AFRL-RI-RS-TR-2020-121

BUILDING RESOURCE ADAPTIVE SOFTWARE SYSTEMS MODEL-BASED ADAPTATION FOR ROBOTICS SOFTWARE

CARNEGIE MELLON UNIVERSITY

JULY 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2020-121 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
GREGORY HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2020	FINAL TECHNICAL REPORT	NOV 2015 – DEC 2019

4. TITLE AND SUBTITLE

BUILDING RESOURCE ADAPTIVE SOFTWARE SYSTEMS MODEL-BASED ADAPTATION FOR ROBOTICS SOFTWARE

5a. CONTRACT NUMBER
FA8750-16-2-0042

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Jonathan Aldrich

5d. PROJECT NUMBER
BRAS

5e. TASK NUMBER
CM

5f. WORK UNIT NUMBER
UA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 N. Randolph Street
Arlington VA 2203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-121

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Model-Based Adaptation for Mobile Robotics (MARS) Defense Advanced Research Projects Agency (DARPA) Building Resource Adaptive Software Systems (BRASS) effort demonstrated novel techniques to make mobile robotics software so that it can be more easily adapted to different hardware, tasks, and environments. The approach builds software so flexible that it is able to adapt itself to changing environments over a period of years or even decades. The primary intellectual leverage comes from models: formal descriptions of the structure and properties of robotic software, and how these respond to environmental change. Using these models, the developed approach enables robots to automatically explore potential adaptations to the system architecture and code and then choose the adaptation that best meets system objectives in the current environment.

15. SUBJECT TERMS

Mobile Robotics, Collaborative Robots, Intelligent Model-Based Adaptation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN DRAGER**
U	U	U	UU	27	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

1 SUMMARY

Mobile Robotics is a growing area of research, but the long-term deployment of such robots has had limited success. A major challenge in the field is that robotics software is often purpose built for a given robot, task, and environment. Adapting that software to accommodate even small changes in the robot's hardware, mission, or ecosystem is difficult. At Carnegie Mellon University (CMU), Collaborative Robots (CoBots) have been assisting humans for the last seven plus years by escorting visitors, delivering messages, and carrying out other tasks. But getting these robots to perform as intended requires a good deal of human effort. So the CoBots remain mostly confined to the controlled environment of the Computer Science building.

Under the Defense Advanced Research Projects Agency (DARPA) Building Resource Adaptive Software Systems (BRASS) program, the Model-Based Adaptation for Robotics Software (MARS) project (http://www.cs.cmu.edu/~brassmars) has changed the way mobile robotics software is made so that it can be more easily adapted to different hardware, tasks, and environments. We are building software so flexible that it can adapt itself to changing environments over a period of years or even decades. Our primary intellectual leverage comes from models: formal descriptions of the structure and properties of robotic software, and how these respond to environmental change. Using these models, our approach enables robots to automatically explore potential adaptations to the system architecture and code and then choose the adaptation that best meets system objectives in the current environment.

2 INTRODUCTION

When a robot is in service for years, it is likely to change in many ways. Sensors are replaced or removed, software is upgraded, and parts show wear and tear. All of these changes affect how the robot senses the world, makes decisions, moves, reasons, and interacts with its environment. Meanwhile, as time goes on, the robot may have to operate in new environments and be expected to perform new tasks.

These changes require time-consuming manual adaptation in today's mobile robotics systems. For example, the person installing or updating a sensor on a robot must learn how that new sensor uses system resources, such as power, and how the data it provides affect downstream components. Today, this requires painstaking data gathering and parameter tuning, much of which is done manually. Upgrading individual software components—not to mention software frameworks, such as the Robotic Operating System (ROS)—may result in incompatibilities that prevent the software from running correctly, consequently requiring extensive programmer time to debug the system.

While developing and maintaining our CoBots, we have observed that addressing these issues takes up enormous amounts of time, thus illustrating how this problem is a significant barrier to the more widespread deployment of similar systems.

We have developed intelligent model-based adaptation, an approach in which developer-specified and automatically discovered models are leveraged to enable robotics software to autonomously adapt to changes in the ecosystem. Our models capture the intent of the system and its components at a higher level of abstraction compared to the source code. For example, a model of software architecture represents the high-level decomposition of the system into components and captures how those components communicate, providing a tool for reasoning through system-level adaptation. Other models represent the system and its environment, state-machine control, triggers for replanning, utility, system behavior, and the current plan.

Once models have been specified by developers or automatically discovered via observations of the system, we can leverage them in a monitoring component that detects the need for adaptation, then use them to search intelligently among possible adaptations and select an appropriate adaptation for a given change in the environment. Our adaptation approaches include replanning and architectural adaptations as well as code-level adaptations.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Sensitivity Analysis

Currently, environmental changes typically require time-consuming manual reconfiguration or tuning; we would prefer robots to adapt to these changes automatically by discovering models that explain the effects of all sorts of configuration changes in a given environment. These models would then be applied to adapt the software, e.g., by choosing among algorithms or sensors to find the most effective tradeoff between the quality of service and use of resources.

We apply sensitivity analysis, a technique based on sampling and machine learning, to automatically discover models governing how the robot's resource use and performance depend on its configuration. Model discovery is expensive, particularly in long-lasting robots that may have a large configuration space; e.g., four sensors that have five Boolean configuration options each yield a million combinations. It is not feasible to test all combinations, so we have developed techniques both to learn the effect of individual configuration options and to discover when multiple options interact [1]. We have also reduced learning-related costs via transfer learning, which combines large amounts of data cheaply gathered from simulations with carefully selected data from the real robot to predict how the robot will operate in different situations.

