Optimal Control of Software Using the Markov Decision Processes

Marian CRISTESCU, Gabriel SOFONEA
"Lucian Blaga" University of Sibiu

In this article we present a technique for requirements analysis and evolution for problems involving the optimal control of a software or physical system. The requirements take the form of a function indicating when the system has reached a desired state. A solution which meets the requirements takes the form of a controller which specifies how the system should act. A human operator iteratively analyses and refines a given presentation of the requirements on a human readable high symbolic level and evaluates the resulting solution by the means of a graphical display.
***Keywords**: computational intelligence, intelligent mobile agent, human computer interface, evolutionary computation, software system.*

Introduction

Our approach aims towards a fruitful synthesis of requirements engineering (RE) and computational intelligence (CI). We target problems involving the optimal control of a software or physical system. Requirements take the form of a function indicating when the system has reached a desired state. A solution which meets the requirements takes the form of a controller which specifies the behavior of the system. We use methods from CI to automatically compute solutions to a given set of requirements. Specifically, we use a form of stochastic dynamic programming called reinforcement learning. Our approach requires that the representation of the requirements is machine-readable, i.e., that the algorithm can use the representation as input to automatically compute a solution to the given set of requirements and the representation of the requirements is human-readable, i.e., that a human operator can understand the representation of the requirements in such a way that it is easy for her to relate changes/refinements in the requirements to the solution. In this context we utilize Markov Decision Processes (MDPs) for the requirements representation to ensure, i.e., the representation of the requirements in a machine-readable format. To ensure, i.e., how to represent the requirements in a human-readable format we draw connections to the field of human computer interfaces (HCIs).

The requirements could then be handed off to a better learning algorithm, to find a better solution, or passed on in a human-readable format.

2. Automated Requirements Engineering

In the introduction we present MDPs as an underlying framework for RE methodology. MDPs are a general way to model sequential decision making problems. In the MDP framework an agent is trying to reach a predefined goal by interacting with the environment. MDPs have been utilized in as diverse problem domains as airline meal provisioning [9], goal management in organizations [5], spoken dialogue systems [2], and planetary exploration [6].

2.1 Anatomy of an MDP

We will focus on finite, discrete, infinite horizon MDPs. An MDP is called finite if it consists of a finite number of states and actions. A discrete MDP consists of discrete states and actions. An infinite horizon indicates that there is no absorbing or end state. A discrete MDP is a tuple P = hS,A, T, ,Ri, that consists of:

- a state space S = {s1, s2, . . . , sN}, of cardinality |S| = N,
- a set of primitive actions A = {a1, a2, . . . , ak}, of cardinality |A| = k,
- a transition function T : S × A × S ! [0, 1],
- a discount factor 2 (0, 1], and
- a reward function R : S ! R1)

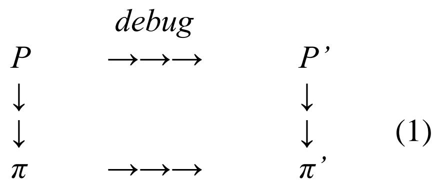
An agent acting inside an MDP perceives at each time step *t* the current state *s € S* of the environment and chooses an action *a € A* from a set of possible actions which results into a relocation of the agent to a new state, *s'*, according to the dynamics of the envi-

ronment specified by the transition function $T(s'|s,a)$, whereupon the agent receives a numerical reward according to the reward function $R(s)$. The transition function must represent a valid probability distribution.
In the MDP framework an agent is acting towards a goal that is specified by the reward function. The reward function tells the agent how well it is performing and the goal of the agent is to maximize the discounted sum of rewards over time. The discount factor determines the value of future rewards. In the context of RE, the reward function equals the set of requirements.
The behavior of the agent is called a policy, $\pi : S \rightarrow A$, which determines an action for the agent for any possible state. Note, that policies can be stochastic mappings, $\pi : S \times A \rightarrow [0,1]$.
From the perspective of RE, a policy is a solution to a given set of requirements, embodied in a reward function. In conventional software engineering (SE) the programmer is trying to build a program according to a given set of requirements that should result in the desired behavior of the executable code. In our methodology we are trying to build a reward function (=set of requirements) that should result in the desired behavior of the resulting policy/controller. Much like in conventional programming it can be hard to know if the resulting behavior of the policy will be the desired behavior. To tackle this problem we are debugging the policy by step-wise altering parameters of the underlying MDP. This is similar to modifying conventional program code with a debugger and therefore altering the behavior of the resulting application. We can think of the MDP as the program code in a SE project and the resulting learned policy as the executable application. The learning step in our case which is needed for computing an optimal policy (=example solution) is analogous to the compile step in software engineering. An MDP also needs an initial state, $s0$, or an initial distribution over states, $S0$. The model is Markov if the state transitions are independent of any previous environment states or agent actions [1].

