Avida-MDE: A Digital Evolution Approach to Generating Models of Adaptive Software Behavior *

Heather J. Goldsby and Betty H.C. Cheng†
{hjg, chengb}@cse.msu.edu
Department of Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, Michigan 48824 USA

ABSTRACT

Increasingly, high-assurance applications rely on autonomic systems to respond to changes in their environment. The inherent uncertainty present in the environment of autonomic systems makes it difficult for developers to identify and model resilient autonomic behavior prior to deployment. In this paper, we propose AVIDA-MDE, a digital evolution approach to the generation of *behavioral models* (i.e., a set of interacting finite state machines) that capture autonomic system behavior that is potentially resilient to a variety of environmental conditions. We use an evolving population of *digital organisms* to generate behavioral models, where the organisms are subjected to natural selection and are rewarded for generating behavioral models that meet developer requirements. To illustrate this approach, we successfully applied it to the generation of behavioral models describing the navigation behavior of an autonomous robot.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; I.6.5 [**Computing Methodologies**]: Simulation and Modeling—*Model Development*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Self-modifying machines*

General Terms

Experimentation.

Keywords

Digital evolution, model-driven engineering, model checking, autonomic computing.

*This work has been supported in part by NSF grants EIA-0000433, CNS-0551622, CCF-0541131, IIP-0700329, CCF-0750787, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Siemens Corporate Research, and a Quality Fund Program grant from Michigan State University.

†Please contact this author for all correspondences.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

1. INTRODUCTION

Increasingly, high-assurance applications rely on autonomic systems to respond to changes in the supporting computing, communication infrastructure, and in the physical environment, while maintaining acceptable behavior [18]; examples include critical infrastructure protection and transportation systems. In an effort to promote separation of concerns, we consider an *autonomic system* to comprise a collection of (non-adaptive) *target systems* and a set of adaptation transitions among target systems in response to environmental changes. Within software engineering communities and industry, there is interest in developing software, including autonomic systems, using model-driven engineering (MDE) [23]. MDE enables developers to "program" at a higher level of abstraction by supporting the systematic transformation of graphical models into more detailed models or formal specifications, and the eventual generation of the corresponding code. In general, the Unified Modeling Language (UML) [1] is frequently used to specify the models that start the MDE process (i.e., initial requirements-level or early design models). Specifically, system structure is captured in a *class diagram*. System behavior is captured in a *behavioral model* comprising a set of communicating finite state machines (one per class), where each state machine is depicted in a *state diagram*.

While MDE is an attractive development approach, the efforts required to create or revise the behavioral model are often error prone and difficult to automate. Autonomic systems potentially amplify this concern because a separate behavioral model must be created/revised for each target system. In this paper, we consider the development of behavioral models, abstractly speaking, to be the search for an appropriate solution from a large space of possible behavioral models. Given the inherent uncertainty present in the environment of autonomic systems, we propose a digital evolution approach to the generation of innovative behavioral models whose behavior may be more resilient than those created manually or through a synthesis algorithm.

We characterize the search space and solution space of generating behavioral models in terms of developer-specified information. Specifically, the *search space* is the set of possible behavioral models that can be generated. It is constrained by the structural elements that define the alphabet used to construct transitions and the existing state diagrams (if any) to be extended. The *solution space* is the set of behavioral models that satisfy the developer's requirements, specified as scenarios and properties. A *scenario* specifies

one path of behavior through a model and a *property* indicates what all paths through a model must satisfy. Thus, scenarios form a lower bound on the solution space by defining the minimal acceptable behavior, and properties form an upper bound on the solution space by defining the maximum acceptable behavior [26]. Although all of the behavioral models within the solution space are guaranteed to support the scenarios and satisfy the properties, their behavior is not necessarily identical. Specifically, the unspecified properties of a given model (e.g., fault-tolerance, obfuscation, readability, size) will likely differ from others in the solution space and make one model better suited for specific environmental conditions. Our objective is to identify multiple behavioral models within the solution space, each of which could represent the behavior of a target system for a particular set of environmental conditions.

In recent years, significant progress has been achieved in generating behavioral models by synthesizing finite state machines from either properties (e.g., [9, 10, 27]), scenarios (e.g., [6, 14, 26, 28]), or both (e.g., [25]). In general, these algorithms identify a single behavioral model within the solution space either using implicit information within the algorithm (e.g., assumptions about naming conventions) or explicit information provided by the user (e.g., state labeling, information regarding scenario interaction). These approaches limit the exploration of the solution space to those envisioned by the developer who hand-crafted the model or designed the synthesis algorithm. As we consider identifying a suite of behavioral models that represent the behavior of multiple target systems of an autonomic system, we see that additional innovation is required.