3.2 State Transition Repair

Complex robot tasks are typically modeled as state machines, where each state encapsulates a feedback controller. Transitions between states are triggered when the observed state of the world matches thresholds in a multidimensional parameter space. However, these thresholds are hard to get right, even for experienced roboticists. It is common for parameter values to work in one physical environment but fail in another or to work on one robot but fail with another. In long-running robotic software, it is important for nontechnical users to be able to give the robot feed-back on its behavior—e.g., telling a robotic assistant that it should have asked a human for help earlier instead of wandering the halls for an hour—and have the robot automatically adjust its state transition thresholds accordingly.

We have introduced satisfiability modulo theories (SMT)-based Robot Transition Repair (SRTR) [2], an approach to adjusting the parameters of robot state machines using sparse, partial specifications of corrections from humans. This approach is general to any robot action as long as it fits the model of a robot finite-state machine. During implementation, we log execution traces of the transition function. Afterward, the human corrects one or more transitions. SRTR then takes the transition function source code and the corrections as input and then produces adjustments to the parameter values—automatically isolating parameters

that are repairable and translating a set of resulting constraints to an off-the-shelf maximum satisfiability modulo theories (MaxSMT) solver that produces adapted parameter values.

3.3 Learning Triggers for Task Switching

Our CoBot robotic assistants can autonomously deliver messages and escort visitors while avoiding obstacles. However, they cannot respond to their surroundings, e.g., reporting a problem such as a water spill. Long-lasting service robots will be assigned many new tasks through-out their lifetime. They must learn how to automatically switch among such tasks.

We evaluated a baseline approach to task switching by modeling tasks as a Markov decision process (MDP) with rewards and composing all tasks together into one MDP. This approach becomes computationally intractable as the number of tasks increases, as expected in long-lasting systems, because the state space grows combinatorically. Furthermore, the global MDP approach necessitates that sensors involved in all tasks run constantly, requiring too much energy and processing power.

We developed a new approach that adds the ability to interrupt one task and switch to another based on new observations. With our method, the robot learns policies and rewards for each task, then learns a task-selection policy that incorporates synergies between different tasks [3]. For example, reporting a spill can be done conveniently if it is observed while performing another task that is not urgent. We also learn the most important observations for triggering the interruption of one task and consequent switch to another. Focusing on these narrow stimuli substantially saves on the estimated costs of sensing. Through our stimuli-based task switching, robotics software can easily scale up to more tasks and seamlessly adapt to changes in the environment.

3.4 Architectural Adaptation: Model-Code Integration

A key aspect of our approach is *architectural adaptation*. During run time, architectural adaptation enables the robot to consider a variety of available algorithms and sensors and automatically find and enact a configuration of the system that meets end-user needs. During system development and maintenance, architectural adaptation allows software engineers to make not just low-level code changes, but high level changes to the system architecture quickly and correctly. In either scenario—run time or development time—architectural adaptation requires a model of the system's architecture that is not just traceable to code but also integrated with it, so that changes to the code are reflected in the model and vice versa.

Our approach leverages an architectural description language (ADL) implemented as a domain-specific language embedded within Wyvern, an extensible programming language under development at CMU. This ADL declares the architectural components and connectors in the system, and describes how they are connected together.

For example, a portion of the architecture used in our case study (slightly edited for clarity) is shown below. The `RobotStatePublisher` has an attribute (the component name) as well as ports for communicating with other components. These ports either require or provide functions listed in the corresponding interfaces. The `Ros1Topic` connector type is declared to allow components to communicate using ROS's publish-subscribe mechanism. The architecture instantiates 3 components (the `RobotStatePublisher` and two others, whose declarations are similar) and two `Ros1Topic` connector instances. The `Ros1Topic` connectors are used to communicate information about robot velocity and the current state machine.

```
component RobotStatePublisher
   val name: String
   port tf_pub: requires TFMessageIface
   port tf_static_pub: requires TFMessageIface
   port joint_states_sub: provides JointStateIface

connector ROS1Topic
   val name: String

architecture TurtleBot3Teleop
   components
         TurtleBot3Node tbot
         RobotStatePublisher state
         TurtleBot3TeleopNode teleop

   connectors
         ROS1Topic cmd_vel, joint_states

   attachments
         connect tbot.joint_states_pub and state.joint_states_sub
                with joint_states
         connect tbot.cmd_vel_sub and teleop.cmd_vel_pub with cmd_vel
```

Our approach is designed to enforce consistency between architecture and implementation. We enforce this consistency by ensuring that each component is implemented independently, using only its ports to communicate with the outside world. This independence is enforced using *object capabilities*. In a language that is *capability-safe*, such as the Wyvern language we use, there is no global state: each module can only access state that is passed to it when it is created [4]. Components in the architecture are therefore implemented as modules that can communicate only via port objects that are passed to them when the component is instantiated.

More concretely, in our approach, components are implemented as parameterized modules, where the module parameters are the ports that are required by that component. For example, the `RobotStatePublisher` is implemented by a module with the following header:

```
module def PdwRobotStatePublisher(tf_pub: TFMessageIface,
                    tf_static_pub: TFMessageIface, ros: ros.ROS)
```

The `RobotStatePublisher` requires two ports, `tf_pub` and `tf_static_pub`, and these are passed to the module interface. Since the component is implemented in the capability-safe Wyvern language, all communication with the outside world must be done through these ports, or through ports that the module's implementation itself provides.

