From: AAAI Technical Report SS-93-03. Compilation copyright © 1993, AAAI (www.aaai.org). All rights reserved.

Design of a Reactive System Based on Classical Planning

John L. Bresina*
Federal Systems Group, Sterling Software Inc.
AI Research Branch, M/S 269-2, NASA Ames Research Center
Moffett Field, CA 94035
bresina@ptolemy.arc.nasa.gov

Abstract

This paper presents a design for a reactive system based on the classical planning techniques of problem reduction and state space search. The proposed approach enables a reactive system to be scaled up to handle larger sets of tasks. Problem reduction synthesizes an initial reactive policy for a given task. When an execution impasse occurs, state space search finds critical choice points, from which control rules are synthesized. These rules alter the policy's future behavior in order to avoid the impasse. This technique, called *critical choice planning*, incrementally debugs the initial policy in the least restrictive way; this "least restrictive property" makes the technique a perfect match for problem reduction. Over time, the problem reduction rules are improved via learning from the debugging experiences.

Introduction

In this paper, we present a design for a reactive system based on the classical planning techniques of problem reduction and state space search. The proposed approach enables a reactive system to be scaled up to handle larger sets of user-provided, or top-level, goals. This approach grew out of work on the Entropy Reduction Engine architecture [4; 3; 9], and like ERE, the proposed reactive system design integrates planning and reaction.

First, we present background material regarding reactive systems. Second, we sketch our overall approach to scaling up such systems. Then, we discuss in more detail the role of problem reduction and a type of state space search called "critical choice planning". The concluding section states the implementation status of this design effort and future work. This section also discusses the improvement of the reduction rules via learning; this discussion is brief because it is not germane to the primary themes of this symposium and it is the most preliminary aspect of the research effort.

*Also affiliated with Rutgers University.

Reactive systems

Current research on *reactive systems* can be viewed as an attempt to shift the focus from plan generation to execution. In contrast to a plan generation system, a reactive system senses and affects the external environment in which it is situated.

Most approaches to the construction of a reactive system's control program, or *reactive policy*, can be viewed in terms of programming, learning, or planning. Examples of the first category include: directly building an instance of a subsumption architecture [6], coding an instance of a Procedural Reasoning System [11], or specifying a situated automata [21] using the REX [12] or GAPPS [13] languages. Within AI research, the most common approach to learning reactive policies has been via some style of reinforcement learning; e.g., temporal difference [24], real-time dynamic programming [1], or Q-learning [25]. Construction of reactive policies via planning is used in, for example, RoboSOAR [15], Theo [17], and ERE [8; 4].

This work's primary objective is to scale up the number of top-level goals that a reactive system can solve without a substantial increase in the space required to store the reactive policy and without a substantial increase in the time required to construct that policy. Systems that utilize the programming approach tend to have a single implicit, top-level goal built into the system by the programmer. Likewise, the learning approaches have assumed a fixed goal. There has not been much research in scaling up these two types of approaches to handle a broad range of goals. The planning approaches have more flexibility in terms of goals; that is, typically, they accept as input explicit top-level goals. The approach presented here attempts to better address the issue of scale than previous planning approaches.

Overall approach

How can reactive systems be scaled up to have competence over a broad range of top-level problems? One answer is based on the well-known principle of modularity – by organizing a reactive system's problem solving expertise into a small set of "modules" that can be flexibly combined to solve problems from the intended domain of competence. The key aspect of this approach is effectively achieving *flexible composition* and,

thus, increasing the transferability of the reactive system's expertise.[1] Our proposed modular approach is cast within the framework of problem reduction. The building blocks of this modular organization are *primitive subproblems* and *primitive reactive policies.* The knowledge of how to use these building blocks to solve a class of domain problems is encoded in a set of problem reduction rules. This reduction approach can more easily scale up because, instead of storing a complex reactive policy for every potential top-level problem, only a small set of relatively simple reactive policies is stored.

