Model-based Adaptation for Robotics Software

Jonathan Aldrich[1], Joydeep Biswas[2], Javier Cámara[3], David Garlan[1], Arjun Guha[2], Jarrett Holtz[2], Pooyan Jamshidi[4], Christian Kästner[1], Claire Le Goues[1], Anahita Mohseni-Kabir[1], Ivan Ruchkin[1], Selva Samuel[1], Bradley Schmerl[1], Christopher Steven Timperley[1], Manuela Veloso[1], and Ian Voysey[1]

[1]*Carnegie Mellon University*
[2]*University of Massachusetts, Amherst*
[3]*University of York*
[4]*University of South Carolina*

We describe *model-based adaptation*, an approach that leverages models of software and its environment in order to enable automated adaptation. The goal of our approach is to build long-lived mobile software systems that can effectively adapt to changes to their environment. An independent evaluation suggests that the approach can be effective in the domain of mobile robotics.

Introduction. Mobile robotics is a growing research area, but one with limited success in long-term robot deployment. A major challenge is that robotics software is often purpose-built for a given robot, tasks, and environment, and adapting that software to even small changes in the robot's hardware, mission, or ecosystem is difficult. CMU's CoBots have been assisting humans at the university for the last 7+ years by escorting visitors, delivering messages, and carrying out other tasks—but they do so only at a high cost in human labor, and remain mostly confined to the controlled environment of the Computer Science building.

The Model-based Adaptation for Robotics Software (MARS) project (http://www.cs.cmu.edu/~brassmars) seeks to fundamentally raise the level of abstraction at which we build mobile robotics software, making it easier to adapt and evolve it. We want to build software that can live and adapt itself to a changing environment 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 vary together in response to environmental change [BFCA14]. Using these models, our approach automatically explores the space of potential adaptations to the system architecture and code, choosing the adaptation that best meets system objectives in the current environment.

Change and Adaptation in Robotics Software. When a robot is in service for years, it is likely to change in many ways: new or removed sensors and actuators, software upgrades, and wear and tear all affect how the robot senses and reasons about the world, and moves and interacts with its environment. The robot may have to operate in new environments and may be expected to perform new tasks.

These changes require time-consuming, manual adaptation in today's mobile robotics systems. For example, installing or updating a sensor on a robot requires learning how that new sensor uses system resources such as power, and how the precision of data it provides affects 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, requiring extensive programmer time to debug.

We have observed that these issues become major sinks of human time in developing and maintaining our CoBots, posing significant barriers to a more widespread deployment of similar systems.

