Proceedings of the 2007 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, Minneapolis, Minnesota, May 2007.

Digitally Evolving Models for Dynamically Adaptive Systems[*]

Heather J. Goldsby, David B. Knoester, Betty H.C. Cheng[†] Philip K. McKinley, Charles A. Ofria
{hjg, dk, chengb, mckinley, ofria}@cse.msu.edu
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA

Abstract

Developing a Dynamically Adaptive System (DAS) requires a developer to identify viable target systems that can be adopted by the DAS at runtime in response to specific environmental conditions, while satisfying critical properties. This paper describes a preliminary investigation into using digital evolution to automatically generate models of viable target systems. In digital evolution, a population of self-replicating computer programs exists in a user-defined computational environment and is subject to instruction-level mutations and natural selection. These "digital organisms" have no built-in ability to generate a model – each population begins with a single organism that only has the ability to self-replicate. In a case study, we demonstrate that digital evolution can be used to evolve known state diagrams and to further evolve these diagrams to satisfy system critical properties. This result shows that digital evolution can be used to aid in the discovery of the viable target systems of a DAS.

1. Introduction

Increasingly, applications rely on Dynamically Adaptive Systems (DASs) to respond to changes in their environment. For example, environmental monitoring systems rely on dynamic adaptation to respond to hardware component failures, network outages, and limited power supplies. The developer of a DAS needs to identify *viable target systems* to be adopted in response to specific environmental conditions. Additionally, these target systems should satisfy critical (system invariant) properties. To facilitate the development process and leverage automated techniques (such as Model Driven Development [26]), UML diagrams can be used to represent DAS requirements. However, manually constructing models for different combinations of environmental conditions is a challenging task. In addition, these models need to be analyzed for adherence to system critical properties. This paper presents our preliminary work in using digital evolution, a form of evolutionary computation, to automatically generate state diagrams that represent a portion of the viable target systems of a DAS.

Several approaches have been developed to model the target systems for a DAS, including: architecture description languages [2, 6, 16, 17], goal models [8, 18], and state machines [18, 36]. The general idea is to model the structure, behavior, or objectives of each possible target system of a DAS to support increased comprehension and possibly analysis. One drawback is that the viable target systems must be modeled by hand, thus limiting the exploration of target systems to those envisioned by the developer.

In this work, we explore an approach inspired by nature. Living organisms are astonishingly adept at adapting to changing environmental conditions both in the short term (phenotypic plasticity) and in the longer term (genetic evolution). Our approach leverages this power by using *digital evolution* to automatically explore the solution space of a DAS and identify viable target systems that satisfy system critical properties. In digital evolution, a population of computer programs, called *organisms*, exist in a user-defined computational environment. Organisms self-replicate (i.e., copy their program), compete for available resources (e.g., CPU time), and are subject to instruction-level mutations. The evolution of these organisms may be guided by defining characteristics of solutions. This enables the organisms, over thousands of generations, to evolve viable and often unanticipated solutions to problems.

In this paper, we show that digital evolution can be used to evolve known state diagrams and to further evolve these diagrams to satisfy system critical properties. We have used and extended AVIDA [29], the most widely used digital evo-

[*]This work has been supported in part by NSF grants EIA-0000433, EIA-0130724, CDA-9700732, CCR-9901017, Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744, Eaton Corporation, Siemens Corporate Research, and a grant from Michigan State University's Quality Fund.

[†]Please contact this author for all correspondences.

lution platform, for this investigation. Two key insights were essential to this approach. First, we utilized an organism as a generator of a state diagram: each organism constructs an in-memory representation of a state diagram during its execution. Second, we extended the AVIDA platform to evaluate the generated state diagrams using a model checker, Spin [11]. To accomplish this, we use Hydra [23], our previously developed UML formalization framework, to translate a state diagram generated by an AVIDA organism into Promela for use with the Spin model checker. Thus, we specify *what* properties the state diagram should satisfy and use a model checker to *verify* that the state diagrams generated by AVIDA organisms adhere to these properties.

We illustrate our technique on the evolution of a state diagram that represents a portion of the behavior of an emergency response system (ERS). The remainder of this paper is organized as follows. Section 2 overviews related work in biologically-inspired approaches to solving computer problems. Section 3 gives background information on the AVIDA platform and Hydra. Section 4 describes our investigation, including how we extended the AVIDA platform to support state diagram evolution and how we use model checking to evaluate state diagrams. Section 5 describes how we applied the approach to the evolution of a state diagram for a portion of a viable target system for the ERS. Section 6 discusses our contributions and concludes with a description of future work.

2. Related Work

Biologically-inspired approaches to solving computer problems fall broadly into two categories, biomimetics and evolutionary computation. *Biomimetic* systems mimic the behavior of natural systems. For example, the social behavior of insect colonies has been used to develop scalable and robust distributed systems [12, 22, 31, 34]. Researchers have also shown that certain natural behaviors have important analogies in cyberspace, for example, using the concept of chemotaxis to facilitate robust network routing [3], and modeling colony concepts and behaviors (diversity, migration, discovery, pheromone emission) to help construct highly-available distributed services [32]. Additionally, Tyrrell *et al.* have taken inspiration from the human immune system and embryonic development to produce fault tolerant hardware called immunotronics and embryonics, respectively [4]. Moreover, they have taken an immunotronic approach to detecting errors in finite state machine representations of hardware [5].