The overall approach can be summarized as follows. Problem reduction synthesizes an initial, *non-primitive* reactive policy for a given top-level problem instance by decomposing the given problem into primitive subproblems and then composing the appropriate primitive reactive polices. A *reactive policy* is a partial function from a world state to a set of recommended actions. An execution impasse occurs when this function is undefined (i.e., the policy has no recommendation) for the current state. Upon impasse, state space search is used to find critical choice points. A *critical choice point* [7] is one from which at least one alternative choice leads to necessary failure and at least one alternative choice does not lead to necessary failure (note, it does not have to lead to necessary success).[2] For each critical choice, a critical situated control rule (SCR) is synthesized. A *situated control rule* [7] maps a state to a set of recommended actions. These rules alter the non-primitive policy's future behavior in order to avoid the impasse. This synthesis of a plan expressed as critical SCRs is called *critical choice planning.* The debugging knowledge gained is used to modify the problem reduction rules so that future non-primitive policies synthesized will have a lower probability of an execution impasse.[3]

Why is critical choice planning used to debug the non-primitive reactive policy? One of the key difficulties in synthesizing a non-primitive policy via problem reduction is avoiding conflicts between the primitive policies that are composed. Two policies conflict if there exists a state for which both policies are defined and they recommend disjoint action sets. Avoiding such conflicts is difficult because a given primitive policy may be part of many different compositions. Hence, modifying a policy so that it does not conflict in a given problem context may cause it to conflict in another context. However, a critical situated control rule only prunes out policy recommendations that lead to necessary failure and does not impose any other preference (or bias) on the rest of the behaviors that can be produced by a primitive policy. Hence, such rules are the *least restrictive* advice necessary to avoid impasses. This "least restrictive property" makes critical choice planning a perfect match for problem reduction, because the probability of conflict during policy composition is minimized.

This completes the summary of the overall approach proposed for the design of a reactive system. The next two sections give more details on the synthesis and execution of non-primitive reactive policies and on the incremental debugging of the initial policy, respectively. Implementation status is briefly discussed in the final section, as is the reduction rule learning process.

Problem reduction: An old solution to a new problem

Standard problem reduction operates by applying *non-terminal* reduction rules to recursively decompose problems (abstract situation–goal pairs) into conjunctions of simpler subproblems until primitive subproblems are recognized by *terminal* reduction rules which specify their "obvious" solutions [19]. In terms of this organization of reactive expertise, there are three primary aspects of problem reduction knowledge: (i) *solution* expertise – how to recognize primitive subproblems for which a solution is known (i.e., for which a primitive reactive policy schema exists); (ii) *composition* expertise – how to compose primitive policies to synthesize a non-primitive policy that solves a conjunction of subproblems; and (iii) *decomposition* expertise – how to decompose top-level problems to maximally utilize the problem solving expertise represented by the set of primitive subproblems. The first type of expertise is encoded in terminal reduction rules, and the second two types are encoded in non-terminal reduction rules.

What kind of subproblems should be regarded as primitive? One desirable characteristic is "commonality"; a *common* subproblem is one that is part of the decomposition of many top-level problems. Another desirable characteristic is "independently solvable"; that is, (ideally) the manner in which a subproblem is solved should not depend on the other subproblems in the decomposition. Independence is important for reducing the complexity of resolving subproblem interactions, which is a key difficulty for problem reduction approaches. The main objective of the reduction learning process is to increase the degree of subproblem independence in the reduction rule set.[4]

We assume the system starts with the following knowledge: a set of primitive common subproblem classes, a set of primitive reactive policy schema, a set of terminal reduction rules which encodes the association between the primitive subproblems and the primitive policies, and a set of non-terminal reduction rules which encode "simple" decomposition knowledge but do not encode composition knowledge, i.e., they do not take into account subproblem interactions. After input of a top-level problem instance, the proposed operation of the reactive system can be summarized as follows. First, a

[1] Brooks' subsumption architecture can be viewed as a type of modular approach; however, the issue of flexibly composing a set of modules to solve different top-level tasks has not yet been addressed. An architecture instance is built with a fixed composition designed for the single intended task.

[2] Note that, within the reduction framework, a critical choice point is with respect to a particular subproblem, rather than the top-level problem.

[3] There is also an impasse-triggered debugging process in SOAR [14]. However, SOAR uses a different recovery method and does not employ problem reduction.