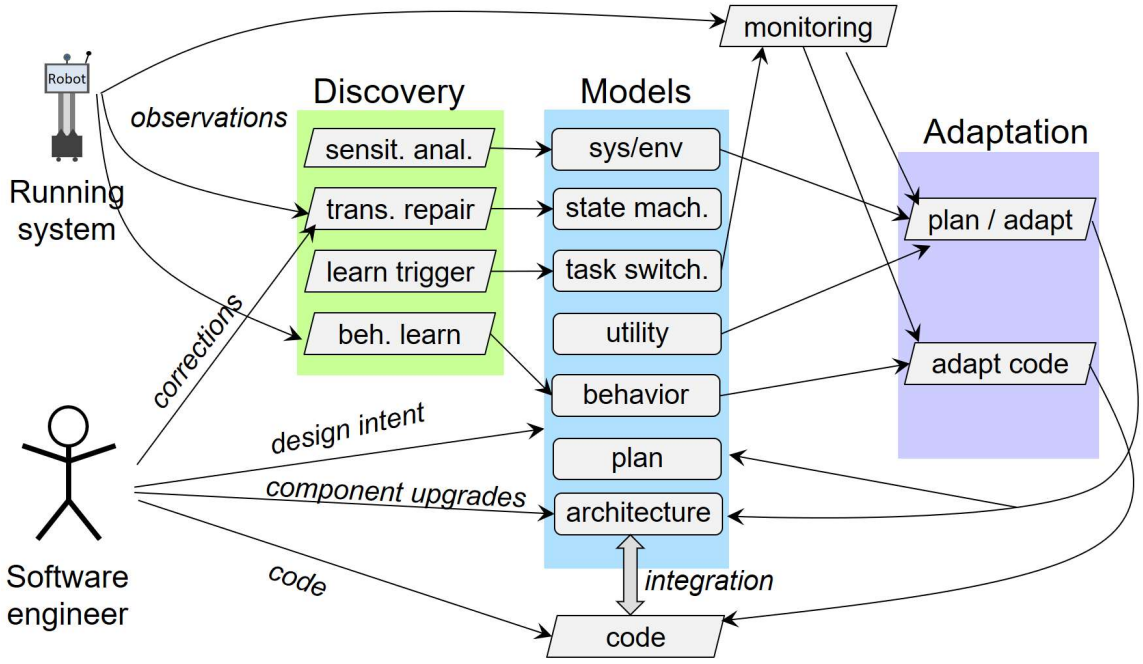

Figure 1. The Intelligent Model-Based Adaptation approach.

Our Approach: Intelligent Model-based Adaptation. We propose Intelligent Model-Based Adaptation (Figure 1), an approach in which developer-specified and automatically-discovered models are leveraged to autonomously adapt robotics software to a changing ecosystem. Our models capture the intent of the system and its components at a higher level of abstraction compared to 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 about 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 discovered automatically from observations of the system, we can leverage them in a monitoring component that detects the need for adaptation, then use them to perform intelligent search through the space of possible adaptations and select an adaptation appropriate to a given change in the environment. Our adaptation approaches include replanning and architectural adaptations, as well as code-level adaptations.

Sensitivity Analysis. The environmental changes described above typically require time-consuming manual reconfiguration or tuning today; we would like robots to automatically *discover* models that explain the effects of all sorts of configuration changes in a given

environment. These models can then be applied to adapt the software, for example choosing among algorithms or sensors to more effectively trade off between quality of service and resource use.

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-lived robots that may have a large configuration space: for example, 4 sensors that have 5 boolean configuration options each yield a million combinations. It is infeasible to test all combinations, so we have developed techniques both to learn the effect of individual configuration options and discover when multiple options interact [JSV+17]. We also reduce learning-related costs via transfer learning, which combines large amounts of data gathered cheaply from simulation with carefully selected data from the real robot to predict how the robot will operate across different situations.

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, but 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 non-technical users to be able to give the robot feedback on its behavior—for example, 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 introduced SMT-based Robot Transition Repair (SRTR) [HGB18], 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 execution, we log execution traces of the transition function. After execution, the human corrects one or more transitions. SRTR then takes as input the transition function source code and the corrections, and produces adjustments to the parameter values—automatically isolating parameters that are repairable and translating a set of resulting constraints to an off-the-shelf MaxSMT solver which produces adapted parameter values.

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 surrounding environment, e.g. reporting a problem such as a water spill. Long-lived service robots will be assigned many new tasks over their lifetime, and must learn how to automatically switch among them.

We evaluated a baseline approach to task switching, 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-lived systems, because the state space grows combinatorially. Furthermore, the global MDP approach requires constantly running the sensors involved in all tasks, requiring too much energy and processing power.

We have developed a new approach, adding the ability to interrupt a task and switch to another task based on new observations. Our approach learns policies and rewards for each

task, then learns a task selection policy that incorporates synergies between different tasks [MV18]. For example, reporting a spill can be done conveniently if it is observed while performing another non-time-critical task. We also learn the most important observations for triggering the interruption of one task and consequent switch to another task. Focusing on these narrow stimuli saves substantially 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 a varying environment.

Architectural adaptation: models and code integration. A key aspect of our approach is architectural adaptation—the ability to consider the variety of algorithms and sensors available, and automatically find and enact a configuration of the system that meets end user needs. This approach requires a model of the system's architecture that is not just traceable to code, but integrated with it, so that changes to the code are reflected in the model and vice versa.

Our approach enforces component isolation, ensuring that components by default cannot communicate with each other or access system resources [MSPA17]. This creates loose coupling between components and facilitates architectural adaptation. An architecture description language specifies how to bind those components together:

```
architecture SenseAct
    component s:Sensor
    component a:Actuator
    connector senseData:ROSTopic
    connect s.out, a.in with senseData
```

Our robots are based on ROS, so the (greatly simplified) architecture above uses a ROS *topic* connector to pass sensor input to an actuator that uses it for further processing. The `ROSTopic` connector generates the appropriate communication boilerplate so that the `Sensor` and `Actuator` components can be written without knowing the details of the connector, allowing a connector with different characteristics or even from a later version of ROS to be substituted without affecting the component code.