Our robots are based on ROS, so the previously described architecture uses a ROS topic connector to pass information between components. The ROS topic connector uses Wyvern's support for compile-time metaprogramming to generate the appropriate communication boilerplate so that the sensor and actuator components can be written without knowing the details of the connector. This allows a connector with different characteristics or even a connector from a later version of ROS to be substituted without affecting the component code.

Our approach promotes loose coupling between components and connectors, which facilitates architectural adaptation. Since all dependencies between components are explicit, the architecture can be changed just by modifying the architecture description shown above, without changing code files at all.

Given that preexisting robotics code is not written with an explicit software architecture, as described above, we developed a technique for architectural discovery for a ROS 1 system and instantiated it as part of a tool called rosdiscover. Our approach operates by performing a static analysis of the ROS system's source code and launch files to determine the nodes in the system and the mechanisms by which they communicate with each other. It produces a YAML (Yet Another Markup Language) file that contains this information. We have provided a utility that converts this output YAML file to Wyvern architecture specifications.

In case studies applying architectural adaptation in the BRASS project challenge problems, we demonstrated that architectural adaptation can be used at development time to convert robotics software from using ROS 1 connectors to ROS 2 connectors with only 4 lines of code (in the example above, this includes changing the occurrences of ROS1Topic to ROS2Topic). This is a change that would be very invasive based on conventional technology. We also demonstrated the ability to use architectural adaptation at run time in order to more flexibly adapt to unanticipated environmental changes.

3.5 Multimodel Integration and Planning

As described, our approach leverages multiple models, including models of the robotic system's tasks, power usage, and physical environment as well as its software architecture. In a long-running system, new models must be added to reflect new software, hardware, and resources as well as new environments and tasks. Such long-lasting systems require a systematic approach to integrating multiple models and using them to reason about how to adapt the robot's configuration and behavior.

In our approach to model integration, we use a translator to project each model to a view in a common language, where the views are checked for consistency [5]. An aggregator then composes this information. Models for task planning and architectural reconfiguration are generated in the input language Probabilistic Symbolic Model Checker (PRISM) for checking [6]. A planner leverages probabilistic model checking together with probabilistic computation tree-logic formulas describing safety properties and utility to produce plans for carrying out individual tasks and for reconfiguring the system when needed. The key benefits of this approach include extensibility (new types of models can be added), generality (the planning mechanism supports an arbitrary number of quantifiable quality dimensions), assurance (the probabilistic planner provides quantitative guarantees about behavior), and automation (system reconfiguration actions and task planning can be directly synthesized from models).

3.6 Adapting Source Code

Many classes of adaptations require changes to source code. Our goal is to automatically generate adapted source code in many situations—e.g., when a new component is added to the system but contains a defect or an interface mismatch compared to a previous version of the component.

Unfortunately, long-lasting robotics software poses challenges of scale and complexity that significantly out- pace the ability of state-of-the-art patch generation techniques, which typically target bugs that can be fixed locally and require a comprehensive repeatable test suite to localize the fix and evaluate repairs. Robotics code tends to have nonlocal effects, making it difficult to identify where a patch should be applied. Furthermore, the requirement to simulate both the robot and its environment makes validation significantly more complex; constructing large, repeatable test suites is rarely feasible, and doubts have been raised about whether typical robotics defects can be found in simulation.

To address these challenges, we have built a test generation framework called Houston (https://github.com/squaresLab/Houston), which captures a robotics system as a black box with an explicit configuration space, a space of possible incoming events, and observable behaviors. Houston's abstractions allow the systematic generation of tests for constructing the models of behavior as well as for guiding the selection of code- level transformations and evaluating their effectiveness. Leveraging this framework, we demonstrated that our techniques can find, verify, and correct many recently found real-world defects in Ardu-based robotics systems using purely software-in-the- loop simulation [7].

4 RESULTS AND DISCUSSION

The primary results of this project are found in three forms: 1) The papers we published and disseminated to the research community, 2) The code and systems we developed, and 3) The simulated adaptations we were able to achieve in the BRASS evaluations conducted by Lincoln Labs, the US Army Ground Vehicle Systems Center (GVSC), and the Southwest Research Institute (SWRI). This section highlights the results found in the published papers, which may be read for in-depth description.

4.1 Belief Space Metareasoning for Exception Recovery

Due to the complexity of the real world, autonomous systems use decision-making models that rely on simplifying assumptions to make them computationally tractable and feasible to design. However, since these limited representations cannot fully capture the domain of operation, an autonomous system may encounter unanticipated scenarios that cannot be resolved effectively. We first formally introduce an introspective autonomous system that uses belief space metareasoning to recover from exceptions by interleaving a main decision process with a set of exception handlers. We then apply introspective autonomy to autonomous driving. Finally, we demonstrate that an introspective autonomous vehicle is effective in simulation and on a fully operational prototype [8].

Autonomous systems have been deployed across many applications, such as autonomous vehicles, search and rescue robots, and space exploration rovers. Simply put, these systems make decisions based on decision-making models that have inherent limitations. For example, a self-driving car may not be capable of driving on poorly marked roads or in heavy rain. Hence, in order to guarantee reliable operation, some restricting assumptions must be satisfied. This reduces the complexity of designing and verifying solutions for efficient planning and execution. However, as a result of incomplete decision-making models, these systems may encounter a wide range of unanticipated scenarios that cannot be resolved optimally, feasibly, or even safely.