[4] See [16] for a related effort, in which an impasse in the reduction search triggers the learning of new decompositions.

search through the reduction (And-Or) space is carried out. During reduction, the top-level problem is recursively decomposed until the conjunctive subproblems are recognized by terminal reduction rules.

The solution of this reduction search is an And-tree that encodes a non-primitive reactive policy. This reduction tree is then executed (as described below) until either the desired goal is satisfied or an impasse occurs. Upon impasse, critical choice planning is used to debug the reduction tree. A recovery plan is synthesized and executed to bring the agent from the dead-end state to one of the critical choice points, and then execution resumes using the modified reduction tree.

Each node in the reduction tree has a *while-condition* which expresses the applicability of the subtree rooted at the node; i.e., *while* this condition is true, the subproblem should be pursued. Each intermediate node specifies the execution order of its subproblems, and each leaf node specifies a primitive reactive policy (and its arguments). A primitive reactive policy returns a set of recommended actions. The recommendation is based solely on the current world model; the policy maintains no other historical information, nor any internal state variables. This "reactive" aspect enables the prediction of the behavior of a policy given an arbitrary world state, which is necessary for critical choice planning.

For illustration purposes, we use a retrieval task within the NASA TileWorld domain [20]. The task involves retrieving a tile from the NE corner of room B and carrying it back to the SW corner of the room A. The agent (which has four grippers) is initially in the SW corner of room A. The agent can move, grasp, or release in a compass direction (S, E, W, or N). As can be seen in Figure 2 (a snapshot during an execution trace), room A is in the SW corner of the grid, room B is in the NE corner, and there are immovable obstacles in both rooms. Figure 1 shows a small portion of a reduction solution (And-tree) for the retrieval task. The top-level problem (node `rn0` in Figure 1) is decomposed into getting the tile in room B and then bringing it back to room A. The first subproblem (node `rn1`) is decomposed into first going to a door in room A, secondly going to a door in room B, thirdly entering the room, and fourthly getting the tile. This fourth subproblem (node `rn6`) is decomposed into going beside the tile (leaf node `rn11`) and then grasping the adjacent tile. To get beside the tile, the `go-toward` primitive policy is used, which non-deterministically reduces the Manhattan distance from the destination (in this case, the NE corner of room B).

The reduction tree is directly executed by first sensing the current environment and updating the world model; and then, starting at the root node, evaluating the while-conditions of subproblems in the order specified by each node's execution orders until an enabled leaf node (i.e., primitive subproblem) is found. The associated primitive policy is evaluated and one of the recommended actions is randomly chosen and executed; i.e., the policies can be non-deterministic. Note that, in each sense-act cycle, the policy interpretation starts at the root node, rather than continuing to execute a selected primitive policy until the associated while-condition becomes

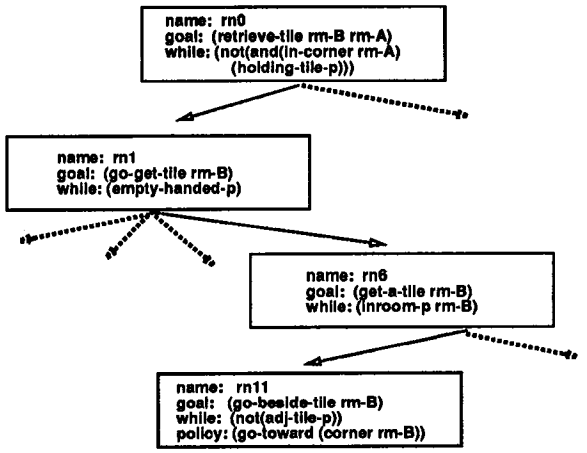

Figure 1: Portion of example reduction tree.

false.[5] Like primitive reactive policies, this reduction tree maps world states to recommended actions; hence, we refer to such a tree as a *non-primitive* reactive policy. The next section presents details on how the initial non-primitive policies are incrementally debugged.

Critical choice planning: A new twist on an old method

The debugging process is invoked when execution reaches an impasse in the context of one of the primitive subproblems (uniquely identifiable from the reduction tree). The impasse could be due to a bug in the subproblem's primitive policy; the bug being a lack of coverage for the current situation. Or the impasse could be due to a subproblem interaction; that is, the manner in which a previously executed primitive policy solved its subproblem has made it impossible for the current policy to solve its subproblem. We refer to this previous subproblem as the *interacting* subproblem.