An alternative approach is to focus on the process that produced those behaviors: evolution. *Evolutionary computation* includes any mechanism that satisfies the three conditions necessary for evolution to occur [30]: replication, variation (mutation), and differential fitness (competition). The most well-known method of evolutionary computation is the genetic algorithm (GA) [10], an iterative search technique in which the individuals in the population are encodings of candidate solutions to an optimization problem. In each generation, the fitness of every individual is calculated, and a subset of individuals are selected, recombined and/or mutated, and moved to the next generation. Genetic programming (GP) [15] is a related method where the individuals are actual computer programs. GP uses replication, recombination, and competition to search for a globally optimal solution among all possible programs. GA/GP have been applied to a wide variety of problems [25, 33], including the automation of unit testing [33], in some cases developing patentable solutions [15]. Of particular interest to the evolution of state diagrams, GA/GP have been used to optimize the state transition table of sequential logic circuits [1], as well as to design complete hardware circuits [15, 21].

Digital evolution is a type of evolutionary computation. When applied in science and engineering, digital evolution provides a means to explore very large solution spaces to problems, potentially discovering solutions that would not otherwise be apparent [7, 9]. Unlike biomimetic approaches, digital evolution is not connected to behaviors found in existing living organisms. Moreover, in digital evolution, an organism's survival depends on how well it compares to other organisms; whereas, in other evolutionary computation approaches, such as GA/GP, an organism's survival depends on how well it performs according to an explicitly stated fitness function.

3. Background

This section provides a brief introduction to the AVIDA platform [29] and Hydra [23], our UML formalization framework.

3.1. AVIDA Digital Evolution Platform

AVIDA is a digital evolution platform that has previously been used to conduct pioneering research in the evolution of biocomplexity, with an emphasis on understanding the evolutionary design process in nature. Specific studies address the evolutionary origin of complex features [20], the evolutionary design of modularity [24, 27], and organism robustness [19, 35]. More recently, AVIDA has been used to study the evolution of collaborative behaviors in distributed systems [13, 14].

Figure 1 contains a simplified class diagram of AVIDA's structure (the shaded classes are our extensions to the platform and are described in Section 4). Each Organism has a VirtualCPU and a Genome that comprises a set of Instructions. The VirtualCPU comprises virtual hardware resources (registers, stacks, instruction pointer) and executes Instructions. Each type of Instruction is similar in appearance and functionality to a traditional assembly language instruction. The Instructions enable an organism to perform simple mathematical operations, such as addition, multiplication, and bit-shifts, as well as interact with the organism's environment. The standard AVIDA instruction set is Turing-complete, and therefore theoretically able to evolve any computable function. A key property of AVIDA's instruction set that differs from traditional computer languages, however, is that it is not possible to construct a *syntactically incorrect* genome; that is, all possible genomes are executable. Hence, while random mutations will produce many genomes that do not perform any meaningful computation, their instruction sequences will still be valid. Indeed, this property is critical to the evolutionary process [28].

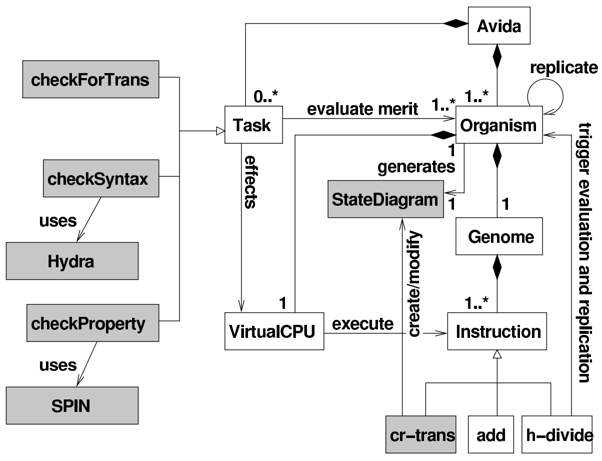
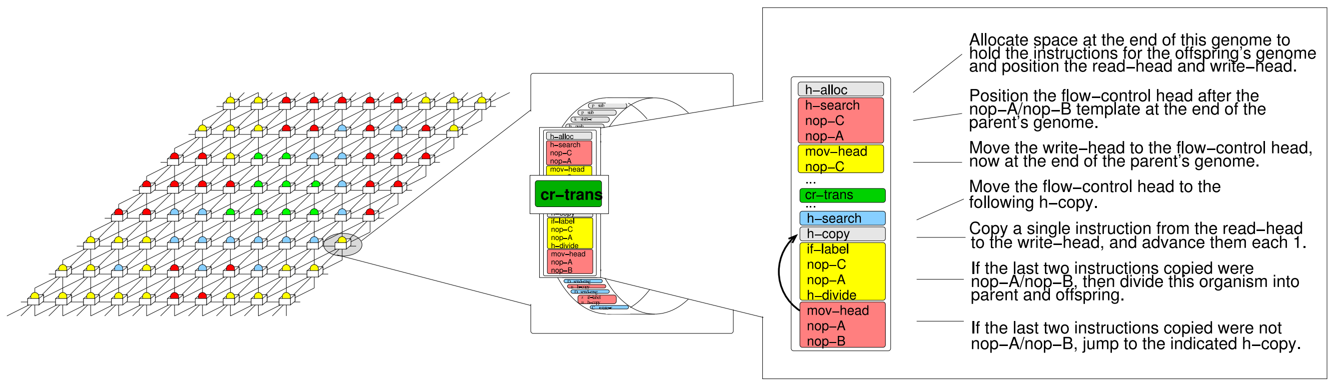