In this paper, we describe a digital evolution approach to discovering behavioral models that represent the behavior of different target systems and, as such, different possible responses to different environmental conditions. Evolutionary computation methods such as the genetic algorithm (GA) and genetic programming (GP) have achieved considerable success in the world of computing, in some cases producing human-competitive designs [13]. *Digital evolution* [20] is a branch of evolutionary computation in which a population of self-replicating computer programs exists in a user-defined computational environment and is subject to mutations and natural selection. These "digital organisms" compete for available resources (e.g., virtual CPU cycles) that enable the organism to survive and thrive; in addition, these organisms are subject to instruction-level mutations during replication. Organisms remain in the population until they die of either old age or are overwritten by another organism. Whereas GAs and GPs evaluate each individual in the population and explicitly select individuals to move to the next generation, the evolution of digital organisms is more open-ended. Specifically, organisms are asynchronously evaluated; if an organism exhibits desirable behavior, then the relative amount of resources that the organism receives is increased. Because the evaluation is not used to explicitly select organisms to survive, poorly performing organisms may continue to exist in the population and could eventually produce a novel solution. Digital evolution provides a means to *harness* the power of evolution and apply it to problems in science and engineering [17]. Within software engineering, digital evolution provides a means to explore large search spaces and identify solutions that human developers may not discover using traditional techniques. For this investigation, we have used and extended Avida [20], the most widely used digital evolution platform for studying evolution in biology.

In this paper, we present Avida for Model Driven Development (Avida-MDE), which is a digital evolution-based tool for generating behavioral models that adhere to developer requirements specified as scenarios and properties.[1] To create Avida-MDE, we extended Avida in three key ways: First, we enabled a developer to define the search space by providing *instinctual knowledge*, which is information available to an organism at birth. Second, we enabled Avida-MDE organisms to generate behavioral models using their instinctual knowledge. Third, Avida-MDE evaluates an organism based upon how well its generated behavioral model satisfies the developer's requirements.

Overall, Avida-MDE enables developers to automatically generate behavioral models for autonomic systems that meet their requirements and may include additional properties to provide resiliency. We illustrate the Avida-MDE approach by using it to generate behavioral models for an autonomous robot navigation system [11]. The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 provides background on Avida. Section 4 describes our approach, Avida-MDE. Section 5 applies Avida-MDE to generate target systems for an autonomous robot navigation system. Finally, in Section 6, we conclude and discuss future work.

2. RELATED WORK

A behavioral model comprises a set of interacting state diagrams, where a state diagram is a type of finite state machine. A state diagram depicts the *states* in which an object can be found during its lifetime in response to events and internal conditions. A change in state generally changes the way an object behaves. Specifically, a change between two states is represented as a *transition* that may have a *trigger* (or event) that causes the transition to occur, a *guard* that must be met for the transition to take place, and/or one or more *actions*.

Although numerous techniques for generating finite state machines (FSMs) have been proposed within software engineering (e.g., [6, 9, 10, 14, 25, 26, 27, 28]) and evolutionary computation (e.g., [2, 4, 8, 15, 16, 24]), a more limited number [6, 9, 10, 14, 25, 26, 27, 28] address the generation of state diagrams that represent the behavior of software. These techniques synthesize one or more state diagrams from properties (e.g., [9, 10, 27]), scenarios (e.g., [6, 14, 26, 28]), or both (e.g., [25]). In general, scenario-based synthesis techniques accept a set of scenarios as input and produce a set of communicating FSMs as output. Each scenario specifies a sequential set of messages passed between objects within the system. These messages form the alphabet. To compose scenarios, these algorithms rely on implicit information assumed by the algorithm [14] or explicit information provided by the user, such as state labeling [26], or more explicit representations of scenarios [6, 28]. In essence, property synthesis techniques establish a one-to-one mapping between a formally specified property and an FSM [9, 10, 27], where each FSM represents all possible behaviors that satisfy the property. These FSMs can

[1] In a companion paper, we further describe the possible software engineering applications of this approach [5].

then be executed in parallel to verify that all behaviors are satisfied. Uchitel *et al.* describe an approach that synthesizes modal transition systems (MTS) from both scenarios and properties [25]. Modal transition systems differ from FSMs in that they explicitly model behavior that may or may not occur. Essentially, they capture the search space of possible behavioral models.

These approaches differ from AVIDA-MDE in three key ways: First, AVIDA-MDE uses an evolving alphabet that is created by combining the triggers, guards, and actions inferred from the class diagram (described in Section 4). One ramification of this alphabet is that AVIDA-MDE generates different transitions than those generated from the alphabets of other approaches. As a result, the generated behavioral model has the potential to be less intuitive and perhaps offer more resiliency than those created with traditional techniques. Second, this technique supports generating behavioral models that include multiple, interacting FSMs, satisfy both scenarios and properties, and can extend a previously developed behavioral model. Third, AVIDA-MDE is a digital evolution-based technique for generating behavioral models for autonomic systems. In essence, we are harnessing digital evolution by using mutations to generate behavioral models that developers may not otherwise specify or even consider.

3. AVIDA

While evolutionary computation has been studied since the 1960's, the subfield of digital evolution is much younger. The first experiments with populations of self-replicating computer programs were performed in 1990 in a system called Coreworld [21], and later improved upon in Tierra [22]. In 1993, Ofria and colleagues began development of AVIDA [20], in which self-replicating digital organisms evolve in an open-ended fashion with more parallels to natural evolution than other forms of evolutionary computation. Indeed, until recently AVIDA has been used primarily by biologists: observing evolution in digital organisms enables researchers to address questions that are difficult or impossible to study with organic life forms.