A trace of the actions taken during execution is kept, and upon impasse a state space is initialized by projecting this trace, ending in the dead-end state. Then a backtracking search continues from this state until a solution is found for the current subproblem. The search space consists of states reachable by the reactive policy from states in the execution trace. If the impasse is due to a bug in the primitive policy, then the search will never leave the current subproblem's subspace. However, if the impasse is due to a subproblem interaction, the search will have to backtrack over subproblem boundaries until it reaches the interacting subproblem. Then another solution must be found to the interacting subproblem that is extendable to solve the impasse subproblem.

After a solution is found, critical choice planning searches in the relevant subproblems' subspaces. If the impasse was due to a subproblem interaction, then the relevant subproblems are the impasse subproblem, the interacting subproblem, and all subproblems in between (w.r.t. execution order). Otherwise, the only states searched are those of the impasse subproblem reachable

[5] If the agent's environment does not require this degree of reactivity, policy interpretation can be streamlined.

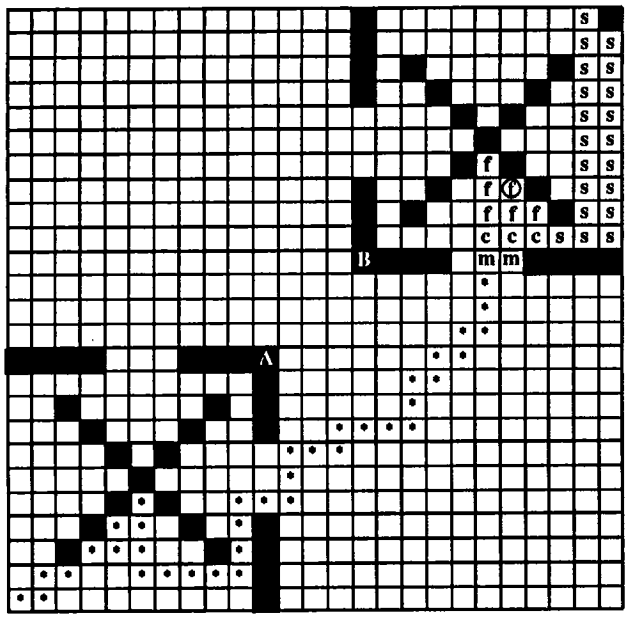

Figure 2: Dead-end in subproblem `(go-beside-tile rm-B)` due to lack of coverage in its primitive policy.

from the subproblem's initial situation (within the execution trace). Critical choice points are found by a node labeling process with respect to the impasse subproblem. Each node in the state space is labeled with one of the following: necessary failure, necessary success, critical choice, or mixed (i.e., none of the above). The labeling process begins at the possible end states (i.e., leaf nodes) for the impasse subproblem, each of which is either a necessary failure or a necessary success. Other nodes are labeled based on the labels of their children; for example, if all the children of a node are critical choices, then the node is labeled as "mixed".

Figure 2 illustrates part of this process. The agent (indicated by a circle to the SE of room B's center) has encountered a dead-end because it can not reduce the Manhattan distance to the tile in the NE corner. The impasse subproblem is `(go-beside-tile rm-B)`, node `rn11` in Figure 1. The asterisks from the agent's initial location in room A to room B indicate the action history trace. The cells labeled "c" indicate critical choice points for this subproblem; "f" indicates necessary failure; "s" indicates necessary success; and "m" indicates mixed. Note that only a small portion of the top-level problem's state space is searched – even the state space of the impasse subproblem is not exhaustively searched.

A situated control rule (SCR) is then synthesized for each critical choice point. Its antecedent is a (partial) state description and its consequent is the set of all the actions that do not lead to necessary failure (i.e., those that produce children nodes labeled either necessary success, critical choice, or mixed). For example, the SCR for the critical choice point that is the most western "c" in Figure 2 consists of the consequent `((move 'E))` and the antecedent of `(and (agent-loc (19 15)) (empty-handed-p))`. The critical SCRs are stored in the reduction tree with the affected primitive subproblem. During subsequent execution, when this subproblem is reached, the SCRs' recommendations are followed whenever applicable; i.e., they have priority over the subproblem's primitive policy.

