically detect unwanted feature interactions (DeVries and Cheng, 2018), in contrast to `Providentia` and `Providentia-SAW` that introduce additional feature interactions by adding new non-functional requirements to a goal model. Bisbal and Cheng also explore unwanted feature interactions due to non-functional conflicts due to shared resources (Bisbal and Cheng, 2004). Note that `Providentia` and `Providentia-SAW` do not add unwanted feature interactions but rather evaluate the feature interactions between FRs with NFRs.

Obstacles and requirements. Obstacle mitigation is a strategy for identifying and resolving obstacles to goal satisfaction. van Lamsweerde (2009) and van Lamsweerde and Letier (2000) have described a set of strategies for obstacle mitigation, however this approach does not specify to what degree of non-satisfaction that NFRs can become without impacting or degrading the overall system. `Providentia` can be used to supplement these strategies by extending the non-functional goal model in KAOS, and moreover, automatically optimize the FR/NFR weighting scheme.

Requirements monitoring is an approach for quantifying requirements at run time for use in detecting and mitigating obstacles as the system executes, including a monitoring framework developed by Feather et al. (1998). Sawyer et al. (2010) have posited that requirements can be promoted to live run-time entities for use in self-adaptation feedback loops, with a notable example being the SAS MAPE-K feedback loop (Kephart and Chess, 2003). However, these approaches mainly focus on FRs. `Providentia`-optimized NFRs are also intended to be used in adaptation decisions at run time, supported by a set of well-defined FRs and their accompanying utility functions.

Goal modeling. Many approaches similar to `Providentia` use goal modeling to address dependencies between FRs (Nagel et al., 2013) or represent NFRs as soft goals (Yu, 1997; Giorgini et al., 2005). Other approaches use probabilistic methods to improve NFR/FR satisficement (Cailliau and van Lamsweerde, 2017; Paucar and Bencomo, 2016) or optimize SAS satisficement (Letier and van Lamsweerde, 2004; Yang et al., 2017). However, `Providentia` focuses solely on NFR/FR dependencies to optimize FR and NFR satisficement in an SAS without prior knowledge of system performance that most probabilistic methods require. Furthermore, `Providentia` does not use early-phase requirements engineering or high-level abstraction (Dalpiaz et al., 2013; Mylopoulos et al., 1992), but rather focuses on a run-time model used by an SAS.

Non-functional requirements. Other techniques have been introduced to quantify NFRs, generally representing NFRs as soft goals (Yrjönen and Merilinna, 2009; Kobayashi et al., 2016; Yamamoto, 2015). Although both the RDM and SVS case studies use the KAOS goal modeling framework, Providentia is independent of any framework (e.g., NFR Framework, iStar, and KAOS). In contrast to modeling NFRs as soft goals, the weighted approach enables greater flexibility for an SAS to use in finding an optimal reconfiguration strategy at run time. Salehie *et al.* use a Goal-Action-Attribute Model (GAAM) and an automated weighting scheme called Analytic Hierarchy Process to prioritize NFRs (Salehie and Tahvildari, 2012). However, priorities may shift due to uncertainty and requirement interactions at run time. Therefore, Providentia uses a genetic algorithm to optimize goal and weight selection instead of prioritization to make the goal model more robust to uncertainty at run time. Contributing work has decomposed NFR behaviors into monitored patterns rather than explicit requirements in a goal model (Supakkul et al., 2010) and used quantifiable metrics to represent NFRs separately from the FR goal model Sykes et al. (2010). Providentia monitors requirements at run time and does not separate NFRs from the goal model of FRs, as separating NFRs and FRs may prevent the requirements engineer to identify cross-cutting concerns in NFRs.

7. Conclusion

This paper described Providentia, a design-time approach to automatically quantify NFRs at run time. Providentia uses a genetic algorithm to determine an optimal weighting scheme of FRs to describe each NFR, where optimal results yield the highest overall fitness. To illustrate the effectiveness of Providentia, we used an industry-provided RDM application that distributes messages across a network while experiencing random sources of uncertainty. We extended previous work (Bowers et al., 2018) to use a second case study to evaluate the effectiveness of the technique in a different domain.