A simple approach to ensuring the necessary conditions of normal operation is to place the entire responsibility on the operator deploying the autonomous system. However, although relying on human judgment can improve performance, it is desirable to limit human involvement when the conditions of normal operation are violated. In fact, most of this responsibility should ideally be delegated to the autonomous system. We therefore offer a metareasoning framework that activates secondary decision-making models designed to restore normal operation with or without human involvement given any violation of its necessary conditions.

Despite tremendous progress in metareasoning centered on monitoring and controlling anytime algorithms, there have been few attempts to build autonomous systems that use metareasoning to recover from exceptional situations effectively. Such a system presents many challenges. First, because an unanticipated scenario is not captured by a decision-making model by definition, the model does not have the information needed to resolve that exception. Next, while a decision-making model can be extended to capture a set of unanticipated scenarios, a naïve approach will exponentially grow the complexity of the model with the number of exceptions. This is often infeasible for complex exceptions in real world applications. Finally, since a decision-making model cannot capture every unanticipated scenario, there will always be exceptions that cannot be resolved properly.

Recent work in exception recovery has focused on fault diagnosis—detecting and identifying faults—during normal operation. For instance, many approaches diagnose faults by exploiting methods that use particle filters or multiple model estimation with neural networks. While these approaches detect and identify exceptions, they do not offer a comprehensive framework that can also handle exceptions without human assistance. Building on recent work in fault diagnosis, our goal was to provide an exception recovery framework that detects, identifies, and handles exceptions.

We offer an approach for building introspective autonomous systems that use belief space metareasoning for exception recovery. This approach makes decisions by interleaving decision processes: a main decision process designed for normal operation and a set of exception handlers. As the system completes its task, it activates its decision processes based on a belief over potential exceptions. If its belief suggests normal operation, it executes its main decision process. Otherwise, if its belief indicates exceptional operation, it suspends its main decision process and executes an exception handler. It can also gather information or transfer control to an operator given uncertainty in its belief.

Our key contributions were: (1) a formal definition of an introspective autonomous system and its properties, (2) a framework for profiling decision processes, (3) an application of introspective autonomy to autonomous driving, and (4) a demonstration that an introspective autonomous vehicle is effective in simulation and on a fully operational prototype.

4.2 Automatic Extrinsic Calibration of Depth Sensors with Ambiguous Environments and Restricted Motion

Autonomous mobile robots that use multiple depth sensors to perceive their environments, rely on extrinsic calibration to combine the individual views from each sensor into a single coherent view of the surroundings. Such extrinsic calibration is tedious to perform manually, and requires specific scenes for calibration. Current state of the art automatic approaches do not consider the content of scenes used for calibration, and thus are not robust to partially informative scenes in long-term deployments. We developed Delta-Calibration, an automated

extrinsic calibration technique that takes into account the information in a scene for calibration. Delta-Calibration relies on constrained sensor motion to minimize the effects of desynchronization, and ego-motion estimation from each depth camera to detect significant changes in pose, which we term Delta-Transforms. We derive a solution to the extrinsic calibration using such Delta-Transforms taking into account uncertain axes of motion in the environment, and further infer necessary and sufficient conditions on the Delta-Transforms such that Delta-Calibration results in a unique, non-singular, and numerically stable extrinsic calibration. We presented quantitative and qualitative results demonstrating the effectiveness of Delta-Calibration at computing extrinsic calibration over different arrangements of depth sensors [9].

4.3 Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems

Most software systems provide options that allow users to tailor the system in terms of functionality and qualities. The increased flexibility raises challenges for understanding the configuration space and the effects of options and their interactions on performance and other non-functional properties. To identify how options and interactions affect the performance of a system, several sampling and learning strategies have been recently proposed. However, existing approaches usually assume a fixed environment (hardware, workload, software release) such that learning has to be repeated once the environment changes. Repeating learning and measurement for each environment is expensive and often practically infeasible. Instead, we pursue a strategy that transfers knowledge across environments but sidesteps heavyweight and expensive transfer learning strategies. Based on empirical insights about common relationships regarding (i) influential options, (ii) their interactions, and (iii) their performance distributions, our approach, Learning to Sample (L2S), selects better samples in the target environment based on information from the source environment. It progressively shrinks and adaptively concentrates on interesting regions of the configuration space. With both synthetic benchmarks and several real systems, we demonstrated that L2S outperforms state of the art performance learning and transfer-learning approaches in terms of measurement effort and learning accuracy [10].

4.4 Machine Learning Meets Quantitative Planning: Enabling Self-Adaptation in Autonomous Robots

Modern cyber-physical systems (e.g., robotics systems) are typically composed of physical and software components, the characteristics of which are likely to change over time. Assumptions about parts of the system made at design time may not hold at run time, especially when a system is deployed for long periods (e.g., over decades). Self-adaptation is designed to find reconfigurations of systems to handle such run-time inconsistencies. Planners can be used to find and enact optimal reconfigurations in such an evolving context. However, for systems that are highly configurable, such planning becomes intractable due to the size of the

adaptation space. To overcome this challenge, we developed an approach that (a) used machine learning to find Pareto-optimal configurations without needing to explore every configuration and (b) restricted the search space to such configurations to make planning tractable. We explored this solution in the context of robot missions that need to consider task timeliness and energy consumption. An independent evaluation showed that our approach resulted in high-quality adaptation plans in uncertain and adversarial environments [11].