AVIDA Operation. Figure 1 depicts an AVIDA population and the structure of an individual organism. Each digital organism consists of a circular list of instructions (its *genome*) and a virtual CPU. Instructions are executed by each organism's virtual CPU. The standard AVIDA instruction set is designed so that random mutations will always yield a syntactically correct program, albeit one that may not perform any meaningful computation.

An AVIDA environment comprises a number of *cells*, where a cell is a compartment in which an organism can live. Each cell can contain at most one organism, and the size of an AVIDA population is bounded by the number of cells in the environment. Organisms are *self-replicating*, that is, the genome itself must contain the instruction to create an offspring. When an organism replicates, a cell to contain the offspring is selected from the environment, and any previous inhabitant of the target cell is replaced (killed and overwritten) by the offspring. Each population starts with a single organism that is capable only of replication, and different genomes are produced through random mutations introduced during replication. Mutation types include: replacing the instruction with a different one, inserting an additional, random instruction into the offspring's genome, and removing an instruction from the offspring's genome.

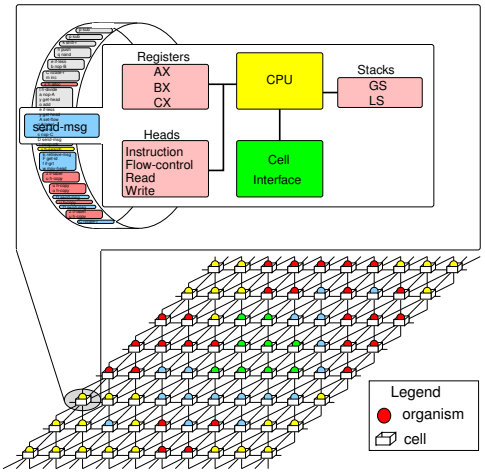


Figure 1: Elements of AVIDA platform

Developers use *tasks* to describe desirable organism behavior. Generally, tasks are defined in terms of the externally visible behaviors of the organisms (their phenotype). For example, a developer might define a task that rewards an organism for outputting the correct result of a particular computation (e.g., bitwise AND of two input values). In our study, tasks are defined in terms of application execution scenarios, functional properties, and software engineering metrics associated with the behavioral model constructed by an organism; see Section 4. Performing a task increases an organism's *merit* that determines how many instructions its virtual CPU is allowed to execute relative to the other organisms in the population. For example, an organism with a merit of 2 will, on average, execute twice as many instructions as an organism with a merit of 1. Since digital organisms are self-replicating and compete for space, a higher merit (all else being equal) results in an organism that replicates more frequently, spreading throughout and eventually dominating the population. Hence, AVIDA satisfies the three conditions necessary for evolution to occur [3]: replication, variation (mutation), and differential fitness (competition). AVIDA does not simulate evolution, it is an *instance* of evolution.

4. AVIDA-MDE

AVIDA-MDE extends AVIDA to harness the power of evolution to search for behavioral models for autonomic systems that satisfy developer requirements. Next, we discuss the extensions and then overview the experimental process for generating behavioral models.

4.1 Defining the Search Space

AVIDA-MDE enables developers to describe the search space for behavioral models using three characteristics: (1) The number of state diagrams present in the behavioral model; (2) Any elements of existing state diagrams; (3) The alphabet, defined on a per state diagram basis, for generating transitions. These three characteristics form the instinctual knowledge that an organism is provided with at birth. Because AVIDA-MDE is designed to be used in the context of MDE, instinctual knowledge is primarily defined in terms of the UML class diagram, which describes the structural elements of the system.

Instinctual knowledge is provided to the organisms in a *seed file* that describes information about each class and its accompanying state diagram (if one exists). Figures 2 (a) and (b) depict an elided class diagram for the robot navigation system and an elided portion of a seed file, respectively. From this information, AVIDA-MDE infers the number of state diagrams that should be present in the behavioral model. Specifically, a behavioral model should include a state diagram for each class (e.g., three state diagrams for the portion of the robot navigation system depicted in Figure 2 (a)). If the class has a *seed state diagram*, an existing state diagram, then it is described in the seed file. Additionally, a unique alphabet for each state diagram's transitions is described. For each class, a list of triggers (operations of the class), a list of guards (expressions built using the attributes of the class), and a list of actions (the operations of classes related to the class via associations) is provided. A null option is included as a member of each of these lists, thus enabling an organism to create a transition without a trigger, guard, or action. The alphabet for a given class's state diagram is defined in terms of all possible combinations of triggers, guards, and actions. For example, the ObstacleAvoidanceTimer (for the robot navigation system) seed file information (depicted in Figure 2 (b)) can be inferred from the class diagram (Figure 2 (a)). The triggers (i.e., sensorData(), wheelStopped(), timerEvent()) correspond to the operations of class ObstacleAvoidanceTimer. The guards (i.e., obstacle= 0 and obstacle= 1) are formed using the attribute of class ObstacleAvoidanceTimer and its values. The actions (i.e., SensorInterface.readSensor(), SensorInterface.readObstacleSensor(), NavigationControl.suspend(), and NavigationControl.restart()) are the operations of the classes with which the ObstacleAvoidanceTimer is associated (i.e., the SensorInterface and the NavigationControl classes).