Figure 1. Simplified AVIDA class diagram

During an AVIDA experiment, the *merit* of a given digital organism 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, a higher merit (all else being equal) results in an organism that replicates more frequently, spreading throughout and eventually dominating the population. Merit of a digital organism is updated based upon the Tasks that are performed by the organism. Tasks are designed by the developer and reward desirable behavior (they may also punish undesirable behavior), thereby driving natural selection. For example, to encourage a particular computation, a developer might define a task that rewards an organism by doubling its merit when it outputs the correct result of that computation to the environment. Tasks are generally defined in terms of externally visible behaviors of the organisms (their phenotype). This approach allows maximum flexibility in the evolution of a solution for a particular task. The evolved solution might not be optimal when considering the task in isolation, but it is likely to have other properties that made it well-suited for its environment – robustness to mutation, for example.

Figure 2(a) depicts a population of AVIDA organisms in their environment. The 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 has a circular list of directed *connections* to neighboring cells; these connections define the topology of the environment. Three different topologies are currently available: GRID, TORUS, and CLIQUE (completely connected), and we are planning additional topologies, including a configuration where the cells are mobile and may change their topology during the experiment. When a digital organism replicates, a cell to contain the offspring must be selected from the environment. This target cell may be selected at random from the environment (MASS-ACTION), or it may be selected from the neighborhood of the parent (LOCAL-REPLACEMENT). In all cases, any previous inhabitant of the target cell is replaced (killed and overwritten) by the offspring. LOCAL-REPLACEMENT is best suited for experiments where digital organisms interact with each other (for example, there are instructions in AVIDA that enable organisms to communicate), while MASS-ACTION is better suited for experiments where organisms do not interact. At any point in time, the population of digital organisms may contain many different genomes. Some may be closely related (e.g., parent and offspring), while others may be related only through a distant ancestor. Each population starts with a single organism that is only capable of replication, and different genomes are produced through random mutations that occur during replication. The organisms in Figure 2(a) are shaded to represent different active genomes. Figure 2(b) depicts the genome of a specific organism that is executing the cr-trans instruction on its virtual CPU.

Of the different instructions that may be present in an organism's genome, those that are part of the *replication cycle* are of critical importance. During the replication cycle, an organism's genome experiences variation in the form of random instruction-level mutations. Figure 2(c) depicts the instructions involved in a typical replication cycle of the default AVIDA organism (mutations are likely to modify this sequence in descendants). The first step in replication is for the parent to allocate space for the offspring in its genome. The parent then executes its "copy-loop," where instructions are copied individually from the parent's genome to the offspring's. Finally, the parent organism executes an `h-divide` instruction, which splits its genome into two parts, creating two organisms. Each time an instruction is copied, a mutation may be introduced according to a prede-


cr–trans
h–alloc
h–search
nop–C
nop–A
mov–head
nop–C
...
cr–trans
...
h–search
h–copy
if–label
nop–C
nop–A
h–divide
mov–head
nop–A
nop–B
Allocate space at the end of this genome to hold the instructions for the offspring's genome and position the read–head and write–head.
Position the flow–control head after the nop–A/nop–B template at the end of the parent's genome.
Move the write–head to the flow–control head, now at the end of the parent's genome.
Move the flow–control head to the following h–copy.
Copy a single instruction from the read–head to the write–head, and advance them each 1.
If the last two instructions copied were nop–A/nop–B, then divide this organism into parent and offspring.
If the last two instructions copied were not nop–A/nop–B, jump to the indicated h–copy.

(a) an Avida population containing 4 primary genomes (b) genome of an individual organism (c) instruction sequence for Avida replication cycle

Figure 2. An AVIDA population, organism genome, and replication cycle

fined probability. These mutations may take the form of a replacement (substituting a random instruction for the one copied), an insertion (inserting an additional, random instruction into the child's genome), or a deletion (removing the copied instruction from the child's genome). When an organism replicates, a target cell that will house the new organism is selected from the environment. An organism already present in the target cell is *replaced* (killed and overwritten) by the offspring.