4.5 Transfer Learning for Performance Modeling of Deep Neural Network Systems

Modern deep neural network (DNN) systems are highly configurable with a large number of options that significantly affect their non-functional behavior, for example inference time and energy consumption. Performance models allow one to understand and predict the effects of such configuration options on system behavior, but are costly to build because of large configuration spaces. Performance models from one environment cannot be transferred directly to another; usually models are rebuilt from scratch for different environments, for example different hardware. Recently, transfer learning methods have been applied to reuse knowledge from performance models trained in one environment to another. We performed an empirical study to understand the effectiveness of different transfer learning strategies for building performance models of DNN systems. Our results show that transferring information on the most influential configuration options and their interactions is an effective way of reducing the cost to build performance models in new environments [12].

4.6 Transfer learning for improving model predictions in highly configurable software

Modern software systems are built to be used in dynamic environments using configuration capabilities to adapt to changes and external uncertainties. In a self-adaptation context, we are often interested in reasoning about the performance of the systems under different configurations. Usually, we learn a black-box model based on real measurements to predict the performance of the system given a specific configuration. However, as modern systems become more complex, there are many configuration parameters that may interact and we end up learning an exponentially large configuration space. Naturally, this does not scale when relying on real measurements in the actual changing environment. We proposed a different solution: Instead of taking the measurements from the real system, we learn the model using samples from other sources, such as simulators that approximate performance of the real system at low cost. We defined a cost model that transformed the traditional view of model learning into a multi-objective problem that not only takes into account model accuracy but also measurement effort as well. We evaluated our cost-aware transfer learning solution using real-world configurable software including (i) a robotic system, (ii) 3 different stream processing applications, and (iii) a non-Structured Query Language (NoSQL) database system. The experimental results demonstrated that our approach can achieve (a) a high prediction accuracy, as well as (b) a high model reliability [13].

4.7 Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE

As software systems grow in complexity and the space of possible configurations increase exponentially, finding the near-optimal configuration of a software system becomes challenging. Recent approaches address this challenge by learning performance models based on a sample set of configurations. However, collecting enough sample configurations can be very expensive since each such sample requires configuring, compiling, and executing the entire system using a complex test suite. When learning on new data is too expensive, it is possible to use Transfer Learning to "transfer" old lessons to the new context. Traditional transfer learning has a number of challenges, specifically, (a) learning from excessive data takes excessive time, and (b) the performance of the models built via transfer can deteriorate as a result of learning from a poor source. To resolve these problems, we proposed a novel transfer learning framework called Bellwether Transfer Learner (BEETLE), which is a "bellwether"-based transfer learner that focuses on identifying and learning from the most relevant source from amongst the old data. Research evaluated BEETLE with 57 different software configuration problems based on five software systems (a video encoder, a SAT solver, a SQL database, a high-performance C-compiler, and a streaming data analytics tool). In each of these cases, BEETLE found configurations that are as good as or better than those found by other state-of-the-art transfer learners while requiring only a fraction (1/7th) of the measurements needed by those other methods. Based on these results, we say that BEETLE is a new high-water mark in optimally configuring software [14].

4.8 On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems

Quality assurance for highly-configurable systems is challenging due to the exponentially growing configuration space. Interactions among multiple options can lead to surprising behaviors, bugs, and security vulnerabilities. Analyzing all configurations systematically might be possible though if most options do not interact or interactions follow specific patterns that can be exploited by analysis tools. To better understand interactions in practice, we analyzed program traces to characterize and identify where interactions occur on control flow and data. To this end, we developed a dynamic analysis for Java based on variability-aware execution and monitor executions of multiple small to medium-sized programs. We found that the essential configuration complexity of these programs is indeed much lower than the combinatorial explosion of the configuration space indicates. However, we also discovered that the interaction characteristics that allow scalable and complete analyses are more nuanced than what is exploited by existing state-of-the-art quality assurance strategies [15].

4.9 MOSAICO: Offline Synthesis of Adaptation Strategy Repertoires with Flexible Trade-Offs

Self-adaptation improves the resilience of software-intensive systems, enabling them to adapt their structure and behavior to run-time changes (e.g., in workload and resource availability). Many of these approaches reason about the best way of adapting by synthesizing adaptation plans online via planning or model checking tools. This method enables the exploration of a rich solution space, but optimal solutions and other guarantees (e.g., constraint satisfaction) are computationally costly, resulting in long planning times during which changes may invalidate plans. An alternative to online planning involves selecting at run time the adaptation best suited to the current system and environment conditions from among a predefined repertoire of adaptation strategies that capture repair and optimization tasks. This method does not incur run-time overhead but requires additional effort from engineers, who have to specify strategies and lack support to systematically assess their quality. We researched Multi-Objective Synthesis of AdaptatIon COllections (MOSAICO), an approach for *offline* synthesis of adaptation strategy repertoires that makes a novel use of discrete abstractions of the state space to flexibly adapt extra-functional behavior in a scalable manner. The approach supports making trade-offs: (i) among multiple extra-functional concerns, and (ii) between computation time and adaptation quality (varying abstraction resolution). Our results show a remarkable improvement on system qualities in contrast to manually-specified repertoires. More interestingly, moderate increments in abstraction resolution can lead to pronounced quality improvements, whereas high resolutions yield only negligible improvement over medium resolutions [16].

The last two decades have seen a continuous growth in the complexity of software-intensive systems, which are increasingly relied on for a wide variety of tasks in different application domains, such as energy, communications, and security. The critical nature of these applications has led to a central concern for their *resilience* in the presence of environmental changes, faults, and attacks.

