Model-Based Reconfiguration of Schema-Based Reactive Control Architectures

Ashok K. Goel, Eleni Stroulia, Zhong Chen, Paul Rowland
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA
Contact: goel@cc.gatech.edu; (404)-894-4994

Abstract

Reactive methods of control get caught in local minima. Fortunately schema-based reactive control systems have built-in redundancy that enables multiple configurations with different modes. We describe a model-based method that exploits this redundancy, and, under certain conditions, reconfigures schema-based reactive control systems trapped in behavioral cycles due to the presence of local minima. The qualitative model specifies the functions and modes of the perceptual and motor schemas, and represents the reactive architecture as a structure-behavior-function model. The model-based method monitors the reactive processing, detects failures in the form of behavioral cycles, analyzes the processing trace, identifies potential modifications, and reconfigures the reactive architecture. We report on experiments with a simulated robot navigating a complex space.

1 Introduction

Reactive methods for control are characterized by a direct mapping of perceptions of the world to actions on it. A major advantage of these methods is that they result in rapid response. A major problem is that they get caught in local minima [Arkin, 1989]. Fortunately many reactive control designs contain redundancies. For example, schema-based designs [Arbib, 1992] have built-in redundancy that allows multiple configurations with different modes. For schema-based reactive agents, we view failure recovery as redesign task. That is, instead of finding ways of somehow escaping a local minima, by introducing noise for example, we view failure as an opportunity to reason about the current modes of perceptual and motor schemas, and to configure a set of modes that is immune to the world conditions that led the current mode to be stuck in a local minima.

We describe a hybrid agent architecture called Reflecs in which a deliberative reasoner detects and recovers from failures due to the presence of local minima within the context of autonomous reactive agents. The deliberative reasoner in Reflecs uses a model-based method derived from a theory of self-redesign in adaptive agents described in [Stroulia and Goel, 1994; 1996]. The general theory is instantiated in a "shell" called Autognostic. Autognostic provides both a language for representing information-processing in an autonomous agent as a SBF model, and a library of model-based methods for monitoring the information-processing, analyzing behaviors and detecting failures, assigning blame, and adapting the information-processing architecture. In Reflecs, the SBF model of a reactive control system enables monitoring of its processing, detection and analysis of its failures, and, under certain conditions, reconfiguration of its architecture through mode switching. In this paper, we describe the Reflecs architecture and its model-based method for reconfiguring schema-based reactive control systems. We also report on experiments with a simulated robot navigating a complex navigation space. The main result is a model-based technique that enhances the autonomy of schema-based reactive agents.

2 Schema-Based Reactive Control

AuRA [Arkin, 1990] exemplifies schema-based reactive control for mobile robots. Its reactive control system consists of perceptual and motor schemas, and mappings from the former to the latter. Each perceptual schema is responsible for one sensory modality, and each motor schema is responsible for one type of action. The perceptual schemas directly feed into the motor schemas. Each motor schema outputs a vector in response to feeds from specific perceptual schemas. The direction and magnitude of the vector are determined by the nature and gain of the schema. Figure 1 illustrates the relations of the two basic motor schemas in AuRA: move-to-goal and avoid-obstacle. The move-to-goal schema pulls the robot towards the goal and the avoid-obstacle schema pushes the robot away from an obstacle. The force vectors generated by the motor schemas are summed and normalized to decide the direction and the speed of the robot. The normalized vector is given to the robot for execution.

Schema-based reactive control architectures typically have built-in redundancy. For example, the avoid-

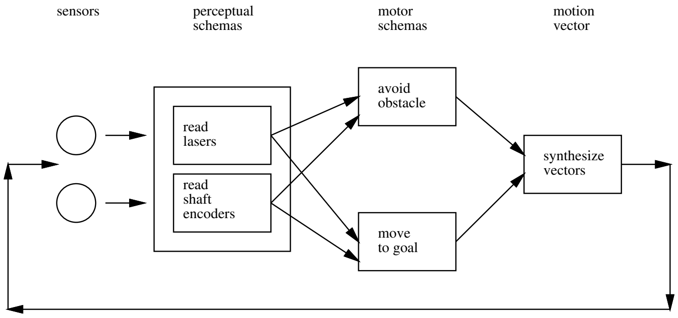

Figure 1: The perception-action cycle

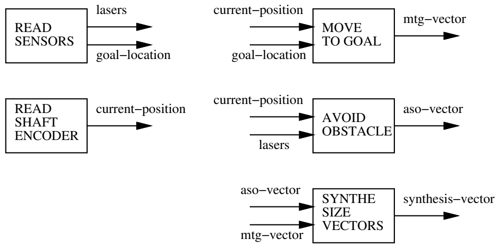

Figure 2: Functional specification of reactive schemas

obstacle schema in Figure 1 really stands for a family of motor schemas, which includes avoid-static-obstacle and avoid-barcode-obstacle as two core members. The latter schema is useful in visually-engineered navigation worlds in which specific objects are marked by barcodes readable by a rotating laser sensor. The reactive controller operates in a specific configuration characterized by the modes of the various schemas: any schema can be either in an on mode or an off mode.