3.2. Hydra Formalization Framework

We defined an AVIDA task to check whether a generated state diagram satisfies a given critical property, such as a safety invariant. One task we defined for AVIDA utilizes Hydra, our UML formalization engine, to generate a Promela specification of a model. Previously, McUmber and Cheng [23] developed a metamodel-based formalization framework to support the mapping of UML models to formal specification languages, thus making the models amenable to formal analysis. Hydra is a prototype tool that supports the automatic mapping and code generation process. For example, Hydra supports the automated translation of UML to Promela, the specification language for the model checker Spin [11]. The general UML-to-Promela formalization approach is to map objects to processes in Spin (*proctypes*) that exchange messages via *channels*. Nested and concurrent states are also formalized as processes. Using Hydra, we can generate a Promela model for state diagrams that can be model checked with Spin for adherence to critical properties. Additional details on the modeling and analysis process, and the underlying formalization framework can be found in [23].

4. Preliminary Investigation

Our investigation into using digital evolution to evolve organisms that generate state diagrams focuses on two key questions:

1. Can an AVIDA organism generate a known state diagram?
2. Can an AVIDA organism further evolve a known state diagram to satisfy system critical properties?

Next, we discuss the extensions we made to the AVIDA platform in an effort to address each of these questions.

4.1. Generate a known state diagram

The first step was to use digital evolution to attempt to generate a known state diagram. Our hypothesis was that if we enabled AVIDA organisms to construct states and transitions and provided the organisms with a predefined set of transition labels, then an organism could generate a known state diagram. To this end, we extended AVIDA in two ways: (1) we provided an AVIDA organism with an instruction to generate states and transitions and (2) we equipped AVIDA to assess whether the generated state diagram was the desired diagram.

We defined a new instruction, cr-trans, which generates both states and transitions. The cr-trans instruction takes three parameters that represent the label for the transition, starting state, and destination state, respectively. If an instance of the cr-trans instruction uses a state label that does not yet exist in the organism's state diagram, then a state with that label is added to the diagram. If an instance of the cr-trans instruction uses a unique transition, starting state, and ending state combination, then a transition with that label is added to the diagram.

AVIDA represents the labels of states and transitions as integers, where these integer labels can be associated with strings provided by the developer. We enabled AVIDA to represent labels as integers because the AVIDA instruction set is tailored to performing integer manipulation. There are two different sets of integer labels; one set represents state labels and the other represents transition labels. To associate an integer label with a string, the developer must provide an ordinal mapping between integer labels and string values. For transitions, these string values should represent combinations of triggers, guards, and actions. In the following, for clarity, we refer to the states and transitions by their string values, rather than by their integer labels.

Figure 3 depicts two example executions of the cr-trans instruction and the resulting state diagrams. The generated state diagrams specify a portion of the behavior of a ProcessingComponent for a temperature sensor. In Figure 3(a), we see the state diagram resulting from a call to cr-trans with a transition label and two unique state labels. In Figure 3(b), we see that the state diagram resulting from a subsequent call to cr-trans with a different transition label, one unique state label, and one preexisting state label.

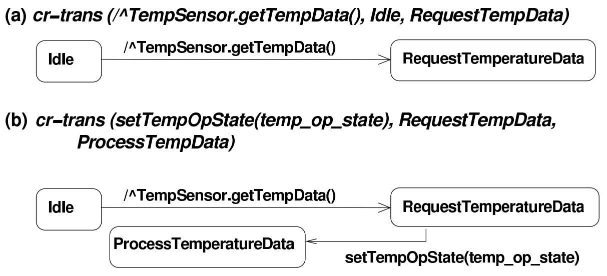

(a) cr–trans (/^TempSensor.getTempData(), Idle, RequestTempData)
Idle
/^TempSensor.getTempData()
RequestTemperatureData
(b) cr–trans (setTempOpState(temp_op_state), RequestTempData, ProcessTempData)
Idle
/^TempSensor.getTempData()
RequestTemperatureData
ProcessTemperatureData
setTempOpState(temp_op_state)

Figure 3. Examples of cr-trans **instruction**

To enable AVIDA to assess whether an organism generated the known diagram, we extended AVIDA with a checkForTrans task. Each instance of the checkForTrans task is configured by the developer to assess whether a specific transition is present within the generated state diagrams. Multiple instances of the checkForTrans task may be combined to check for a specific state diagram. If a desired transition is present in the model, then the organism is awarded additional merit. However, the organism is not further rewarded for multiple occurrences of the desired transition.

For example, Figure 4 depicts a known state diagram. A developer configures four checkForTrans tasks, one for each transition in the known state diagram. These tasks are used to determine if the evolved diagram depicted in Figure 3(b) is the known state diagram. In this case, the organism receives additional merit for the /^ TempSensor.getTempData() transition, which connects the Idle state and the RequestTemperatureData. However, the organism does not receive merit for the other three checkForTrans tasks, since the generated state diagram does not contain the other desired transitions.