Autonomic and self-adaptive systems are generally considered to be efficient approaches for engineering *resilient* software systems in a cost-effective manner. Such systems are characterized by the separation of the adaptation concern into a control layer that endows the system with the ability to modify its structure and behavior in response to run-time changes.

Many self-adaptation approaches have shown the effectiveness of employing architectural descriptions for reasoning about the best way of adapting the system by synthesizing adaptation plans *online* via planning or model checking tools. These enable the exploration of a rich solution space and can yield in some cases plans that meet formal guarantees, such as optimality with respect to quality objectives or constraint satisfaction.

However, such guarantees come at a high cost due to the computational overhead of the synthesis process, resulting in planning times that often go beyond the desirable duration of the adaptation cycle. When such situations occur, the resulting plan commonly becomes invalid due to changes in the system and its environment that occur during planning time. Moreover, synthesis does not guarantee a priori the existence of a plan that will satisfy adaptation goals, given a set of assumptions about the environment, potentially leading to situations in which the planning activity does not yield *any* solutions at run time.

To improve on this situation, some recent approaches to self-adaptation draw from the areas of discrete event planning and supervisory control for synthesizing *offline* adaptation behavior that can be reused at run time to gracefully degrade when environment assumptions are broken or recover from run-time failures. These approaches mitigate run-time overhead, but tend to focus exclusively on functional behavior.

Offline approaches that go beyond functional behavior are often grounded in control theory and target tunable variables to produce control strategies for adaptive systems with formal guarantees. Although some of them can handle the satisfaction of multiple objectives they cannot explicitly consider trade-offs among them nor between computation time and quality of the solution obtained, to the best of our knowledge.

Aside from synthesis, a class of alternative approaches to this problem involves selecting at run time an adaptation from a predefined repertoire of strategies that captures repair and optimization tasks. Approaches that select at run time the strategy that is best suited to the current state of the system and its environment incur a negligible run-time overhead compared to online synthesis, and are better equipped than existing approaches based on offline synthesis to reason about trade-offs among multiple extra-functional properties, such as performance, cost, or security [17]; [18]; [19]. Unfortunately, this type of approach demands additional effort from software engineers, who have to write the specification of adaptation strategies [20]; [21]. Moreover, engineers lack support to systematically assess in a scalable manner the quality of the strategy repertoires produced. Hence, the development of a strategy repertoire involves iterative testing and debugging, during which developers have to manually assess aspects such as coverage of the state space (i.e., if there is always an applicable adaptation strategy for all relevant situations that can be given), which strategies get selected under what conditions, or what the impact of those selections on quality objectives is.

Our MOSAICO approach enables *offline* synthesis of adaptation strategy repertoires. Our approach is inspired by a class of approaches that employ discrete abstractions of continuous system dynamics to leverage controller synthesis techniques in the area of supervisory control of discrete-event systems. In a nutshell, our technique consists of discretizing the system/environment state space and synthesizing optimal adaptation plans for the different points of the discrete abstraction via probabilistic model checking. Our approach combines the

best of both worlds by eliminating the run-time overhead of online synthesis while retaining the ability to provide *nearly-optimal* solutions, and reducing the specification effort required from engineers, compared to predefined strategy specification and selection.

To the best of our knowledge, this is the first synthesis approach that can handle trade-offs between computation time and quality of the solution, multiple extra-functional concerns, and provide feedback about the quality of the repertoires generated across different regions of the state space.

We build on work for online synthesis of self-adaptation under uncertainty, which is intended for adaptation in a single point of the state space and does not provide any estimations about the quality or feasibility of adaptation prior to deployment. In contrast, the present approach is intended to synthesize adaptation collections that cover an entire region of the state space, providing feedback about issues such as coverage, constraint violations, or quality of the adaptations. Moreover, our approach overcomes the scalability limitation of earlier work by making novel use of discrete abstractions of the state space. This enables us to: (i) trade-off solution quality for computation time, (ii) parallelize the synthesis process by distributing different regions of the abstraction across different computation nodes, and (iii) transfer the synthesis process from run time to development time, if needed. Our approach fully supports probabilistic reasoning. However, we have only addressed the novel use of discrete abstractions and the way in which they enable time/quality trade-offs here.

We validated our approach using the Rainbow framework for self-adaptation and a Denial-of-Service (DoS) attack scenario in Znn.com, a custom-built web system that mimics a news site with multimedia news articles. Our results show: (i) consistency between synthesized repertoires and design specifications (e.g., adaptations are appropriate to the conditions under which they are chosen), (ii) flexibility in trading off computation time and quality of adaptation, and (iii) substantial improvement in quality by comparison with manually-specified repertoires.

4.10 Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search

Many software systems operate in environments where change and uncertainty are the rule, rather than exceptions. Techniques for self-adaptation allow these systems to automatically respond to environmental changes, yet they do not handle changes to the adaptive system itself, such as the addition or removal of adaptation tactics. Instead, changes in a self-adaptive system often require a human planner to redo an expensive planning process to allow the system to continue satisfying its quality requirements under different conditions; automated techniques typically must re-plan from scratch. We propose to address this problem by reusing prior planning knowledge to adapt in the face of unexpected situations. We present a planner based on genetic programming that reuses existing plans. While reuse of material in genetic

algorithms has recently been successfully applied in the area of automated program repair, we find that naïvely reusing existing plans for self-planning actually results in a loss of utility. Furthermore, we proposed a series of techniques to lower the costs of reuse, allowing genetic techniques to leverage existing information to improve planning utility when replanning for unexpected changes [22].