Multi-model integration and planning. As described above, 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-lived systems require a systematic approach to integrating multiple models and using them to reason about and adapt the robot's configuration and behavior.

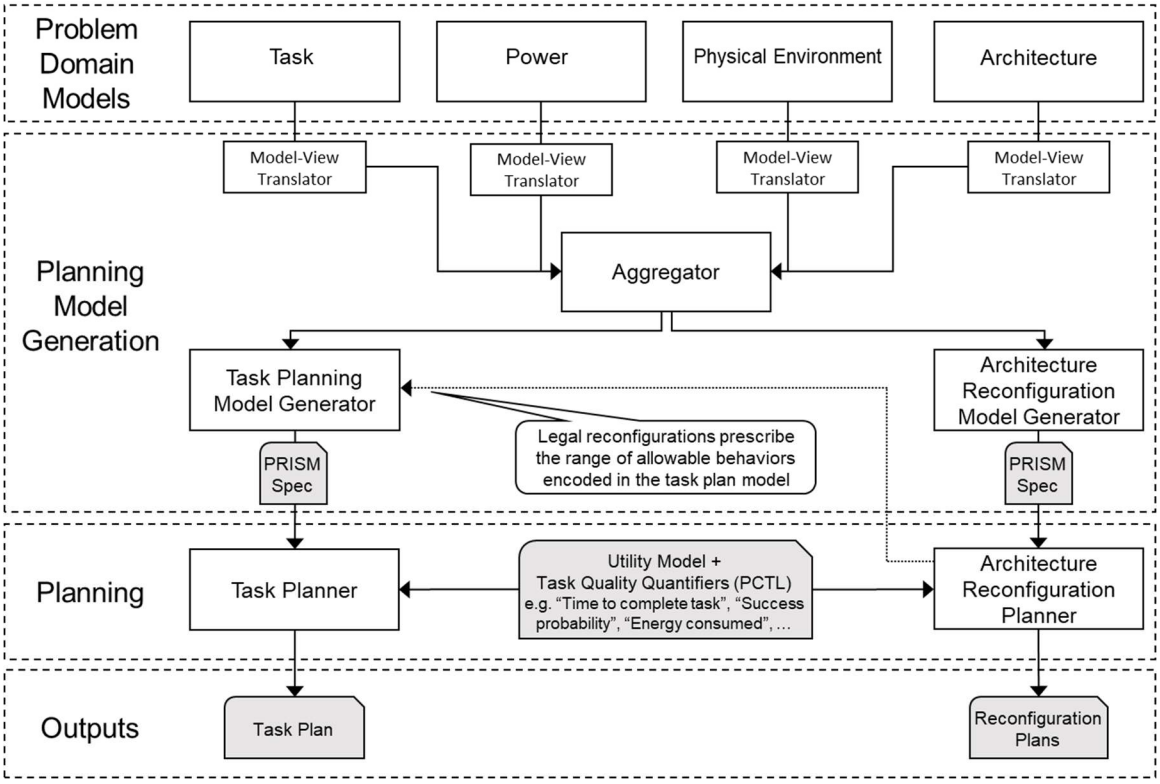

Figure 2. Our approach to model integration.

In our approach to model integration, illustrated in Figure 2, we use a translator to project each model to a view in a common language, where the views are checked for consistency [RSI+18]. An aggregator then composes this information. Models for task planning and architectural reconfiguration are generated in the input language for the PRISM model checker. A planner leverages probabilistic model checking together with probabilistic computation tree logic (PCTL) 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).

Adapting source code. Many classes of adaptations require changes to source code. Our goal is to automatically generate adapted source code in many situations—for example 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-lived robotics software poses challenges of scale and complexity that significantly outpace 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 non-local effects, making it difficult to identify where a patch should be applied. Furthermore, the requirement to simulate both the robot and its environment adds significant complexity to

validation; constructing large, repeatable test suites is often infeasible, 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 in Figure 1 as well as 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 in software-in-the-loop simulation [TAK+18].