2.2 MDP Life-cycle
An analogy between SE and MDPs illustrates the coarse concept of the RE methodology: In SE the basic software life-cycle determines the evolution of a program. The software life-cycle is a term used to describe the various phases through which software travels. A classic software process model used in SE is the waterfall model of the software life-cycle. The phases of the waterfall model are analysis, design, implementation, and test. Other software process models used in SE are, e.g., the spiral model [8] and extreme programming (XP) [3]. The idea of the software life-cycle is also true for MDPs. The hypothetical "MDP life-cycle" basically consists of the same phases as the software life-cycle. The difference is that instead of maintaining software, we are maintaining MDPs and policies (=agent behaviors).

2.3 Iterative Requirements Analysis and Evolution
The set of requirements are represented by the reward function of an MDP. If the learned solution w.r.t. the given requirements does not produce the desired behavior, a human operator iteratively modifies parameters of the underlying MDP while observing the agent behavior which changes according to the alterations of the MDP parameters. This basic mode of operation forms a kind of feedback loop:
1. Observe agent behavior π — Is the agent doing the right thing w.r.t. a given set of requirements?;
2. Modify requirements embodied in the reward function of an MDP $P \rightarrow P'$, and;
3. Compute an example solution π'' of the altered MDP P' based on P.
Repeat steps 1, 2 and 3 until the agent is doing the right thing.
The architecture of RE methodology consists of three basic components (B1 and B2 count as one component):
(A) A visual representation of agent behavior;
(B1) A human readable representation of the underlying MDP and requirements;

(B2) A human computer interface for the purpose of navigating inside the underlying MDP,
and
(C) An algorithm for quickly solving related MDPs.



(1)

The iterative evolution of requirements and solutions can best be illustrated as solving sequences of related MDPs. Diagram 1 illustrates two related MDPs *P* and *P'* and their respective policies *π* and *π'*. With each iteration of the process a slightly modified new MDP *P'* based on the previous old version of MDP *P* is created. The idea is that the human operator modifies the underlying MDP only a little with each debug step. In order to see the effects of this change it is necessary to solve the new MDP *P'* based on the old MDP *P* and the old policy *π*.

3. Requirements Engineering Approach

The RE approach has direct connections and interrelations to a variety of different fields of study, i.e. RL (related to component C), SE (related to component B2), Human Computer Interfaces (HCI) (related to components A, B1, B2), and Utility theory and Evolutionary Computation (EC) (related to component B1).

3.1 Reinforcement Learning

In the field of RL we think that work on MDPs [20] and quickly solving related MDPs [5] are relevant to the concept of our RE architecture.

Besides the basic idea of our RE methodology as an interactive tool that makes the agent do the right thing, we can use existing learning techniques for the purpose of giving hints to the agent to accelerate the learning process. This can be done manually by e.g. providing sample trajectories and replaying them [3], or by designing a shaping reward function in a restricted editing mode that just allows sound transformations [6], or by using a supplied control policy like in the JAQL framework [3]. Other learning methods for instructing the agent are apprenticeship learning [1], imitation learning [9], shaping [6, 2] and reward functions [7, 4].

An open problem when solving large MDPs are the long learning times. To tackle this problem in the context of the RE architecture an algorithm that can quickly solve related MDPs would be beneficial. Another idea to cope with the long learning times and to provide more immediate feedback to the user is to generate an approximate solution of the MDP that coarsely gives the human user an intuition on how the behavior of the agent will change before finally learning the solution to the altered MDP.

3.2 Software Engineering

From the world of SE we would like to apply general ideas from the field of software debuggers [9] and specific concepts found in a typical state-of-the-art software debugger (e.g., Microsoft's Visual Studio .NET debugger), i.e., watches (for visualizing parameters of the MDP), breakpoints (for marking single states or regions of states in state-space), and edit and continue (a feature that allows to directly continue policy execution after editing parameters of the MDP).

The following list describes the functionality of typical debugger commands found in a conventional debugger in the context of debugging MDPs:

Go: Executes policy from the current state or state-action pair until a breakpoint or the terminal-state is reached, or until the task pauses for user input.

Restart: Restarts agent at start state.

Stop Debugging: Leaves debugging mode.

Break: Halts the agent at its current state or state-action pair.

Apply MDP Changes: Applies parameter changes to MDP.

Step Into: Single-steps through states or state-action pairs w.r.t. current policy, and enters each MDP region that is encountered.

Step Over: Single-steps through states or state-action pairs w.r.t. current policy. If a periphery state is reached, the appropriate MDP region is executed without stepping through it.