4.11 Uncertainty Reduction in Self-Adaptive Systems

Self-adaptive systems depend on models of themselves and their environment to decide whether and how to adapt, but these models are often affected by uncertainty. While current adaptation decision approaches are able to model and reason about this uncertainty, they do not consider ways to reduce it. This presents an opportunity for improving decision-making in self-adaptive systems, because reducing uncertainty results in a better characterization of the current and future states of the system and the environment (at some cost), which in turn supports making better adaptation decisions. We proposed uncertainty reduction as the natural next step in uncertainty management in the field of self-adaptive systems. This required both an approach to decide when to reduce uncertainty, and a catalog of tactics to reduce different kinds of uncertainty. We presented an example of such a decision, examples of uncertainty reduction tactics, and described how uncertainty reduction requires changes to the different activities in the typical self-adaptation loop [23].

4.12 SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications

Research in self-adaptive systems often uses web applications as target systems, running the actual software on real web servers. This approach has three drawbacks. First, these systems are not easy and/or cheap to deploy. Second, run-time conditions cannot be replicated exactly to compare different adaptation approaches due to uncontrolled factors. Third, running experiments is time consuming. To address these issues, we present the Simulator for Web Infrastructure and Management (SWIM), an exemplar that simulates a web application. SWIM can be used as a target system with an external adaptation manager interacting with it through its transmission control protocol (TCP)-based interface. Since the servers are simulated, this use case addresses the first two problems. The full benefit of SWIM is attained when the adaptation manager is built as a simulation module. An experiment using a simulated 60-server cluster, processing 18 hours of traffic with 29 million requests takes only 5 minutes to run on a laptop computer. SWIM has been used for evaluating self-adaptation approaches, and for a comparative study of model-based predictive approaches to self-adaptation [24].

4.13 Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation

Proactive latency-aware adaptation is an approach for self-adaptive systems that considers both the current and anticipated adaptation needs when making adaptation decisions, taking into account the latency of the available adaptation tactics. Since this is a problem of selecting adaptation actions in the context of the probabilistic behavior of the environment, Markov decision processes are a suitable approach. However, given all the possible interactions between the different and possibly concurrent adaptation tactics, the system, and the environment, constructing the MDP is a complex task. Probabilistic model checking has been used to deal with this problem, but it requires constructing the MDP every time an adaptation decision is made to incorporate the latest predictions of the environment behavior. We developed Proactive Latency-Aware Self-Adaptation (PLA-SDP), an approach that eliminates runtime overhead by constructing most of the MDP offline. At runtime, the adaptation decision is made by solving the MDP through stochastic dynamic programming, weaving in the environment model as the solution is computed. We also presented extensions that support different notions of utility, such as maximizing reward gain subject to the satisfaction of a probabilistic constraint, making PLA-SDP applicable to systems with different kinds of adaptation goals [25].

4.14 Synthesizing Tradeoff Spaces of Quantitative Guarantees for Families of Software Systems

Designing software in a way that guarantees run-time behavior, while achieving an acceptable balance among multiple quality attributes, is an open problem. Providing guarantees about the satisfaction of the same requirements under uncertain environments is even more challenging. Tools and techniques to inform engineers about poorly-understood design spaces in the presence of uncertainty are needed, so that engineers can explore the design space, especially when tradeoffs are crucial. To tackle this problem, we developed an approach that combines synthesis of spaces of system design alternatives from formal specifications of architectural styles with probabilistic formal verification. The main contribution is a formal framework for specification-driven synthesis and analysis of design spaces that provides formal guarantees about the correctness of system behaviors and satisfies quantitative properties (e.g., defined over system qualities) subject to uncertainty, which is treated as a first-class entity. We illustrated our approach in two case studies: a service-based adaptive system and a mobile robotics architecture. Our results show how the framework can provide useful insights into how average case probabilistic guarantees can differ from worst case guarantees, emphasizing the relevance of combining quantitative formal verification methods with structural synthesis, in contrast with techniques based on simulation and dynamic analysis that can only provide estimates about average case probabilistic properties [26].

5 CONCLUSION

Mobile robotics systems have the potential to play a greater role in society, assisting with tasks in the workplace, on the battlefield, and in the home. But this revolution can only take place if the robots can easily adapt to new environments and maintain themselves with little human effort. Our preliminary results suggest that models are an important technical enabler for the kind of adaptation that will enhance the benefits of mobile robotics systems for society.

6 REFERENCES

[1] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y., "Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis", In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[2] Holtz, Jarrett, Guha, Arjun, Biswas, Joydeep. "Interactive Robot Transition Repair With SMT." In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[3] Mohseni-Kabir, A., Veloso, M., "Robot task interruption by learning to switch among multiple models." In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2018.*

[4] Melicher, D., Shi, Yi., Potanin, A., Aldrich, J. "A capability-based module system for authority control." In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* 2017.

[5] Ruchkin, I., Sunshine, J., Iraci, G., Schmerl, B. and Garlan, D., "IPL: An Integration Property Language for Multi-Model Cyber-Physical Systems," *22nd International Symposium on Formal Methods (FM2018)*, 2018.

[6] PRISM Model Checker, University of Oxford, 2011. http://www.prismmodelchecker.org/

[7] Timperley, Christopher Steven and Afzal, Afsoon and Katz, Deborah S. and Hernandez, Jam Marcos and Le Goues, Claire. "Crashing Simulated Planes is Cheap: Can Simulation Detect Robotics Bugs Early?" In *Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation*, 2018.