To further improve Providentia, we incorporated SAW to optimize the weighting scheme of the fitness subfunctions for both case studies. Experimental results suggest that the Providentia-optimized systems are more robust against system and environmental uncertainty and fulfills its requirements to a higher degree when compared to the systems without Providentia.

Future directions for this research include extending the Providentia search procedure to execute at run time, applying both techniques to a real-world cyber-physical system, and evaluating other search techniques that may better handle competing concerns in fitness calculation than a linear-weighted sum (e.g., multi-objective optimization). We also intend to work towards open-sourcing the RDM and SVS applications, respectively.

Acknowledgments

This work has been supported in part by grants from the NSF (CNS-1657061, CNS-1305358, and DBI-0939454), the Michigan Space Grant Consortium, the Comcast Innovation Fund, Oakland University, Ford Motor Company, General Motors Research, the Air Force Research Laboratory (AFRL) under agreement numbers FA8750-16-2-0284 and FA-8750-19-2-0002, and Michigan State University. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and do not necessarily represent the opinions of the sponsors.

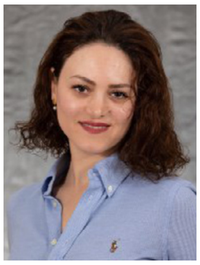
References

Aceituna, D., Do, H., 2015. Exposing the susceptibility of off-nominal behaviors in reactive system requirements. In: Proceedings of the IEEE International Conference on Requirements Engineering (RE). IEEE, pp. 136–145.
Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 1–10.
Bencomo, N., Belaggoun, A., 2013. Supporting decision-making for self-adaptive systems: from goal models to dynamic decision networks. In: Proceedings of the Requirements Engineering: Foundation for Software Quality Companion. Springer, pp. 221–236.
Bencomo, N., Belaggoun, A., 2014. A world full of surprises: Bayesian theory of surprise to quantify degrees of uncertainty. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 460–463.
Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E., 2010. Requirements reflection: requirements as runtime entities. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. ACM, Cape Town, South Africa, pp. 199–202.
Bisbal, J., Cheng, B.H.C., 2004. Resource-based approach to feature interaction in adaptive software. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self–Managed Systems. ACM, pp. 23–27.
Bowers, K.M., Fredericks, E.M., Cheng, B.H.C., 2018. Automated optimization of weighted non-functional objectives in self-adaptive systems. In: Proceedings of the International Symposium on Search Based Software Engineering (SSBSE 2017). Springer, pp. 182–197.
Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., Schulenburg, S., 2003. Hyper--heuristics: An emerging direction in modern search technology. In: Glover, F., Kochenberger, G.A. (Eds.), Handbook of Metaheuristics, Vol. 57. Springer US, pp. 457–474.
Cailliau, A., van Lamsweerde, A., 2017. Runtime monitoring and resolution of probabilistic obstacles to system goals. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE Press, pp. 1–11.
Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J., 2009. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin, Heidelberg, pp. 468–483.
Cheng, S.-W., Garlan, D., Schmerl, B., 2009. Evaluating the effectiveness of the rainbow self-adaptive system. In: Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems SEAMS, pp. 132–141.
Chua Chow, C., Sarin, R.K., 2002. Known, unknown, and unknowable uncertainties. Theory Decis. 52 (2), 127–138.
Chung, L., Nixon, B., Yu, E., Mylopoulos, J., 2000. Non-functional requirements. Softw. Eng.
Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., 2012. Non-Functional Requirements in Software Engineering. Springer Science & Business Media.
Craenen, B., Eiben, A.E., 2001. Stepwise adaption of weights with refinement and decay on constraint satisfaction problems. In: Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation. Morgan Kaufmann Publishers Inc., pp. 291–298.
Dalpiaz, F., Borgida, A., Horkoff, J., Mylopoulos, J., 2013. Runtime goal models: Keynote. In: Proceedings of the IEEE Seventh International Conference on Research Challenges in Information Science (RCIS). IEEE, pp. 1–11.
Dardenne, A., Van Lamsweerde, A., Fickas, S., 1993. Goal-directed requirements acquisition. Science Comput. Programm. 20 (1), 3–50.
Deb, K., Pratap, A., Agarwal, S., Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evolut. Comput. 6 (2), 182–197.
van der, H.K., 1996. Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Leiden University.
Eiben, A.E., van der Hauw, J.K., 1997. Solving 3-SAT by GAs adapting constraint weights. In: Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC'97). IEEE, pp. 81–86.
Eiben, A.E., van der Hauw, J.K., 1998. Adaptive penalties for evolutionary graph coloring. In: Artifical Evolution. Springer, pp. 95–106.
Esfahani, N., Kouroshfar, E., Malek, S., 2011. Taming uncertainty in self-adaptive software. In: Proceedings. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, pp. 234–244. doi:10.1145/2025113.2025147.
Feather, M.S., Fickas, S., Van Lamsweerde, A., Ponsard, C., 1998. Reconciling system requirements and runtime behavior. In: Proceedings of the Ninth International Workshop on Software Specification and Design, 1998, pp. 50–59.
Fredericks, E.M., DeVries, B., Cheng, B.H.C., 2014. AutoRELAX: automatically RELAXing a goal model to address uncertainty. Empir. Softw. Eng. 19 (5), 1466–1501.
deGrandis, P., Valetto, G., 2009. Elicitation and utilization of application-level utility functions. In: Proceedings of the 6th International Conference on Autonomic Computing. ACM, pp. 107–116.
Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P., 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37 (10), 46–54.
Giorgini, P., Mylopoulos, J., Sebastiani, R., 2005. Goal-oriented requirements analysis and reasoning in the tropos methodology. Eng. Appl. Artif. Intell. 18 (2), 159–171.