4.2 Generating Models

We defined a new set of instructions to enable AVIDA-MDE organisms to generate behavioral models. These instructions are general in that they can be used to generate behavioral models for any autonomic system. The AVIDA-MDE set of instructions are used to: (1) *select* instinctual knowledge elements to manipulate, (2) *construct* new transitions, and (3) *replicate* the organism.

To add a transition to a state diagram, the organism selects several items from its instinctual knowledge: First, the organism must select the state diagram to be extended. Second, the organism must identify the members of the state diagram's alphabet (i.e., a trigger, a guard, and an action) that should be combined to create the transition's label. Third, the organism must identify the start and destination state for the transition. To that end, the selection instructions are used to index the lists that make up the organism's instinctual knowledge. Specifically, each type of list has its own set of instructions (one instruction per list item). The name of an instruction includes the type of element it is selecting (e.g., state diagram (sd), origin state (s-orig), destination state (s-dest), trigger (trig), guard(guard), or action (action)) and an integer representing a position in the list of that type. When state diagram selection instruction is executed, it changes the state diagram being manipulated and thus the triggers, guards, actions, and states being selected. When the other selection instructions are executed, they change the index pointing to the selected element. The addTrans instruction constructs the transition described by the selection instructions.

For example, Figure 2 (c) depicts an elided portion of an AVIDA-MDE organism's genome. Instruction sd-2 selects the ObstacleAvoidanceTimer state diagram. (The instinctual knowledge for the ObstacleAvoidanceTimer is depicted in Figure 2 (b).) Instruction trig-0 selects <*null*> as a trigger, meaning there is no explicit event activating the transition. Instruction guard-2 selects the second guard in the list, obstacle= 1, as a guard. Instructions s-dest-4 and s-orig-1 select state 1 as the origin and state 4 as the destination. Instruction action-3 selects the third action, NavigationControl.suspend(), as an action. Lastly, instruction addTrans constructs this transition and adds it to the ObstacleAvoidanceTimer state diagram (depicted in Figure 2 (d)).

Similar to the original AVIDA instructions, only syntactically correct genomes can be constructed using the AVIDA-MDE instructions. To that end, all of the instructions have acceptable default behavior. For example, if the selection instructions attempt to reference a list element that does not exist, e.g., the 11th element of a 10 item list, after executing the instruction, by default, the index will point to the last element of the list. Additionally, once the addTrans instruction is executed, the selection indices are reset to 0 (the first element of the lists).

Each AVIDA-MDE organism maintains an internal, graph-based representation of its behavioral model. However, we enabled AVIDA-MDE to export models into XMI format files to be used with XMI-based UML modeling tools, such as Argo-UML for visualization and the Hydra translation tool [19], which is subsequently described.

4.3 Evaluating Behavioral Models

We defined a set of tasks to reward AVIDA-MDE organisms for generating behavioral models that meet the developer's requirements, specified in terms of example scenarios, safety properties, and software engineering metrics (e.g., minimum number of transitions). Organisms that perform these tasks will have higher merit, execute their instructions faster, and have a better chance of replicating more frequently, thereby dominating the population. Specifically, if an organism receives a reward for a task, then its merit is multiplied by 2^{reward}. Because each task is rewarded individually, AVIDA-MDE does not explicitly arbitrate conflicts between rewards. However, because of the compute time required to assess property tasks, we do impose a partial order on these tasks in that an organism must satisfy the scenario tasks, prior to attempting the property tasks.

Scenarios. The scenario tasks (checkScenario) reward organisms that generate behavioral models that include developer-specified scenarios. For each scenario, the developer must specify the messages between objects and may optionally include a start state for each object and specify whether the scenario should iterate. The reward for a checkScenario task is the percentage of the scenario execution path included in the state diagrams. For example, if the developer specifies a scenario with four messages, an organism that generates a state diagram with two of the messages would receive a reward of 0.5. To receive the maximum reward for a scenario, an organism must generate a state diagram for each object involved in the scenario that

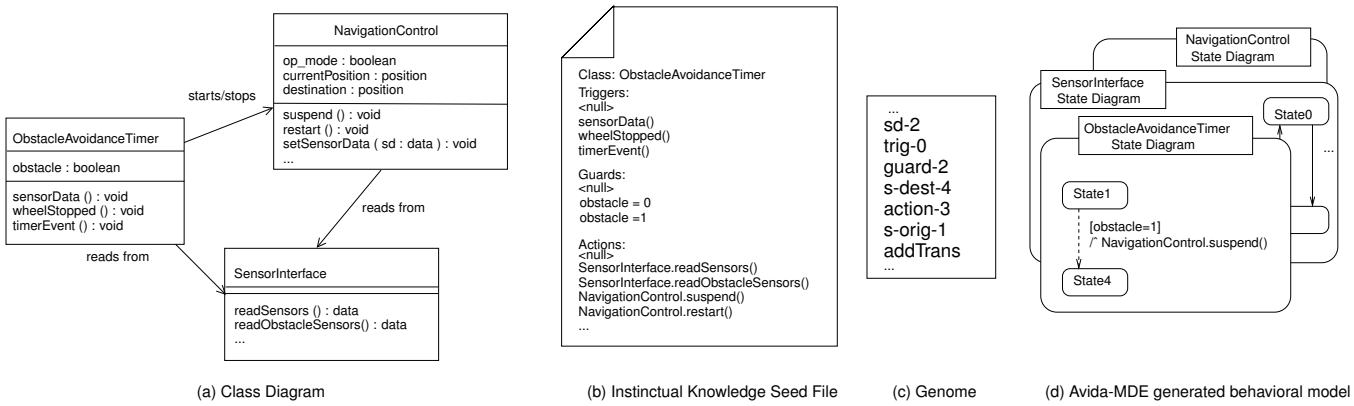


Figure 2: Class diagram, instinctual knowledge, genome, and generated behavioral model

includes a path specifying the messages sent and received by the object.

