Automated iterative requirements analysis and evolution[1)]

Werner Gaisbauer and Brian Sallans
Austrian Research Institute for Artificial Intelligence (OeFAI)
Freyung 6/6, A-1010 Vienna, Austria
email: {gaisb,brian}@oefai.at

Abstract:
We present a novel 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. Our approach forms a closed-loop system where the requirements and solutions iteratively evolve towards the desired requirements and solutions over time. In this context we use existing machine learning techniques, i.e., reinforcement learning, to automatically compute a solution for a given set of requirements.

1 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 (1) 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 (2) the representation of the requirements is human-readable, i.e., that a human operator

[1)] The Austrian Research Institute for Artificial Intelligence is supported by the Austrian Federal Ministry for Education, Science and Culture and by the Austrian Ministry for Transport, Innovation and Technology. Support for part of this research was provided by the FWF Austrian Science Fund project S9106-N04.

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 the above context we utilize Markov Decision Processes (MDPs) for the requirements representation to ensure (1), i.e., the representation of the requirements in a machine-readable format. To ensure (2), i.e., how to represent the requirements in a human-readable format we draw connections to the field of human computer interfaces (HCIs). Given (1) and (2) we can close the loop which results in an iterative process by which a human operator can input a set of requirements, see the automatically computed results on the screen, and then go back and modify the requirements if the results are not what she wanted. This feedback based debugging cycle is important because it can be very hard to know how an agent-based system will behave given a certain set of requirements, embodied in a reward function. When the process is finished it results in a set of requirements and an example solution w.r.t. the set of requirements. 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.

Our approach is inherently related to the field of software engineering and more specifically to the area of software debuggers. By using MDPs we are "programming" in a fifth-generation programming language (5GL): we do not have to tell the machine *how* to do something by forming step-wise instructions (e.g., like in the conventional high-level programming language C++) but rather by simply specifying *what* it should do by creating (=programming) and debugging an MDP. The advantage in this case is that the *how* step is automatically computed by the machine learning algorithm.

2 Automated Requirements Engineering

As outlined in the introduction we use MDPs as an underlying framework for our novel RE methodology. MDPs are a general way to model sequential decision making problems. In the MDP framework an agent is trying to reach a pre-defined goal by interacting with the environment. MDPs have been utilized in as diverse problem domains as airline meal provisioning [9], goal management in organizations [15], spoken dialogue systems [12], and planetary exploration [6].

2.1 Anatomy of an MDP

In this paper 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 $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, T, \gamma, R \rangle$, that consists of

- a *state space* $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$, of cardinality $|S| = N$,

- a set of primitive *actions* $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$, of cardinality $|A| = k$,
- a *transition function* $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$,
- a *discount factor* $\gamma \in (0, 1]$, and
- a *reward function* $R : \mathcal{S} \rightarrow \mathbb{R}$[1)]

An agent acting inside an MDP perceives at each time step t the current state $s \in \mathcal{S}$ of the environment and chooses an action $a \in \mathcal{A}$ 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 environment specified by the transition function $T(s'|s, a)$, whereupon the agent receives a numerical reward (positive or negative) according to the reward function $R(s)$. The transition function must represent a valid probability distribution: $\forall s, a \sum_{s' \in \mathcal{S}} T(s'|s, a) = 1$. 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, $r_{tot} = \sum_{t=0}^{\infty} \gamma^t R(s_t)$. The discount factor γ determines the value of future rewards according to $\gamma^t R(s_t)$, where s_t is the state the agent reaches at time t. In the context of RE, the reward function equals the set of requirements. The behavior of the agent is called a *policy*, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, which determines an action for the agent for any possible state. Note, that policies can be stochastic mappings, $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. An optimal policy w.r.t. MDP P, denoted $\pi^*_{\mathcal{P}}$, is a policy that maximizes the reward received by the agent, $E[r_{tot}]$. 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, s_0, or an initial distribution over states, S_0. The model is *Markov* if the state transitions are independent of any previous environment states or agent actions [10]. For a more in-depth discussion of MDPs, see Puterman's text [20] or Bellman's article [4].

[1)] Note, that a more general definition of the reward function is $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$.