2.1 Local Minima and Behavioral Cycles

Local minima present a major problem for reactive control including schema-based designs. Imagine that the goal and the obstacle in a navigation space create a potential field around them. In reactive control, the robot slides down the potential hill to reach the minimum (the goal) and move away from the potential maximum (the obstacles). Now consider a navigation space in which an obstacle is located very close to the goal. In this case, the attractive force of the goal may be cancelled by the repulsive force of the obstacle. This results in behavioral cycles in which the agent finds itself at the same spatial location at different time instances, without accomplishment of any goal in intervening time period.

The following scenario based on the 1993 AAAI Robot Competition illustrates the problem. The robot task is to navigate a world to reach a box containing some pucks, collect the pucks, carry them to a home location, and deposit them. The navigation world contains many static obstacles in addition to the box containing the pucks. It is also visually engineered by placing laser readable barcodes on all obstacles. The robot's schema-based reactive controller is configured to use the move-to-goal and avoid-static-obstacle schemas. Although the avoid-barcode-obstacle is available in the control architecture, the avoid-static-obstacle is used because it provides more reliable detection of static obstacles. This configuration works well until the robot comes close to the box containing the pucks. At this stage, the robot views the box as another obstacle (because the box is not marked by any barcode), and starts to back off. Caught between the-box-as-a-goal and the-box-as-an-obstacle, the robot's behavior becomes cyclic. Eventually it gets stuck at a location close to the box, but never reaches it, and, thus, fails in its mission. In the following, we use this realistic scenario as a running example.

3 Combining Reaction and Deliberation

To address the problem of local minima and behavioral cycles in reactive control, Reflecs introduces a deliberative reasoner in the agent architecture. The reactive controller works autonomously as outlined above. But after each perception-action cycle, it communicates state information to the deliberative reasoner. This includes information about the current position about the robot, the inputs to the reactive controller in the form of sensor readings, the behavior outputs of the the individual schemas, and the synthesized output of the reactive controller as a whole. The state information is used by the deliberative reasoner to continuously monitor the behavior and detect failures. If no failure is detected, then it does nothing. But if a failure is detected, then it analyzes the failure, and determines a reconfiguration for the reactive controller. It then sends the configuration information to the reactive controller. The information identifies specific schemas and their desired modes (on/off).

4 A Functional Model

The deliberative reasoner uses a Structure-Behavior-Function (SBF) model of the reactive control system. The SBF model specifies three kinds of knowledge. First, it specifies the functions and modes of each perceptual and motor schema in the reactive system. A schema function is specified by the types of information it takes as input and the types of information as output. Figure 2 illustrates the specification of the schemas in Figure 1. When the model-based reasoner detects a failure, knowledge about the schema functions and modes enables identification about the schemas and modes responsible for the behavior.

Second, the SBF model specifies the task structure of reactive processing in terms of tasks (Function in SBF), methods (Behaviors in SBF), and the leaf tasks which are performed by the hardware-encoded perceptual and motor schemas (Structure in SBF). The task structure explains the leaf tasks performed by the perceptual and motor schemas get composed into the robot's task. Figure 3 illustrates the SBF representation of the methods that compose the tasks of the schemas in Figure 1 into the robot task in our running example. This organi-

```
name:         box-cycle-behavior
applied to: step-in-box-cycle
subtasks:   (read-sensors step-in-execute-cycle
             read-shaft-encoder)
control:    (``serial-op'' read-sensors
             step-in-execute-cycle
             read-shaft-encoder)

name:         cycle-behavior
applied to: step-in-execute-cycle
subtasks:   (move-to-goal avoid-static-obstacle
             synthesize)
control:    (``serial-op'' (``parallel-op''
             move-to-goal avoid-static-obstacle)
             synthesize)
```

Figure 3: Method specification in the SBF language

zation of the SBF model enables localization of model-based reasoning.