Properties. We defined a set of tasks to reward organisms that generate behavioral models that adhere to formal properties specified by the developer. Evaluating whether a UML model adheres to a formal property involves three key steps: (1) Because UML lacks a formally defined semantics, to analyze a UML model, it must first be translated to a formal representation. (2) To ensure that properties are not *vacuously* satisfied, the formal model must be analyzed to ensure that at least one execution path through the model satisfies the property. (3) The formal model is analyzed (i.e., model checked) for adherence to the formally specified property. To enable Avida-MDE to perform these steps we defined three tasks and extended Avida to use corresponding existing software engineering tools. Next, we describe the three tasks and their respective tool support.

First, the checkSyntax task translates the UML class and state diagrams into a formal specification language. For this task, we use Hydra, an existing UML formalization engine [19], to translate UML to Promela, the specification language for the model checker Spin [7]. Next, the checkWitness task uses Spin to verify that the Promela model does not vacuously satisfy the property specified by the developer in Linear Temporal Logic (LTL). We accomplish this objective by negating the property and using Spin to search for a counter-example [12]. Lastly, if the checkWitness task passes, then the checkProperty task uses Spin to verify that the Promela specification satisfies the same property.

SE Metrics. Additionally, we use software engineering metric tasks to reward organisms for generating state diagrams that meet commonly advocated software engineering metrics. Succeeding at these tasks makes it more likely that the behavioral model will also satisfy the properties. Currently, we have defined the min-trans and determinism tasks. The min-trans task rewards an organism for generating a behavioral model with fewer transitions, relative to the behavioral models generated by the rest of the population. Specifically, the task identifies the maximum number of transitions used by an organism (max) and the number of transitions used by the organism being evaluated (current). The reward is computed by taking the difference between the max and current and dividing it by the max. The determinism task rewards an organism for generating behavioral models that are *deterministic*, that is at most one transition can be taken from each state for a given event and guard combination.[2] This task calculates the percentage of nondeterministic states per state diagram, and uses the sum of these percentages as a reward. Other SE metric tasks could reward organisms for minimizing coupling, etc.

4.4 Experimental Process

To provide intuition for how Avida-MDE organisms generate a behavioral model, consider a typical experiment. A population starts with a single organism that is only capable of replication and a maximum population size of 3,600 organisms. As the organism and its offspring replicate, different genomes are produced through random mutations. Organisms that generate behavioral models exhibiting desired characteristics receive more merit and thus replicate faster. Therefore, over time, the population progressively comprises organisms that generate behavioral models that exhibit more of the desired characteristics. If an organism generates a behavioral model that supports all the key scenarios and satisfies all of the properties, then it has successfully, and automatically, generated a UML behavioral model for a target system. We refer to such behavioral models as *compliant behavioral models*.

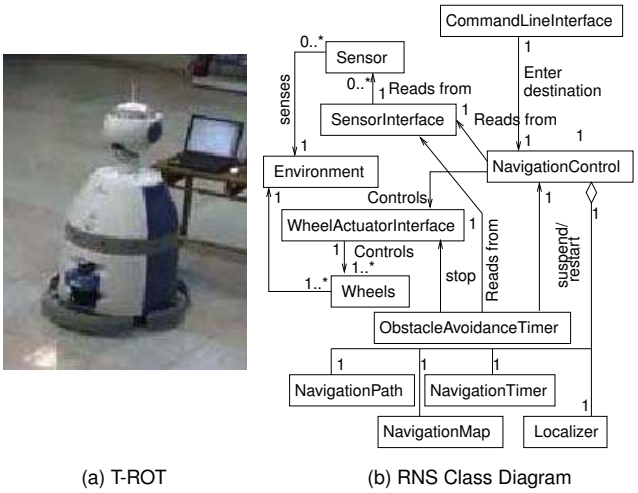
In general, we run 100 parallel Avida-MDE experiments. Multiple experiments are performed to account for the stochastic nature of the evolutionary process. Each experiment runs for 200,000 updates, where an update, on average, executes 30 instructions per organism (updates are the standard unit of time in Avida experiments).