Gray, J., Bapty, T., Neema, S., Schmidt, D., Gokhale, A., Natarajan, B., 2003. An approach for supporting aspect-oriented domain modeling. In: Generative Programming and Component Engineering. Springer, pp. 151–168.
Holland, J.H., 1992. Adaptation in Natural and Artificial Systems. MIT Press, Cambridge, MA, USA.
Ji, M., Veitch, A., Wilkes, J., 2003. Seneca: Remote mirroring done write. In: Proceedings of the USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA, pp. 253–268.
Keck, D.O., Kuehn, P.J., 1998. The feature and service interaction problem in telecommunications systems: a survey. IEEE Trans. Softw. Eng. 24 (10), 779–796.
Keeton, K., Santos, C., Beyer, D., Chase, J., Wilkes, J., 2004. Designing for disasters. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies. USENIX Association, Berkeley, CA, USA, pp. 59–62.
Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36 (1), 41–50.
Kobayashi, N., Morisaki, S., Atsumi, N., Yamamoto, S., 2016. Quantitative non functional requirements evaluation using softgoal weight. J. Internet Serv. Inf. Secur. 6 (1), 37–46.
Kumari, A.C., Srinivas, K., 2013. Software module clustering using a fast multi-objective hyper-heuristic evolutionary algorithm. Int. J. Appl. Inf. Syst. 5 (6), 12–18.
van Lamsweerde, A., 2009. Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley.
van Lamsweerde, A., Letier, E., 2000. Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Softw. Eng. 26 (10), 978–1005.
Letier, E., van Lamsweerde, A., 2004. Reasoning about partial goal satisfaction for requirements and design engineering. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, pp. 53–62.
Li, Y., Chen, J., Feng, L., 2013. Dealing with uncertainty: a survey of theories and practices. IEEE Trans. Knowl. Data Eng. 25 (11), 2463–2482.
McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C., 2004. Composing adaptive software. Computer 37 (7), 56–64.
Mylopoulos, J., Chung, L., Nixon, B., 1992. Representing and using nonfunctional requirements: a process-oriented approach. IEEE Trans. Softw. Eng. 18 (6), 483–497.
Nagel, B., Gerth, C., Post, J., Engels, G., 2013. Kaos4SOA-extending KAOS models with temporal and logical dependencies. In: Proceedings of the CAiSE Forum, pp. 9–16.
Neema, S., Bapty, T., Scott, J., 1999. Development environment for dynamically reconfigurable embedded systems. In: Proceedings of the International Conference on Signal Processing Applications and Technology. Orlando, FL.
Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L., 1999. An architecture-based approach to self-adaptive software. Intell. Syst. Appl. IEEE 14 (3), 54–62.
Paucar, L.H.G., Bencomo, N., 2016. The reassessment of preferences of non-functional requirements for better informed decision-making in self-adaptation. In: Proceedings of the IEEE International Requirements Engineering Conference Workshops (REW). IEEE, pp. 32–38.
Ramirez, A., Cheng, B., 2011. Automatically deriving utility functions for monitoring software requirements. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems Conference. Wellington, New Zealand, pp. 501–516.
Ramirez, A.J., Knoester, D.B., Cheng, B.H.C., McKinley, P.K., 2009. Applying genetic algorithms to decision making in autonomic computing systems. In: Proceedings of the 6th International Conference on Autonomic Computing, pp. 97–106. (Best paper award)
Rashid, A., Sawyer, P., Moreira, A., Araújo, J.a., 2002. Early aspects: A model for aspect-oriented requirements engineering. In: Proceedings of the IEEE Joint International Conference on Requirements Engineering. IEEE, pp. 199–202.
Salehie, M., Tahvildari, L., 2012. Towards a goal-driven approach to action selection in self-adaptive software. Softw. Pract. Exper. 42 (2), 211–233.
Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A., 2010. Requirements-aware systems: A research agenda for RE for self-adaptive systems. In: Proceedings of the 18th IEEE International Requirements Engineering Conference (RE), pp. 95–103.
Supakkul, S., Hill, T., Chung, L., Tun, T.T., do Prado Leite, J.C.S., 2010. An NFR pattern approach to dealing with NFRs. In: Proceedings of the 18th IEEE International Requirements Engineering Conference (RE). IEEE, pp. 179–188.
Sykes, D., Heaven, W., Magee, J., Kramer, J., 2010. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In: Proceedings of the ACM Symposium on Applied Computing. ACM, pp. 431–438.
DeVries, B., Cheng, B.H.C., 2018. Automatic detection of feature interactions using symbolic analysis and evolutionary computation. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 257–268.
Walsh, W., Tesauro, G., Kephart, J., Das, R., 2004. Utility functions in autonomic systems. In: Proceedings of the First IEEE International Conference on Autonomic Computing. IEEE Computer Society, pp. 70–77.
Weisenburger, P., Luthra, M., Koldehofe, B., Salvaneschi, G., 2017. Quality-aware runtime adaptation in complex event processing. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE Press, pp. 140–151.
Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M., 2009. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 17th IEEE International Requirements Engineering Conference (RE'09), pp. 79–88.
Yamamoto, S., 2015. An approach for evaluating softgoals using weight. In: Proceedings of the Information and Communication Technology. Springer, pp. 203–212.
Yang, Z., Jin, Z., Li, Z., 2017. Achieving adaptation for adaptive systems via runtime verification: a model-driven approach. arXiv:1704.00869.
Yrjönen, A., Merilinna, J., 2009. Extending the NFR framework with measurable nonfunctional requirements. In: Proceedings of the NFPinDSML@ MoDELS.
Yu, E.S.K., 1997. Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of the Third IEEE International Symposium on Requirements Engineering, pp. 226–235.
Yu, Y., Leite, J.C., Mylopoulos, J., 2004. From goals to aspects: discovering aspects from requirements goal models. In: Proceedings of the IEEE International Requirements Engineering Conference. IEEE, pp. 38–47.

Kate M. Bowers is a Ph.D. student in the Department of Computer Science and Engineering at Oakland University. Her research interests include cyber-physical systems, selfadaptive systems, software engineering, and embedded cybersecurity. Contact her at kmlabell@oakland.edu.

Erik M. Fredericks is an Assistant Professor in the Department of Computer Science and Engineering at Oakland University. Dr. Fredericks received his Ph.D. degree in Computer Science and Engineering from Michigan State University in 2015. His research interests include search-based software engineering, self-adaptive systems, model-driven engineering, and cyber-physical systems. Contact him at fredericks@oakland.edu.



Reihaneh H. Hariri is a Ph.D. student in the Department of Computer Science and Engineering at Oakland University. Her research interests include cyber-physical systems, searchbased software engineering, artificial intelligence, and big data. Contact her at rhosseinzadehha@oakland.edu.

Betty H. C. Cheng is a Professor in the Department of Computer Science and Engineering at Michigan State University. Her research interests include model-driven engineering, formal and automated analysis of high-assurance systems, search-based software engineering, and adaptive and autonomic systems. She received a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. Contact her at chengb@cse.msu.edu.