Finally, a shortest path from the dead-end to one of these critical choice points is found and the inverses of the path's actions are executed to recover from the impasse. In the example, the recovery is `((move 'S) (move 'S))`. Then execution continues using the modified non-primitive policy; eventually, the goal is satisfied.

The non-determinism of reactive policies (primitive and non-primitive) implies that policies can produce a set of different behaviors. The *goal satisfaction probability* of a reactive policy is the proportion of the possible behaviors that satisfy the goal. We assume that the initial non-primitive policy has a non-zero goal satisfaction probability.[6] Through execution experience and debugging, this probability can be increased as much as required.[7] The debugging process as presented here is incremental and it has the usual advantages (and disadvantages) of incremental approaches. For instance, the debugging effort is focussed on the impasses encountered in actual experience; low probability bugs may never need to be overcome. Depending on the problem domain – especially with regards to how costly execution failures are – some (or even all) of the debugging could be done before execution begins. The proposed techniques can be used regardless of when the debugging is carried out.

Concluding remarks

In this final section, we present the implementation status, discuss future work, and make hypotheses regarding the approach's inherent advantages.

A prototype of the critical choice planning technique has been implemented and tested on the retrieval task and related problem instances using the NASA Tile-World simulator [20]. In these exploratory experiments, the initial, non-primitive policy was hand-constructed; in the future, it will be synthesized by a version of the REAPPR problem reduction system [2; 5].

Techniques for assimilating the knowledge gained via debugging are still being explored. We intend to utilize the critical choice points for a primitive policy in two ways: to modify the policy so that it better solves the subproblem, or to modify the associated subproblem so that the (unmodified) policy better solves it. The first method can be accomplished by generalizing the critical SCRs and incorporating them into a terminal reduction rule. One way to accomplish the second method is to create a new non-terminal rule which decomposes the impasse subproblem into subproblems whose search spaces do not include the critical choices.[8] For example,

[6] This assumption can be relaxed if we base the critical choice planning on *reaction-first search* [10], which considers possible actions not recommended by the policy after exhausting the (reachable) policy subspace.

[7] Note that even if we debug a policy until it has a 1.0 goal satisfaction probability, the set of critical SCRs will (almost) never constitute a "universal plan" [23].

[8] A related technique, SteppingStone [22], learns intermediate subgoals for a means-ends problem solver.

the critical choices in Figure 2 can be avoided by introducing the intermediate subproblem of going to the SE corner of room B.

The proposed reactive system design did not constrain how the primitive reactive policies are constructed – programming, planning, or learning could be used. By keeping the primitive subproblems simple enough, the complexity of any of these techniques should be manageable, because the search space of a primitive subproblem can be much smaller than that of a top-level problem. The problem reduction framework not only simplifies the task of constructing the initial set of primitive policies, it also simplifies the tasks of constructing and debugging non-primitive policies. Assuming a reasonable decomposition structure, the critical choice planning search space will be much smaller than the state space for the top-level problem instance. Another advantage of the reduction framework is that encoding learned knowledge within problem reduction rules should yield greater learning transfer than local search control rules, as learned by, e.g., Prodigy [18] and SOAR [14].

The above mentioned advantages of a problem reduction approach in conjunction with the least restrictive property of critical choice planning make these two classical planning techniques a perfect combination for scaling up reactive systems to handle a broader range of top-level problems. Future research will attempt to substantiate these preliminary claims.

Acknowledgements

Thanks to my co-members of the ERE group (past & present): Mark Drummond, Rich Levinson, Andy Philips, and Keith Swanson; thanks to my mentors at Rutgers: Saul Amarel, Charles Schmidt, and Don Smith; and thanks to the other reviewers of previous drafts: John Allen, Wayne Iba, Amy Lansky, Steve Minton, Andrew Philpot, and Ian Underwood.