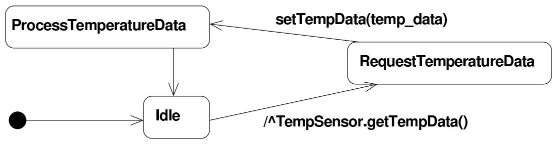


Figure 4. State diagram for the Processing-Component

4.2. Evolve to satisfy a property

Next we investigated whether the organisms could generate diagrams that contain the known state diagram and also adhere to a system critical property, without guidance to a specific solution. As before, we use the cr-trans instruction to enable organisms to generate diagrams. However, to enable AVIDA to evaluate whether a diagram adhered to a property, we extended it with the ability to utilize external tools. Specifically, we defined two tasks: checkSyntax and checkProperty.

The checkSyntax task performs basic syntax checks on the model and converts the in-memory representation of a model into a format that is amenable to analysis by a model checker. Specifically, it "pretty prints" the generated state diagram in XML. This XML specification is combined with the XML specification of the rest of the model, i.e., the class diagram and other state diagrams. Then Hydra is used to translate the XML representation to Promela. The checkProperty task uses a model checker to determine if the model adheres to a system critical property. Specifically, the model checker verifies that all possible paths through the Promela specification adhere to the property. If so, the generated state diagram represents a portion of a viable target system model.

5. Experimental Results

We conducted a case study in using digital evolution to evolve a state diagram for part of an emergency response system (ERS). The ERS is designed to be deployed in an area affected by a natural disaster, such as a tornado or flood, and to assist rescue workers in assessing the status of a specified area. The ERS comprises hardware sensors that monitor the light level, temperature, and sound content, and a processing component that accumulates and communicates this information. The ERS will dynamically adapt to conserve battery power by ignoring broken or faulty sensors. The ERS will enter a sleep state if the sensed information does not vary for an extended period.

For this example, we focus on the relationship between the processing component and the temperature sensor. Figure 5 is an elided class diagram that depicts the relevant objects. The ProcessingComponent reads temperature information from the TempSensor, which monitors the Environment.

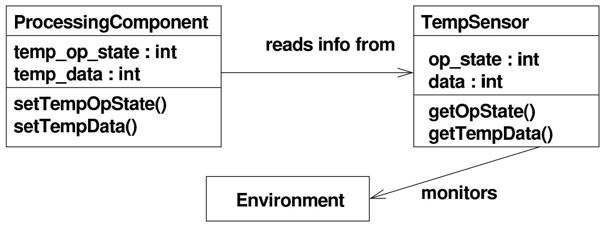


Figure 5. ERS elided class diagram

Our objective was for an AVIDA organism to evolve the known state diagram depicted in Figure 4, and then to further evolve that diagram to meet an additional property. The known state diagram specifies that the ProcessingComponent reads information from the TempSensor using getTempData(), and stores this reading locally using setTempData() – This capability represents the initial behavior of the DAS. The additional property that we wished to evolve extends the behavior of the ProcessingComponent to check the operational state of the TempSensor, and store that setting locally as well. This feature could be used to conserve the battery life of the ERS, for example, by halting all attempts to read data from a non-operational temperature sensor.

Figure 6 depicts an example user-developed state diagram for the ProcessingComponent that includes the known state diagram and satisfies the additional property. This diagram adds one state (DetermineTemperatureOperationalState) and two transitions (/^ TempSensor.getOpState() and setOpState(temp_op_sensor)) to the known state diagram. In general, the ProcessingComponent either requests and then sets data from the TempSensor, or requests and then sets operational state information from the TempSensor. Our expectation was that the AVIDA organisms would generate state diagrams similar to the one depicted here.

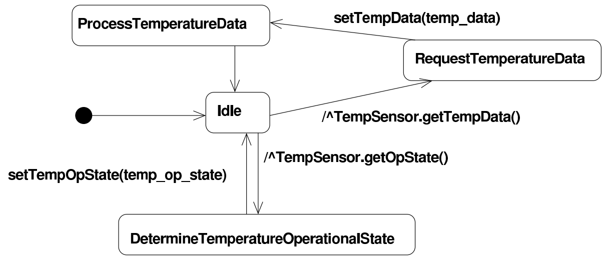

ProcessTemperatureData
setTempData(temp_data)
RequestTemperatureData
Idle
/^TempSensor.getTempData()
/^TempSensor.getOpState()
setTempOpState(temp_op_state)
DetermineTemperatureOperationalState

Figure 6. User developed ProcessingComponent **state diagram**

Figure 7 depicts the state diagram for the TempSensor, which interacts with the ProcessingComponent in two ways. First, in response to a data request (getTempData()) it non-deterministically sets the data and sends it to the ProcessingComponent. Second, in response to an operational state request (getOpState()) it non-deterministically sets its operational state to 0 to indicate that it is operational, or 1 to indicate that it has had a failure, and sends the operational state to the ProcessingComponent. The response of the TempSensor to these events partially determines the possible paths through the generated ProcessingComponents.