Now, let us consider the normal life cycle of an organism. When the organism is created through the replication process, it is provided with instinctual knowledge of class diagram elements and any seed state diagrams. As the organism executes the instructions in its genome, different pieces of instinctual knowledge are selected and used to create transitions. When the organism replicates, the behavioral model it generated is evaluated according to the requirements provided by the developer, and its merit is calculated. As the parent copies its genome to its offspring, mutations may be introduced. These mutations may cause the offspring to generate a behavioral model that differs from that generated by

[2]Because our target application domain is high-assurance systems, it is desirable for the models to be deterministic.

its parent. The offspring is placed in a different cell, possibly replacing another organism. Then both the parent and the child begin execution at the start of their genome, each with the same instinctual knowledge.

Once the experiments are complete, the developer may use the compliant behavioral models in different ways. The differing characteristics of the compliant behavioral models generated by AVIDA-MDE make them potentially more amenable for different environmental conditions that were not explicitly stated. In general, the developer can review such solutions, each of which specify different ways of achieving the compliant behavior, or may adjust other properties to further distinguish the solutions. One possibility is to use these behavioral models to inspire human-created designs. Another possibility is to select behavioral models, improve them manually, and use them as the starting point for the MDE of the autonomic system. Lastly, a developer may use the behavioral models without modification. Feedback from our industrial collaborators indicates that all three options are attractive when looking for innovative solutions.



(a) T-ROT (b) RNS Class Diagram

Figure 3: RNS elided class diagram

5. EXPERIMENTS

We illustrate our approach by generating target system behavioral models for T-ROT, an intelligent, autonomous robot developed by Kim *et al.* [11] (depicted in Figure 3 (a)). Currently, the world's population is aging and the cost of health care is increasing. As a result, the Korea Institute of Service and Technology (KIST) is developing T-ROT to assist in the care and support of the elderly [11], e.g., by performing errands, such as retrieving medicine. Because T-ROT is resource constrained (both in battery life and space), it autonomically adapts its behavior in response to its environment (e.g., by removing and uploading new software components at run time). One particularly challenging problem is to enable T-ROT to autonomously navigate to a designated position in a variety of environments. For example, T-ROT may navigate differently if the elderly person is in peril, if there is a baby in the room, or if the room has many obstacles.

The Robot Navigation System (RNS) comprises several software objects (depicted in Figure 3 (b)). The NavigationControl receives a destination from the CommandLineInterface. It plans a path (NavigationPath) from its current location, identified using the Localizer, to the destination using its map (NavigationMap). The NavigationControl then senses the environment through the SensorInterface and controls the robot wheels through the WheelActuatorInterface. The ObstacleAvoidanceTimer is responsible for detecting obstacles, stopping the wheels, and suspending NavigationControl.

5.1 Experimental Setup

The objective of these experiments is to generate behavioral models for RNS target systems that avoid obstacles and may include additional resilient behavior. Specifically, if T-ROT encounters an obstacle, then it should stop. Next, we describe the application-specific instinctual knowledge provided to the organisms and the behavioral requirements.

Instinctual Knowledge. We provided the AVIDA-MDE organisms with instinctual knowledge that included the UML class diagram for the RNS (depicted in Figure 3) and the seed state diagrams for all the classes, except the ObstacleAvoidanceTimer. KIST had previously created a state diagram for the NavigationControl class, which we modified by removing the elements that corresponded to avoiding obstacles to create the NavigationControl seed state diagram (since the objective of the experiment was to have AVIDA-MDE generate behavioral models that capture this behavior given properties and scenarios). Therefore, the initial behavior captured by the seed state diagrams is that the RNS is able to navigate to a destination, but does not avoid obstacles.

Requirements. We used several tasks as requirements for the generated behavioral models. Specifically, we defined two new scenarios (with an obstacle and without an obstacle), two properties (that T-ROT must avoid obstacles and that T-ROT must reach its destination), and two software engineering metrics (minimum number of transitions and deterministic states).

Specifically, we defined two checkScenario tasks based upon a collaboration diagram specified by KIST [11]. The first task, checkScenario-obstacle, rewards organisms for generating behavioral models that stop T-ROT when it encounters an obstacle. In this execution path, the ObstacleAvoidanceTimer detects an obstacle, suspends the NavigationControl, and stops the WheelActuatorInterface. The second task, checkScenario-no-obstacle, rewards organisms for generating behavioral models that restart T-ROT when the obstacle is no longer present. In this execution path, the ObstacleAvoidanceTimer does not detect an obstacle and thus restarts the NavigationControl. The execution path for the SensorInterface is the same in both scenarios.

Next, we specified two properties that the behavioral models generated by the AVIDA-MDE organisms should satisfy:

1. *Globally, it is always the case that eventually* T-ROT's *current position will be its destination.*
2. *Globally, it is always the case that if the* ObstacleAvoidanceTimer *detects an obstacle, then eventually the* NavigationControl *will be suspended.*

The first property specifies the desirable general behavior of the RNS. This property was satisfied by the seed state diagrams and should be satisfied by all the compliant behavioral models. The second property specifies the new behavior: that the ObstacleAvoidanceTimer should stop T-ROT

if it detects an obstacle. These properties are used as checkProperty tasks.