Step Out: Executes policy out of an MDP region, and stops on the exit periphery-state. Using this command, you can quickly finish executing the current MDP region after determining that a bug is not present in the MDP region.
Run to Cursor: Executes the policy as far as the state or state-action pair that contains the "cursor" in the human-readable MDP display. This is equivalent to setting a temporary breakpoint at the cursor location. This command can be used to return to an earlier state or state-action pair to retest an agent, using e.g. a different reward function.
Step Into Specific Region: Single steps through states in the policy, and enters the specified MDP region.
Set Next State: Sets the next state or state-action pair. Use this command when you want to rerun a "section" within the current MDP/MDP region or to skip a section of an MDP/MDP region you do not want to execute. For instance, a section that contains a known bug and continue debugging other sections.
Occasionally the debugger is paused in break mode, meaning the debugger is waiting for user input after completing a debugging command (like break at breakpoint, step into/over/out/to
cursor, break after Break command or Restart).
A breakpoint can be used to mark states that are interesting w.r.t. the debugging process in state space; policy execution will stop when that state in state-space (=breakpoint) is reached. Like in a conventional debugger the human operator of the debugger can conveniently inspect the MDP (=program) and change parameters accordingly. Advanced breakpoint syntax is a feature that allows for specifying logical conditions on when a breakpoint is reached. In the paper context this feature would be useful for specifying conditions like positive reward cycles and then notifying the user about that event. Positive reward cycles distract the agent from doing
the right thing and cause bugs like the one introduced in the bicycle task where the bicycle tended to move in circles around the start state.
Finally we want to apply insights about how to design the development process to minimize errors, automate as much as possible, increase usability by using concepts like immediacy [5].

3.3 Human Computer Interfaces

From HCI we want to borrow a human readable representation of MDPs. Specific problems are: How can we display reward functions, shaping potentials and states in a human readable way. Other problems in this direction is, e.g., about analogical representation of programs [8] and software visualization for debugging [2]. Another important problem is that of finding a generic representation of the MDP display, i.e., one that works with an arbitrary problem domain. The question of which instruments/tools should we give to a human operator for the task of modifying the parameters of the MDP is also HCI related.

3.4 Evolutionary Computation and Utility Theory

Work in evolutionary computation (EC) about fitness function design is interesting in connection with reward function design [7] and representation and work in interactive evolutionary computation (IEC) [4] might give rise to ideas about possible problem domains and how to incorporate a human into the system. Similarly in utility theory work on utility function design and utility elicitation is related to reward function design.

4. Conclusions

Building upon the idea of the RE methodology it would be possible to construct an Integrated Development Environment (IDE) for MDP design. The debugger would be a subcomponent of the IDE. The MDP-IDE should provide the means for assembling a model of the world, e.g. different physics modules could be hooked up to produce a model, different standard reward functions (reward at goal, reward for moving towards goal, etc.)
An important aspect related to this paper is to design a standard interface for the modules. Another idea for a sub-component of the

MDP-IDE is a profiler that displays statistics of MDPs. The profiler is not for fixing policies but for fine tuning for best performance, e.g., altering the reward function so that it is robust to small perturbations in the world dynamics, or modifying parameters of the MDP so that it is more likely to be easily solved by our approximation method.

References

[1] - Abbeel P., and Andrew Y. Ng., "*Apprenticeship learning via inverse reinforcement learning*" Submitted to the Twenty-First International Conference on Machine Learning, 2004;

[2] - Baecker R., DiGiano C., and Marcus A., "*Software visualization for debugging*", Communications of the ACM, 40(4):44–54, April 1997;

[3] - Beck K., "*Embracing change with extreme programming*", IEEE Computer, 32(10):70–77, October 1999;

[4] - Boehm B. W., "*A spiral model of software development and enhancement*", IEEE Computer, 21(5):61–72, May 1988;

[5] - Kaelbling L.P., Littman M.L., and Moore A.P., "*Reinforcement learning: A survey*", Journal of Artificial Intelligence Research, 4:237–285, 1996;

[6] – Levin E., Pieraccini R, and Eckert W., "*Using Markov decision process for learning dialogue strategies*", In Proceedings of the 1998 International Conference on Acoustics, Speech and Signal Processing (ICASSP-98), volume 1, pages 201–204, New York, NY, May 1998;

[7] – Ng A.Y., Harada D., and Russell S., "*Policy invariance under reward transformations: Theory and application to reward shaping*", In Proceedings of the Sixteenth International Conference on Machine Learning, pages 278–287, San Francisco, CA, 1999;

[8] –Takagi H., "*Interactive evolutionary computation: Fusion of the capacities of EC optimization and human evaluation*", Proceedings of the IEEE, 89(9):1275–1296, September 2001;

[9] - Ungar V., Lieberman S., and Fry A.V., "*Debugging and the experience of immediacy*", Communications of the ACM, 40(4):38–43, April 1997.