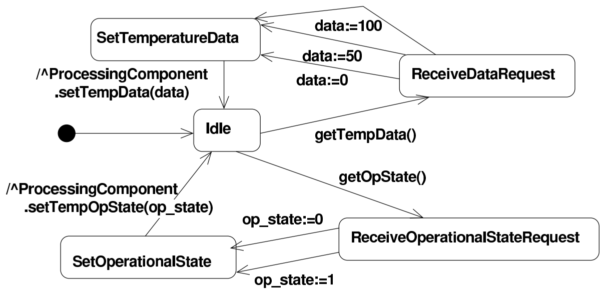


Figure 7. TempSensor **state diagram**

5.1. Experimental Setup

To enable AVIDA organisms to generate a state diagram for the ProcessingComponent, we specified seven tasks to evaluate the generated state diagrams. Specifically, we configured four checkForTrans tasks, called TR0-TR3, such that each checked for a specific transition in Figure 4. We then used the checkSyntax task to check the syntax of generated models and translate them to Promela. Next, we configured a checkProperty task to use the model-checker Spin to assess if the Promela representation of the model met the following property:

Globally, it is always the case that if
TempSensor.op_state==1, then eventually
ProcessingComponent.temp_op_state==1.

This property states that if the TempSensor becomes unavailable, then eventually the ProcessingComponent recognizes that it is unavailable. Because this property is specified as an if-then statement, the verification succeeds by default if the operational state of the TempSensor is never equal to 1. To prevent these false positives, we defined a seventh task (called TR4) that checks that the operational state of the temperature sensor is eventually set to 1. These tasks were configured such that the checkSyntax task would be evaluated only if all of the TR0-TR4 tasks were successful, and the checkProperty task would be evaluated only if the checkSyntax task was successful.

5.2. Experimental Results

We ran AVIDA with a maximum population size of 400 organisms. The population was seeded with a single organism whose genome contained instructions that only enabled the organism to self-replicate (this is the default starting condition for AVIDA experiments). The experiment was run for 7,000 updates, where an update, on average, executes 30 instructions per organism (updates are the standard unit of time in AVIDA experiments).

Figure 8 depicts the number of organisms that evolved to complete the specified tasks. Here we see that more than 75% of all organisms in the population performed tasks TR0-TR4 by update 3000. This means that the majority of evolved state diagrams contained the known state diagram depicted in Figure 4. From update 3000 to 4500 we see that organisms were performing the checkSyntax task; this means that the evolved state diagrams were well-formed, and could be translated to Promela by Hydra. Finally, we see that by update 7000 all seven tasks were performed by greater than 75% of the population. By the end of this experiment, 851 generations, or approximately 340, 400 organisms, generated 26, 705 viable target systems.

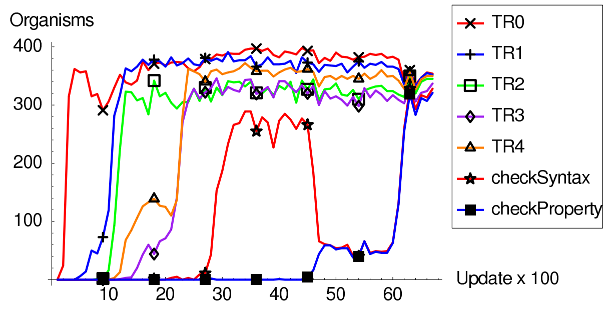


Figure 8. Number of organisms performing each task by update in a representative population. By update 7000, more than 300 of the 400 organisms are performing all tasks; organisms that generate the known state diagram and the desired behavior have been evolved.

Figures 9 and 10 depict two of the many state diagrams that were generated during the evolution of AVIDA organisms. Each of these viable target systems both contain the known state diagram and satisfy the additional property. These figures illustrate the wide variety of unexpected solutions that digital evolution can discover. Observe that neither match the user-developed state diagram depicted in

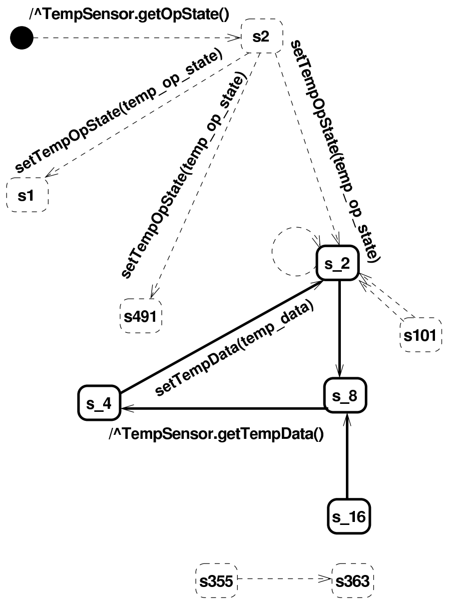

/^TempSensor.getOpState()
s2
setTempOpState(temp_op_state)
s1
setTempOpState(temp_op_state)
setTempOpState(temp_op_state)
s491
s_2
s101
setTempData(temp_data)
s_4
s_8
/^TempSensor.getTempData()
s_16
s355
s363

Figure 9. Complex generated state diagram

Figure 6. In each, thicker lines indicate the states and transitions that were part of the known state diagram, and whose construction were guided by tasks TR0-TR3. The other states and transitions, depicted using dashed lines, were evolved to satisfy the system property. We have labeled the transitions with their developer-assigned string values. However, the states are labeled with the integer labels generated by AVIDA. For Hydra processing purposes, we have prepended an s to each label and have changed negative signs to underscores. Thus, a state labeled by AVIDA as -1 is depicted as s_1.