[8] Svegliato, Justin, Wray, Kyle, Witwicki, Stefan, Biswas, Joydeep, Zilberstein, Shlomo. "Belief Space Metareasoning for Exception Recovery." In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),* 2019.

[9] Holtz, Jarrett, Biswas, Joydeep. "Automatic Extrinsic Calibration of Depth Sensors with Ambiguous Environments and Restricted Motion." In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[10] Jamshidi, Pooyan, Velez, Miguel, Kästner, Christian, and Siegmund, Norbert. "Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems." In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2018.

[11] Jamshidi, Pooyan, Javier Cámara, Bradley Schmerl, Christian Kästner, and David Garlan. "Machine Learning Meets Quantitative Planning: Enabling Self-Adaptation in Autonomous Robots." In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019.

[12] Iqbal, Md Shahriar, Lars Kotthoff, and Pooyan Jamshidi. "Transfer Learning for Performance Modeling of Deep Neural Network Systems." In *USENIX Conference on Operational Machine Learning (OpML 19)*, 2019.

[13] Jamshidi, Pooyan, Velez, Miguel, Kästner, Christian, Siegmund, Norbert, and Kawthekar Prasad. “Transfer learning for improving model predictions in highly configurable software.” In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2017.

[14] Krishna, Rahul, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. "Whence to Learn? Transferring Knowledge in Configurable Systems using BEETLE.", *IEEE Transactions on Software Engineering* (to appear), 2020.

[15] Meinicke, Jens; Wong, Chu-Pan; Kästner, Christian; Thüm, Thomas; Saake, Gunter. "On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems". In *Proceedings Int'l Conf. Automated Software Engineering (ASE)*, 2016.

[16] Cámara, J., Schmerl, B., Moreno, G.A. and Garlan, D. "MOSAICO: Offline Synthesis of Adaptation Strategy Repertoires with Flexible Trade-Offs," *Automated Software Engineering*, **25(3)**, September 2018.

[17] Cheng, S., Garlan, D., Schmerl, B.R. "Evaluating the effectiveness of the rainbow self-adaptive system." In *Proc. 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2009.

[18] Huber, N., van Hoorn, A., Koziolek, A., Brosig, F., Kounev, S. "Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments." *Serv. Oriented Comput. Appl.* 8(1), 73–89 (2014)

[19] Schmerl, B.R., Cámara, J., Gennari, J., Garlan, D., Casanova, P., Moreno, G.A., Glazier, T.J., Barnes, J.M. "Architecture-based self-protection: composing and reasoning about denial-of-service mitigations." In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security (HotSoS)*, 2014.

[20] Cámara, J., Correia, P., de Lemos, R., Garlan, D., Gomes, P., Schmerl, B. R., Ventura, R.: "Evolving an adaptive industrial software system to use architecture-based self-adaptation." In *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2013.

[21] Cheng, S.-W. "Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation." Carnegie Mellon University PhD thesis, 2008.

[22] Kinneer, C., Coker, Z., Wang, J., Garlan, D. and Le Goues, C., "Managing Uncertainty in Self-Adaptive Systems with Plan Reuse and Stochastic Search," *in Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2018.

[23] Moreno, G.A., Cámara, J., Garlan, D. and Klein, M., "Uncertainty Reduction in Self-Adaptive Systems," *Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2018.

[24] Moreno, G.A., Schmerl, B. and Garlan, D., "SWIM: An Exemplar for Evaluation and Comparison of Self-Adaptation Approaches for Web Applications," *Proceedings of the 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2018. *Best Artifact Paper Award Winner*.

[25] Moreno, G.A., Cámara, J., Garlan, D. and Schmerl, B., "Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation," *ACM Transactions on Autonomous and Adaptive Systems*, Vol. **13(1)**, 2018.

[26] Cámara, J., Garlan, D. and Schmerl, B., "Synthesizing Tradeoff Spaces of Quantitative Guarantees for Families of Software Systems," *Journal of Systems and Software*, Vol. **152**:33-49, June 2019.

7 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ADL	Architectural Description Language
BEETLE	Bellwether Transfer Learner
BRASS	Building Resource Adaptive Software Systems program
CoBot	Collaborative Robot
CMU	Carnegie Mellon University
DARPA	Defense Advanced Research Projects Agency
DoS	Denial of Service
DNN	Deep Neural Network
GVSC	US Army Ground Vehicle Systems Center
Houston	not an acronym, a framework for automated integration-test generation for robotic systems
L2S	Learning to Sample
MARS	Model-Based Adaptation for Robotics Software
MaxSMT	Maximum Satisfiability Modulo Theories
MDP	Markov Decision Process
MOSAICO	Multi-Objective Synthesis of AdaptatIon Collections
NoSQL	non-Structured Query Language
PLA-SDP	Proactive Latency-Aware Self-Adaptation
PRISM	Probabilistic Symbolic Model Checker
Rainbow	not an acronym, a framework for developing self-adaptive systems
ROS	Robotic Operating System
SAT	Satisfiability
SMT	Satisfiability Modulo Theories
SRTR	SMT-based Robot Transition Repair
SWIM	Simulator for Web Infrastructure and Management
SWRI	Southwest Research Institute
TCP	Transmission Control Protocol
Wyvern	not an acronym, an extensible programming language for engineering web and mobile applications
YAML	Yet Another Markup Language