5.2 Experimental Results

We ran 100 instances of AVIDA-MDE, each with a maximum population size of 3,600 organisms, for 200,000 updates. The performance of an AVIDA-MDE experiment is dependent upon the complexity of the tasks, seed state diagrams, and available computational resources. These experiments ran for under 24 hours on a high performance computing cluster. By the end of the experiment, seven behavioral models were generated. Although all seven of the generated behavioral models include the key scenarios and adhere to the properties, their overall behavior is not identical. In Figure 4, we present the topology of the seven compliant behavioral models to give a flavor for the variation between models. Although the transition labels used by these models also differ, they have been elided due to space constraints. Seed state diagram elements are depicted as solid lines and shaded states; generated transitions are depicted as dotted lines; states used by the generated transitions, but not the seed state diagrams are depicted as non-shaded states.

Post-Evolution Analysis. To identify compliant behavioral models to use for target systems, we performed some post-evolution analysis. Specifically, we evaluated the behavioral models using five additional criteria: minimum states (the number of generated states used), minimum transitions (the number of generated transitions used), fault tolerance (whether any redundancy is present), readability (how easily a developer could understand the model), and safety (whether the model satisfied two additional properties specified in the following). The safety properties are:

1. *Globally, it is never the case that the* ObstacleAvoidanceTimer *does not detect an obstacle and the* NavigationControl *is suspended.* (preventing spurious suspensions)
2. *Globally, it is always the case that if the* NavigationControl *is suspended, then eventually the* NavigationControl *receives a message confirming that the wheels have been stopped.* (ensuring that T-ROT actually stops)

The analysis results are presented in Figure 4, where a plus (+) indicates the model partially met the criteria, a double plus (++) indicates the model fully met the criteria, a minus (−) indicates the model did not meet the criteria, and a double minus (−−) indicates the model exhibited the inverse of the criteria.

In prioritizing the additional criteria, we opted to focus on safety and fault tolerance, since T-ROT must reliably operate in the presence of an elderly person. Depending on the developer's specific concerns, different criteria could be both selected and prioritized. Three models, models 1, 4, and 7 met both of the safety properties and thus fully met the safety criteria. Additionally, models 1 and 4 exhibited some degree of fault tolerance. Model 1 partially minimizes the number of states used, but does not minimize transitions and is difficult to read. In contrast, model 4 is more fault tolerant (it has a greater amount of redundancy) and minimizes the number of transitions used, but does not minimize states used and is difficult to read. Beyond simply satisfying the properties of avoiding obstacles and reaching a destination, these models might represent implicit behavior that makes them better suited to responding to the uncertainty present in the environment of the RNS.