There are some notable differences between these diagrams. The state diagram depicted in Figure 9 has 11 states and 12 transitions. Many of the elements in this state diagram are unnecessary. For example, states s355, s363, s_16, and s101 are unreachable. In addition, the null transitions connecting s2 to itself, s_16 to s_8, and s101 to s2 are unnecessary. All of these can safely be removed. A less desirable feature of this state diagram is that the operational state of the temperature sensor can only be checked once. Thus, the ProcessingComponent will not be able to determine if the TempSensor fails over time.

In contrast, the state diagram depicted in Figure 10 has 4 states and 6 transitions and no extraneous behavior. To achieve the desired behavior in such a compact model, this diagram relies on non-deterministically following 1 of 2 transitions from state s103 to s107. Although the transition selection from state s107 to s214 appears non-deterministic, it is deterministic due to the behavior of the TempSensor (depicted in Figure 7). Specifically, if the ProcessingComponent calls the getOpState() method of the TempSensor, then the TempSensor will eventually set the temperature operational state by calling the setTempOpState() method of the ProcessingComponent. Similarly, if the ProcessingComponent calls the getTempData() method of the TempSensor, then the TempSensor will eventually set the temperature temperature data by calling the setTempData() method of the ProcessingComponent. Thus, the transition selection from from state s107 to s214 is deterministic. This state diagram is desirable because it enables the ProcessingComponent to continuously monitor the operational state of the TempSensor.

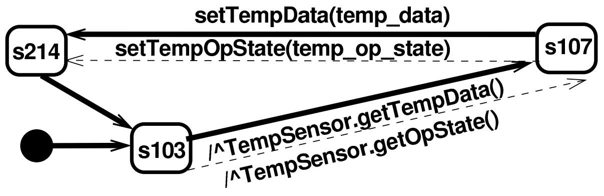

setTempData(temp_data)
s214
setTempOpState(temp_op_state)
s107
s103
/^TempSensor.getTempData()
/^TempSensor.getOpState()

Figure 10. Simple generated state diagram

6. Discussion & Conclusions

We have presented our initial investigations into using digital evolution to evolve organisms that generate state diagrams that represent a portion of a viable target system for a DAS. Specifically, we extended AVIDA to enable an organism to generate a state diagram and to evaluate whether this diagram represents a known state diagram and adheres to critical properties. This technique is a first step in the automatic exploration of the target system requirements space for a DAS.

Our ongoing work addresses several open questions. First, we are investigating various performance improvements, including the use of a high performance computing cluster, to accelerate the speed at which experiments can be performed. Currently, every time an organism generates a state diagram we execute Hydra to translate the diagram to Promela. Then each Promela translation is evaluated with Spin. These steps happen regardless of whether an identical state diagram has been previously evaluated, potentially duplicating work. Furthermore, Spin is a computationally-intensive process, even when not used in combination with AVIDA. Once organisms have evolved to the point where they generate models that are evaluated with Spin, the rate of the experiment slows drastically, as Spin is being executed hundreds of times per update. Since we are intending for AVIDA to be used offline during development time to identify viable target systems, time for discovering candidate models was not an issue for our preliminary studies. Second, we are working to incorporate standard software engineering metrics, such as maximizing cohesion and minimizing coupling, as tasks to evaluate the generated models. We expect these metrics to reduce the number of complex models generated and increase the scalability of this approach. Third, our current experiments rely upon the developer to specify the labels for the transitions. We are investigating using AVIDA to evolve the guards, triggers, and actions that comprise transitions. Fourth, our current experiments evolve a state diagram for one of the classes in a system. Future experiments will attempt to evolve the state diagrams for all of the important classes in the system, perhaps in a collaborative evolution approach. Fifth, our current experiments require the developer to utilize checkForTrans tasks to guide the organisms in the construction of a specific source system model. We are currently evaluating an approach that seeds the evolutionary process with the known state diagram by embedding it within each organism at birth, similar to "instinctual" knowledge. We expect this approach to decrease the time required for organisms to discover viable target systems.

References

[1] B. Ali, A. E. A. Almaini, and T. Kalganova. Evolutionary algorithms and their use in the design of sequential logic circuits. *Genetic Programming and Evolvable Machines*, 5(1):11–29, March 2004.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382, 1998.

[3] O. Babaoglu, G. Canright, A. Deutsch, G. A. D. Caro, F. Ducatelle, L. M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes. Design patterns from biology for distributed computing. *ACM Transactions on Autonomous and Adaptive Systems*, 1(1):26–66, 2006.