An extension of the basic MDP framework concerns the tiling of a monolithic MDP into disjunct regions. The reason why we introduce MDP regions is that they are similar to functions in conventional high level programming languages in the context of debugging MDPs. Here, we explain the mathematical background which is used later on in the paper. As discussed in [11] a *region* is a subset of the state space, $\mathcal{P} \subseteq \mathcal{S}$, along with the corresponding transition and reward functions when restricted to $\mathcal{P}$, $T_{\mathcal{P}} : \mathcal{P} \times \mathcal{A} \times \mathcal{P} \rightarrow [0, 1]$ and $R_{\mathcal{P}} : \mathcal{P} \rightarrow \mathbb{R}$. Thus, the region $\mathcal{P}$ itself constitutes a proper Markov decision process and we can define policies over it. The *exit periphery* of a region, $X_{\mathcal{P}}$, is the set of states having non-zero transition probability out of a region: $X_{\mathcal{P}} = \{s \in \mathcal{P} : T_{\mathcal{S}}(s'|s, a) > 0\}$ for some $s' \in \mathcal{S} \setminus \mathcal{P}$. Informally, the exit periphery states are "doorways": states that must be reached before an agent can transition from one region to another. Corresponding to exit peripheries are entrance peripheries: $I_{\mathcal{P}} = \{s \in \mathcal{P} : T_{\mathcal{S}}(s|s', a) > 0\}$ for some $s' \in \mathcal{S} \setminus \mathcal{P}$.

2.2 MDP Life-cycle

An analogy between SE and MDPs illustrates the coarse concept of our 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

In our approach 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 $\mathcal{P} \Rightarrow \mathcal{P}'$, and

3. Compute an example solution π' of the altered MDP $\mathcal{P}'$ based on $\mathcal{P}$.

Repeat steps 1, 2 and 3 until the agent is doing the right thing.

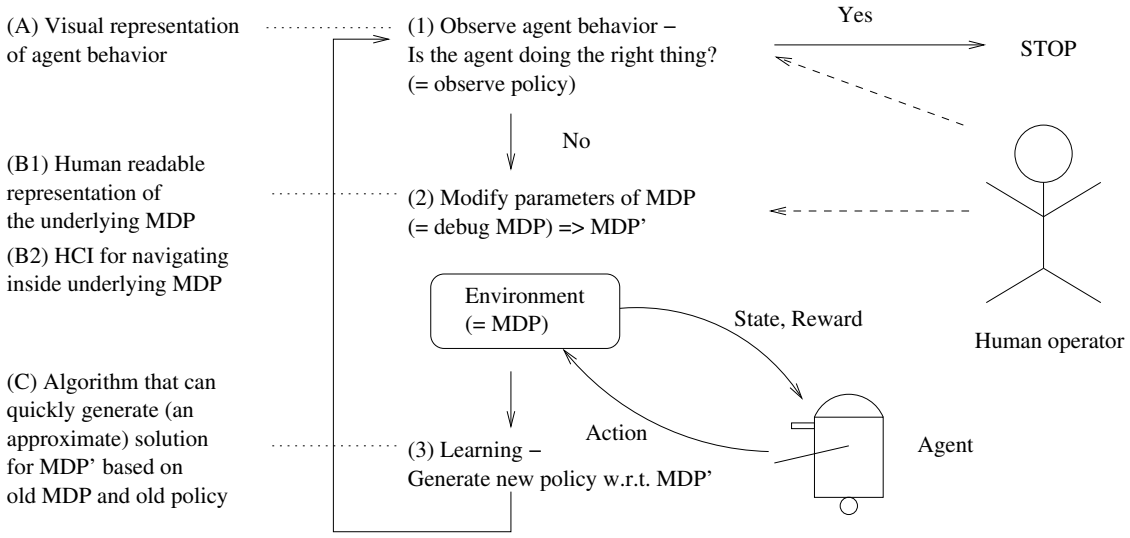

Figure 1: Iterative Requirements Analysis and Evolution

The architecture of our 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.

Figure 1 illustrates the basic operation of our RE method: The three components (A, B, and C) of the architecture are related to the three steps (1, 2, and 3) of the debug loop.

$$\begin{array}{ccc} \mathcal{P} & \xrightarrow{debug} & \mathcal{P}' \\ \downarrow & & \downarrow \\ \pi & \longrightarrow & \pi' \end{array} \tag{1}$$

The iterative evolution of requirements and solutions can best be illustrated as solving sequences of related MDPs. Diagram 1 illustrates two related MDPs $\mathcal{P}$ and $\mathcal{P}'$ and their respective policies π and π'. With each iteration of the process a slightly modified new MDP $\mathcal{P}'$ based on the previous old version of MDP $\mathcal{P}$ 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 $\mathcal{P}'$ based on the old MDP $\mathcal{P}$ and the old policy π.