References

[1] Barto,A.E., Bradtke, S.J., & Singh, S.P. 1991. Real-Time Learning and Control using Asynchronous Dynamic Programming. Tech. rept. 91-57, Computer and Information Sciences, University of Massachusetts at Amherst.

[2] Bresina, J. 1988. REAPPR — An Expert System Shell for Planning. Tech. rept. LCSR-TR-119, LCSR, Rutgers. February.

[3] Bresina, J.L. & Drummond, M.E. 1990. Integrating planning and reaction: A preliminary report. In the working notes of the AAAI Spring Symposium on *Planning in Uncertain, Unpredictable, or Changing Environments.*

[4] Bresina, J., Drummond, M., & Kedar, S. (in press) Reactive, integrated systems pose new problems for machine learning. In *Machine Learning Methods for Planning*, edited by S. Minton. Morgan-Kaufmann.

[5] Bresina, J., Marsella, S., & Schmidt, C. 1986. REAPPR – Improving Planning Efficiency via Expertise and Reformulation. Tech. rept. LCSR-TR-82, LCSR, Rutgers. June.

[6] Brooks, R.A. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1.

[7] Drummond, M. 1989. Situated Control Rules. *Proc. of the KR Conf.* Toronto, Canada.

[8] Drummond, M. & Bresina, J. 1990. Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction. *Proc. of the AAAI Conf.* Boston.

[9] Drummond, M., Bresina, J., & Kedar, S. 1991. The Entropy Reduction Engine: Integrating Planning, Scheduling, and Control. Sigart Bulletin, Vol. 2, Number 4, August. ACM Press.

[10] Drummond, M., Levinson, R., Bresina, J., & Swanson, K. 1992. Reaction-First Search: Incremental Planning with Guaranteed Performance Improvement. Tech. rept. TR-FIA-92-34, NASA Ames.

[11] Georgeff, M.P. & Lansky, A.L. 1987. Reactive Reasoning and Planning. Proc. of the Sixth National Conf. on AI, Seattle, WA.

[12] Kaelbling, L. 1987. REX: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems. *Proc. of AIAA Conf. on Computers in Aerospace.* Wakefield, MA.

[13] Kaelbling, L. 1988. Goals as Parallel Program Specifications. *Proc. of the AAAI Conf.* St. Paul, MN.

[14] Laird, J.E., Newell, A., & Rosenbloom, P.S. 1987. SOAR: An Architecture for General Intelligence. *Artificial Intelligence*, *33*, 1–64.

[15] Laird, J. E. & Rosenbloom, P.S. 1990. Integrating Execution, Planning, and Learning in Soar for External Environments. In *Proc. of the AAAI Conf.*

[16] Marsella, S. & Schmidt, C. (in press) A Method for Biasing the Learning of Nonterminal Reduction Rules. In *Machine Learning Methods for Planning*, edited by S. Minton. Morgan-Kaufmann.

[17] Mitchell, T. M. 1990. Becoming Increasingly Reactive. *Proc. of the AAAI Conf.* Boston.

[18] Minton S., Carbonell, J., Knoblock, C., Kuokka, D.R., Etzioni, O. and Gil, Y., 1989. Explanation-Based Learning: A Problem-Solving Perspective, *Artificial Intelligence*, Vol. 40 (Sept.)

[19] Nilsson, N.J. 1971. Problem Solving Methods in Artificial Intelligence. San Francisco, CA: McGraw-Hill Book Company.

[20] Philips, A.B. & Bresina, J.L. 1991. NASA Tile-World Manual (for System Version 2.0). Tech. rept. TR-FIA-91-11, NASA Ames Research Center.

[21] Rosenschein, S. 1989. Synthesizing Information-Tracking Automata from Environment Descriptions. *Proc. of the KR Conf.* Toronto, Canada.

[22] Ruby, D. & Kibler, D. 1989. Learning Subgoal Sequences for Planning. *Proc. of the IJCAI Conf.*

[23] Schoppers, M. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. *Proc. of the IJCAI Conf.* Milan, Italy.

[24] Sutton, R. 1988. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, Volume 3, Number 1.

[25] Watkins, C.J.C.H. 1989. Learning from Delayed Rewards. Ph.D. thesis, Cambridge Univ., England.