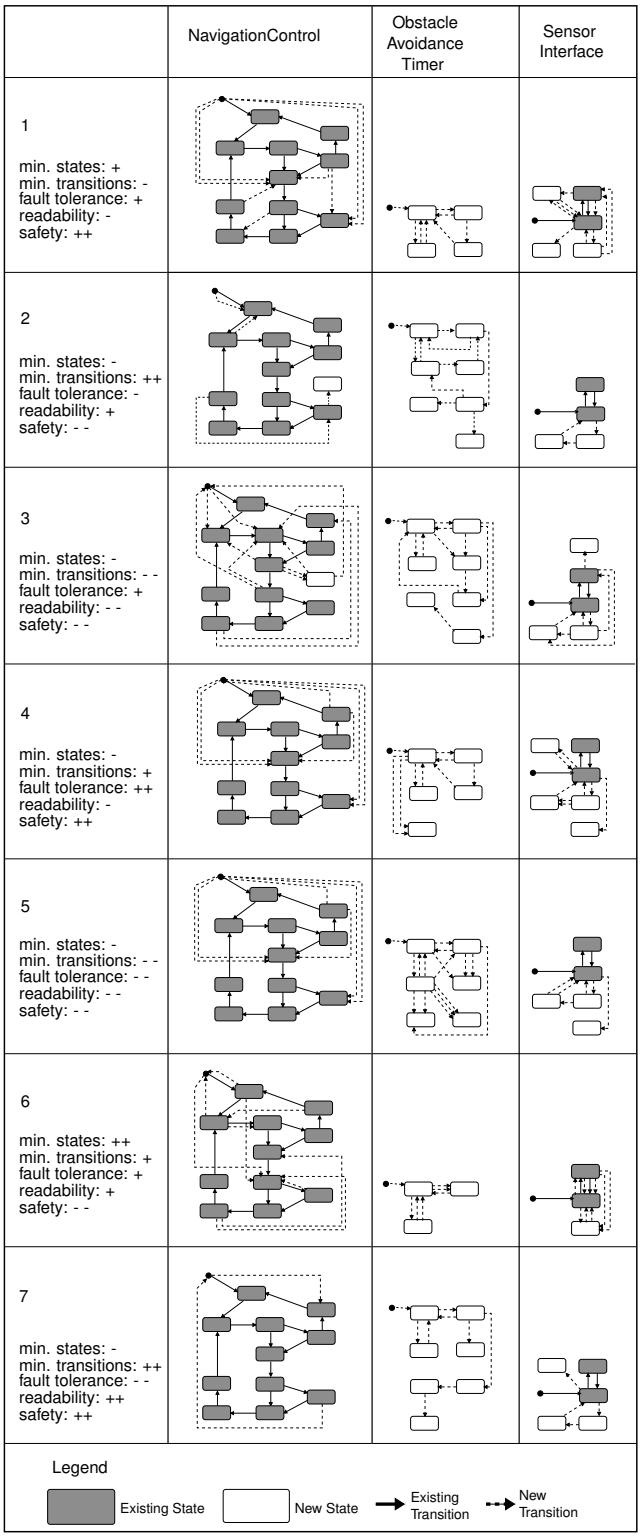


Figure 4: AVIDA-MDE Generated Compliant Behavioral Models

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented AVIDA-MDE, a digital evolution-based tool for generating behavioral models that support scenarios, satisfy properties, and extend an existing behavioral model. We formulated the generation of behavioral models as a search problem. Specifically, the class diagram is used to specify the alphabet for the behavioral model and thus constrain the search space. Additionally, the developer-specified requirements are used to constrain the solution space. AVIDA-MDE improves the productivity of developers by enabling target systems to be automatically generated from requirements specified as scenarios and properties. Additionally, AVIDA-MDE has the potential to improve the quality of the autonomic system by generating target systems that may be better able to react and respond to a variety of environmental conditions. We have successfully applied this method to generating target systems for an autonomic system.

There are several possible directions for future work. One possibility is to use AVIDA-MDE to explore solution spaces for other software engineering problems, such as generating UML behavioral models for members of a software product line or detecting and mitigating feature interaction. Another possibility is to extend AVIDA-MDE to make use of additional behavioral diagrams, e.g., activity diagrams, to further improve the generated solution. Additionally, we are interested in using a variant of AVIDA-MDE to explore existing models to generate properties capturing their additional behavior (i.e., latent properties). Lastly, we envision combining these two approaches into an iterative development approach in which AVIDA-MDE is used to: (1) generate behavioral models; (2) identify the latent properties satisfied by the various behavioral models; and (3) refine the behavioral models by explicitly either disallowing or requiring the identified latent properties.

7. REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[2] K. Chellapilla and D. Czarnecki. A preliminary investigation into evolving modular finite state machines. In *Congress on Evolutionary Computation*, 1999.

[3] D. C. Dennett. The new replicators. In M. Pagel, editor, *The Encyclopedia of Evolution*, volume 1, pages E83–E92. Oxford University Press, 2002.

[4] L. J. Fogel, P. J. Angeline, and D. B. Fogel. An evolutionary programming approach to self-adaptation on finite state machines. In *Evolutionary Programming*, 1995.

[5] H. J. Goldsby, B. H. C. Cheng, P. K. McKinley, D. B. Knoester, and C. A. Ofria. Digital evolution of behavioral models for autonomic systems. In *Proceedings of the 5th International Conference on Autonomic Computing (ICAC 2008)*, Chicago, Illinois, June 2008.

[6] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, 2005.

[7] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2004.

[8] Y. Inagaki. On synchronized evolution of the network of automata. *IEEE Transactions on Evolutionary Computation*, 6, 2002.

[9] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, 2006.

[10] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using spin: A case study on web services. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods*, pages 406–415, 2004.

[11] M. Kim, S. Kim, S. Park, M.-T. Choi, M. Kim, and H. Gomaa. UML-based service robot software development: a case study. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 534–543, 2006.

[12] S. Konrad, H. Goldsby, K. Lopez, and B. H. C. Cheng. Visualizing requirements in UML models. In *Proceedings of the International Workshop on Requirements Engineering Visualization (REV 2006) as part of (RE'06)*, September 2006.

[13] J. R. Koza, M. A. Keane, M. J. Streeter, M. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer, 2003.

[14] I. Kruger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. *In DIPES'98. Kluwer.*, 1999.

[15] S. M. Lucas and T. J. Reynolds. Learning DFA: evolution versus evidence driven state merging. In *Congress on Evolutionary Computation*, 2003.

[16] S. Luke, S. Hamahashi, and H. Kitano. "Genetic" programming. In *Genetic and Evolutionary Computation Conference*, 1999.

[17] P. McKinley, B. H. Cheng, C. Ofria, D. Knoester, B. Beckmann, and H. Goldsby. Harnessing digital evolution. *IEEE Computer*, 41(1):54–63, 2008.

[18] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[19] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE01)*, Toronto, Canada, May 2001.

[20] C. Ofria and C. O. Wilke. Avida: A software platform for research in computational evolutionary biology. *Journal of Artificial Life*, 10:191–229, 2004.

[21] S. Rasmussen, C. Knudson, P. Feldberg, and M. Hindsholm. The coreworld - emergence and evolution of cooperative structures in a computational chemistry. *Physica D*, 42(1-3):111–134, 1990.

[22] T. S. Ray. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 371–408. Addison-Wesley, Reading, MA, USA, 1992.

[23] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), 2006.

[24] H. Shayani and P. J. Bentley. A more bio-plausible approach to the evolutionary inference of finite state machines. In *GECCO conference companion on Genetic and evolutionary computation*, pages 2937–2944, New York, NY, USA, 2007. ACM.

[25] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 34–43, 2007.

[26] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.

[27] H. T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *Proceedings of RE'04, 12th IEEE Joint International Requirements Engineering Conference*, 2004.

[28] J. Whittle and P. K. Jayaraman. Generating hierarchical state machines from use case charts. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 16–25, Washington, DC, USA, 2006.