The next section describes the interrelations of the various components of the RE methodology to different fields of study.

3 Related Work

Our 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] (see figure 1) 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 [13], or by designing a shaping reward function in a restricted editing mode that just allows sound transformations [16], or by using a supplied control policy like in the JAQL framework [23]. Other learning methods for instructing the agent are apprenticeship learning [1], imitation learning [19], shaping [16, 21] and reward functions [17, 14].

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 (see figure 1) 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 [22] 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

*) See section 2.1 for a formal definition of MDP regions and periphery states. The idea is that MDP regions are equivalent to functions in conventional programming code and periphery states are equivalent to entry and exit points in conventional functions.

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 our 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 (see section 2.1).

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 [25].

3.3 Human Computer Interfaces

From HCI we want to borrow a human readable representation of MDPs. Specific problems we will address are: How can we display reward functions, shaping potentials and states in a human readable way. Previous work in this direction is, e.g., about analogical representation of programs [18] 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) [24] 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 Further Work

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 sub-component 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 (e.g., bicycle module, wind module, etc.), different standard reward functions (reward at goal, reward for moving towards goal, etc.) An important aspect related to this work 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] Peter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. Submitted to the Twenty-First International Conference on Machine Learning, 2004.

[2] Ron Baecker, Chris DiGiano, and Aaron Marcus. Software visualization for debugging. *Communications of the ACM*, 40(4):44–54, April 1997.

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

[4] R. E. Bellman. A Markov decision process. *Journal of Mathematical Mechanics*, 6:679–684, 1957.

[5] Daniel S. Bernstein. Reusing old policies to accelerate learning on new MDPs. Technical Report 26, University of Massachusetts, April 1999.

[6] D.S. Bernstein, S. Zilberstein, R. Washington, and J.L. Bresina. Planetary rover control as a Markov decision process. Sixth International Symposium on Artificial Intelligence, Robotics, and Automation in Space, 2001.

[7] Tommaso F. Bersano-Begey and Jason M. Daida. A discussion on generality and robustness and a framework for fitness set construction in genetic programming to promote robustness. In John R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 11–18, Stanford, CA, 1997. Stanford University Bookstore.

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

[9] Jason H. Goto, Mark E. Lewis, and Martin L. Puterman. Coffee, tea, or ...?: A Markov decision process model for airline meal provisioning. *Transportation Science*, 38(2):107–118, February 2004.

[10] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[11] Terran Lane and Leslie Pack Kaelbling. Toward hierarchical decomposition for planning in uncertain environments. In *Proceedings of the 2001 IJCAI Workshop on Planning under Uncertainty and Incomplete Information*, pages 1–7, Seattle, WA, 2001. AAAI Press.

[12] E. Levin, R. Pieraccini, and W. Eckert. 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. IEEE.

[13] Long-Ji Lin. Programming robots using reinforcement learning and teaching. In T. L. Dean and K. McKeown, editors, *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 781–786, Menlo Park, CA, July 1991. MIT Press.

[14] Maja J. Mataric. Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 181–189, San Francisco, CA, 1994. Morgan Kaufmann.

[15] C. Meirina, Y.N. Levchuk, G.M. Levchuk, K.R. Pattipati, and D.L. Kleinman. Goal management in organizations: A Markov decision process (MDP) approach. Command and Control Research and Technology Symposium, Naval Postgraduate School, Monterey, CA, September 2002.

[16] Andrew Y. Ng, Daishi Harada, and Stuart Russell. 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. Morgan Kaufmann.

[17] Andrew Y. Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 663–670, San Francisco, CA, 2000. Morgan Kaufmann.

[18] Damien Ploix. Analogical representations of programs. In *First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, pages 61–69, Los Alamitos, CA, June 2002. IEEE Computer Society.

[19] Bob Price and Craig Boutilier. Implicit imitation in multiagent reinforcement learning. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 325–334, San Francisco, CA, 1999. Morgan Kaufmann.

[20] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, NY, 1994.

[21] Jette Randløv. *Solving Complex Problems with Reinforcement Learning*. PhD thesis, University of Copenhagen, September 2001.

[22] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, New York, NY, 1996.

[23] William D. Smart. *Making Reinforcement Learning Work on Real Robots*. PhD thesis, Brown University, May 2002.

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

[25] Ungar, Lieberman, and Fry. Debugging and the experience of immediacy. *Communications of the ACM*, 40(4):38–43, April 1997.