Third, the SBF model specifies redundancies in the design of the reactive system through prototype tasks, which can have more than one specific instance. Each instance of a prototype task corresponds to a redundant element in the system design. This knowledge enables mode switching.

Figure 4 illustrates the SBF model of the reactive processing for the task in our running example. The main task is get-to-box, which is performed by the get-to-box-method. The get-to-box-method spawns the box-cycle task, which is performed iteratively until the goal is reached. box-cycle is a prototype task. The reactive system uses a specific instance step-in-box-cycle of the box-cycle prototype. step-in-box-cycle is performed by the box-cycle-behavior method. This method decomposes the task into read-sensors, step-in-execute-cycle, and read-shaft-encoder tasks. The step-in-execute-cycle task is performed by the cycle-behavior method, which decomposes it into move-to-goal, avoid-static-obstacle, and synthesize sub-tasks. avoid-static-obstacle is a specific instance of the prototype task avoid-obstacle. avoid-barcode-obstacle is another instance of the avoid-obstacle prototype task available in the reactive architecture.

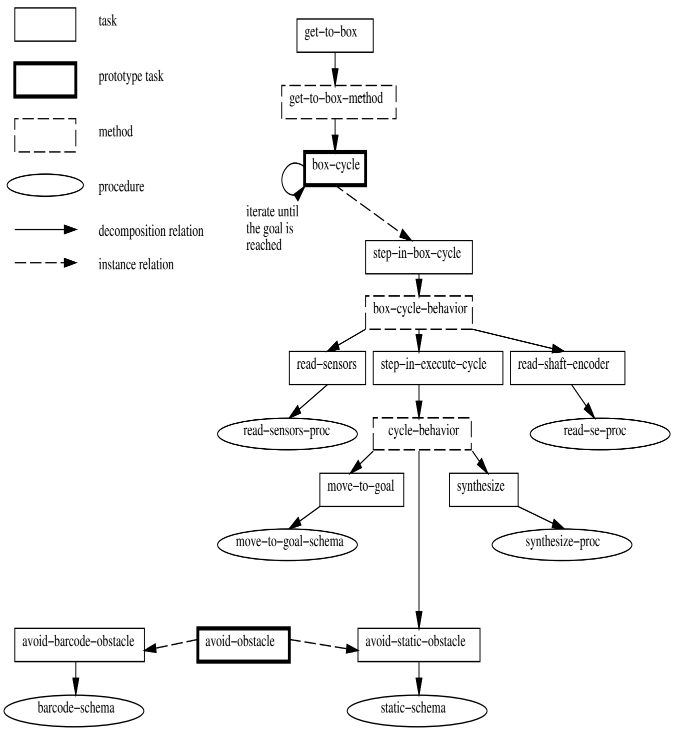

Figure 4: Task structure of reactive processing

5 Model-Based Reconfiguration

Following Autognostic, Reflecs contains knowledge of types of failure (e.g., behavioral cycles due to local minima), a library of model-based failure analysis methods, and a library of redesign methods (e.g., replacing one instance of a prototype task in a task structure with another instance). Below we describe only those methods that are relevant to the problem in our running example.

5.1 Monitoring and Failure Detection

Figure 5 illustrates the model-based method for monitoring reactive processing and detecting failures. The method keeps a trace of the reactive processing and

MONITOR$(task_k, (i, v(i))*)$
Input:
the task to be accomplished, $task_k$, and
the information context, i.e. a set of pairs, $(i, v(i))$, where i is an information type, and $v(i)$ is its value in the current context
Output:
a trace of the tasks performed, the methods applicable, and the method chosen, the updated information context, with the information produced while problem solving, and potentially, a failure description and modification suggestion.

(1) $info\_context = (i, v(i))*$
(2) $trace = 0$
(3) $\forall i : (i, v(i)) \in info\_context$ assimilate-value$(v(i)i)$
(4) **If** $task_k$ is a subtask of itself in trace
$\vee$ there is an exact same sibling subtask in a loop
then EXIT(failure-symptom:=recurrent-task$(task_k, trace, info_context)$,
possible-modification:=recurrent-task-mod)

Figure 5: Method for recurrent-task detection

ANALYZE(*failure-symptom*, *possible-modification*)
Input:
failure-symptom, including the trace of the failed problem-solving episode, the information context of the episode, and the failed task.
suggested modification method.
Output:
modification method, task to be modified

If *possible-modification* = recurrent-task-mod
then EXIT(modification:=recurrent-task-mod, *task*(*failure-symptom*))