[4] D. Bradley, C. Ortega-Sanchez, and A. Tyrrell. Embryonics + immunotronics: A bio-inspired approach to fault tolerance. In J. Lohn, A. Stoica, and D. Keymeulen, editors, *The Second NASA/DoD workshop on Evolvable Hardware*, pages 205–224, Palo Alto, California, 13-15 2000. IEEE Computer Society.

[5] D. W. Bradley and A. M. Tyrrell. Immunotronics - novel finite-state-machine architectures with built-in self-test using self-nonself differentiation. *IEEE Trans. Evolutionary Computation*, 6(3):227–238, 2002.

[6] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In P. Donohoe, editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer, 1999.

[7] S. S. Chow, C. O. Wilke, C. Ofria, R. E. Lenski, and C. Adami. Adaptive radiation from resource competition in digital organisms. In *Science*, volume 305, pages 84–86, 2004.

[8] M. S. Feather, S. Fickas, A. V. Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD '98: Proceedings of the 9th International Workshop on Software Specification and Design*, 1998.

[9] S. Goings, J. Clune, C. Ofria, and R. T. Pennock. Kin selection: The rise and fall of kin-cheaters. In *Proceedings of the Ninth International Conference on Artificial Life*, pages 303–308, Boston, MA, USA, September 2004.

[10] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

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

[12] S. Johnson. *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*. Scribner, 2002.

[13] D. B. Knoester, P. K. McKinley, B. Beckmann, and C. A. Ofria. Evolution of leader election in populations of self-replicating digital organisms. Technical Report MSU-CSE-06-35, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, December 2006.

[14] D. B. Knoester, P. K. McKinley, and C. A. Ofria. Using group selection to evolve leadership in populations of self-replicating digital organisms. Technical Report MSU-CSE-07-7, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, February 2007.

[15] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Genetic Programming. Springer, 1st edition, 2005.

[16] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[17] S. S. Kulkarni and K. N. Biyani. Correctness of component-based adaptation. In I. Crnkovic, J. A. Stafford, H. W. Schmidt, and K. C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 2004.

[18] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu. Towards requirements-driven autonomic systems design. In *DEAS '05: Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, pages 1–7, St. Louis, MO, USA, 2005.

[19] R. E. Lenski, C. Ofria, T. C. Collier, and C. Adami. Genome complexity, robustness, and genetic interactions in digital organisms. In *Nature*, volume 400, pages 661–664, 1999.

[20] R. E. Lenski, C. Ofria, R. T. Pennock, and C. Adami. The evolutionary origin of complex features. In *Nature*, volume 423, pages 139–144, 2003.

[21] J. Lohn, G. S. Hornby, and D. Linden. Evolution, re-evolution, and prototype of an x-band antenna for nasa's space technology 5 mission. In *Sixth International Conference on Evolvable Systems: From Biology to Hardware*, 2005.

[22] A. Martinoli and F. Mondada. Collective and cooperative group behaviours: Biologically inspired experiments in robotics. In *Proceedings of the Fourth Symposium on Experimental Robotics ISER-95*, June 1995.

[23] 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.

[24] D. Misevic, R. E. Lenski, and C. Ofria. Sexual reproduction and muller's ratchet in digital organisms. In *Ninth International Conference on Artificial Life*, pages 340–345, Boston MA, 2004.

[25] R. Neville, A. Sutcliffe, and W. Chang. Optimizing system requirements with genetic algorithms. In *In IEEE World Congress on Computational Intelligence*, pages 495–499, May 2003.

[26] Object Management Group. http://www.omg.org, 2006.

[27] C. Ofria and C. Adami. Evolution of genetic organization in digital organisms. In *Proc. of DIMACS workshop Evolution as Computation*, pages 167–175, Princeton, NJ, 1999.

[28] C. Ofria, C. Adami, and T. Collier. Design of evolvable computer languages. In *IEEE Transactions in Evolutionary Computation*, volume 17, pages 528–532, 2002.

[29] C. Ofria and C. O. Wilke. Avida: A software platform for research in computational evolutionary biology. In *Journal of Artificial Life*, volume 10, pages 191–229. International Society of Artificial Life (ISAL), March 2004.

[30] M. Pagel. *Encyclopedia of Evolution*, pages E83–E92. Oxford University Press, New York, NY, USA, 2002.

[31] R. Schoonderwoerd, J. L. Bruten, O. E. Holland, and L. J. M. Rothkrantz. Pheromone robotics. *Autonomous Robots*, 11(3):319–324, 2004.

[32] J. Suzuki and T. Suda. A middleware platform for a biologically-inspired network architecture supporting autonomous and adaptive applications. In *IEEE Journal on Selected Areas in Communications (JSAC), Special Issue on Intelligent Services and Applications in Next Generation Networks*, February 2005.

[33] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[34] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff. NASA's swarm missions:the challenge of building autonomous software. *IT Professional*, 06(5):47–52, 2004.

[35] C. O. Wilke, J. L. Wang, C. Ofria, C. Adami, and R. E. Lenski. Evolution of digital organisms at high mutation rate leads to survival of the flattest. In *Nature*, volume 412, pages 331–333, 2001.

[36] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM Press.