Evaluation. In partnership with an independent third-party evaluator, MIT Lincoln Laboratory, we are evaluating how well our approach allows robots to adapt. Our evaluation uses the Gazebo simulator to model a TurtleBot delivering messages on a floor of Wean Hall at CMU. The evaluation uses 3 scenarios, each with a different kind of perturbation which tests our system's ability to adapt:

1. New hardware affects the power usage of the TurtleBot. We use sensitivity analysis to discover a model of power consumption, which is then used by our adaptive planner, e.g. changing the plan to visit charging stations when battery power runs low.
2. An upgraded software component is installed, but it has (seeded, in our simulation) defects that adversely affect behavior. We use our code-level adaptation engine to produce patches that allow the robot to complete its task with the new software component.
3. Environmental changes such as a dark corridor or a blocked path force the robot to adapt its plan. We combine task adaptations (e.g. taking an alternate path) with architectural adaptations (e.g. using different sensors and algorithms that operate better in the new environment) to allow the robot to succeed in its task.

Figure 3 shows the results of our evaluation. In each scenario, our adaptations were able to restore full functionality in response to perturbations in the majority of tests.

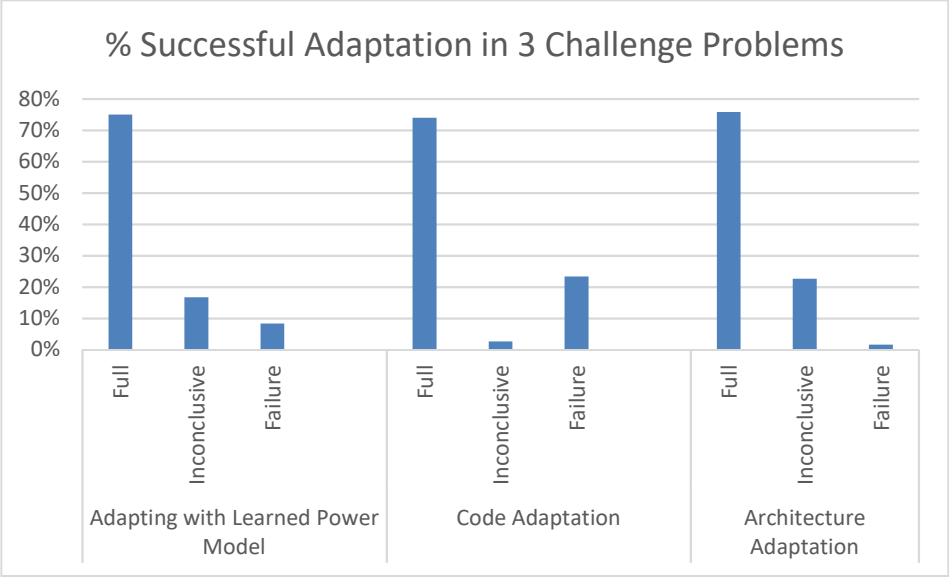

Figure 3. Experimental results for the 3 adaptation scenarios. Full means adaptation enabled normal operation after a perturbation, Inconclusive means we were not able to assess whether adaptation was successful, and Failure means normal operation was not restored.

The Future of Adaptive Robotics Software. Mobile robotics systems have the potential to play an increased role in society, assisting with tasks in the workplace and 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 magnify the benefits of mobile robotics systems to society.

References

[BFCA14] N. Bencomo, R. France, B.H.C. Cheng, U. Aßmann (eds). Models@run.time: foundations, applications, and roadmaps. LNCS 8378, Springer, 2014.

[HGB18] J. Holtz, A. Guha, J. Biswas. Interactive Robot Transition Repair with SMT. IJCAI 2018.

[JSV+17] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, Y. Agarwal, Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. ASE 2017.

[KNP11] M. Kwiatkowska, G. Norman, D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. CAV 2011.

[MSPA17] D. Melicher, Y. Shi, A. Potanin, J. Aldrich. A Capability-Based Module System for Authority Control. ECOOP 2017.

[MV18] A. Mohseni-Kabir, M. Veloso. Robot Task Interruption by Learning to Switch Among Multiple Models. IJCAI 2018.

[RSI+18] I. Ruchkin, J. Sunshine, G. Iraci, B. Schmerl, D. Garlan. IPL: An Integrated Property Language for Multi-Model Cyber-Physical Systems. FM 2018.

[TAK+18] C.S. Timperley, A. Afzal, D.S. Katz, J.M. Hernandez, C. Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? ICST 2018.