Figure 6: Method for recurrent-task analysis

uses this trace to detect behavioral cycles, where the trace is expressed in the SBF language. In particular, the method compares the inputs to each invocation of each task with its earlier invocations. If a task is invoked repeatedly with exactly the same information as input, then the method determines that a failure of type recurrent-task has occurred. In our running example, while executing the box-cycle task, reactive processing repeats the step-in-box-cycle until the goal is reached. The model-based method for failure detection compares the inputs to the box-cycle task in the current invocation of the task with the inputs to earlier invocations of the task available in the trace. When the same inputs appear in the trace, the recurrent-task failure is flagged. In addition, recurrent-task-mod is suggested as a possible method for modifying the task structure of reactive processing.

5.2 Causal Analysis

Figure 6 illustrates Reflecs' method for analyzing the recurrent-task failure. Since the step-in-box-cycle is not preceded by another task in the reactive processing (there is no subtask of get-box scheduled before it - Figure 4), the method simply confirms the recurrent-task-mod as the modification method. If the step-in-box-cycle has been preceded by other tasks in the reactive processing, then the causal analysis method would have inspected them for potential modification and may have suggested additional modification methods ([Stroulia and Goel, 1994; 1996] provide details).

5.3 Redesign

The redesign process invokes the modification method of recurrent-task-mod on the recurrent-task of step-in-box-cycle. Figure 7 illustrates the recurrent-task-mod method. The method instructs attempts to find changeable tasks under the recurrent-task of step-in-box-cycle. A changeable task refers to a task instance that can be replaced by another instance. The SBF model of reactive processing specifies that the task structure (Figure 4) contains only one changeable task instance, avoid-static-obstacle. Thus it is replaced by the alternative task instance avoid-barcode-obstacle. This information is communicated to the reactive controller in the form of

REDESIGN(*modification*)
Input:
modification method, task to be modified
Output:
modified task structure

If *method*(*modification*) = recurrent-task-mod
then *tasks* = *changeable-tasks*(*task*(*modification*))
If tasks
then *try-next-combination*(*tasks*)

Figure 7: Method for recurrent-task modification

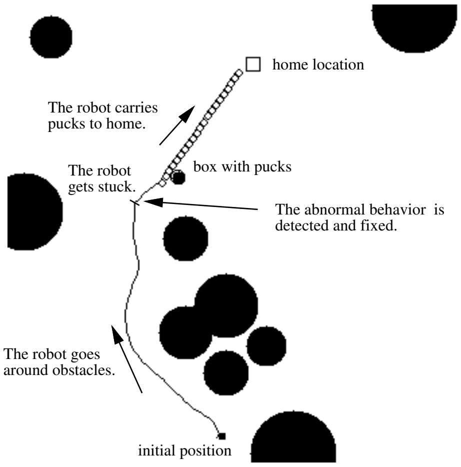

Figure 8: Robot simulation

switching the mode of avoid-static-obstacle to off and the mode of avoid-barcode-obstacle to on. This leads to a reconfigured reactive architecture that is immune to the conditions that led the earlier architecture to get stuck in a local minima.

6 Evaluation

We have extensively evaluated Reflecs on a simulated robot for running example based on the 1993 AAAI Robot Competition. The reactive controller in Reflecs is based on AuRA and is written in C. The model-based reasoner is based on Autognostic and is written in Lisp. The two communicate through a shared data structure.

The simulation program generates random navigation worlds containing goals and obstacles. Each navigation world contains twenty obstacles. The x and y coordinates of the obstacles are uniformly distributed random numbers between 0 and 64. The radius of the obstacles too are uniformly distributed random numbers between 1 and 5. The start and goal locations in each world are well-defined - an obstacle is deleted if it overlaps with the start or goal locations. Figure 8 illustrates the robot task in progress in such a world.

We conducted experiments with more than twenty such randomly generated simulated navigation worlds. Since the reactive controller views the goal itself as an ob-

Table 1: Time and steps taken for the experiments

Exp. Number	Steps Taken	Time with Deliberative Monitoring	Time with Reactive Monitoring
1	93	8	366
2	71	5	231
3	84	6	318
4	65	6	206
5	60	5	184

stacle, each of these navigation worlds presented a local minima as a result of which the robot's behavior became cyclic. In each case, Reflecs' model-based method detected the spatial-temporal cycle in the robot's behavior after no more than two repetitions. And in each case, the model-based reasoner successfully identified the modes in the reactive control architecture responsible for the robot's failure, and, successfully reconfigured the modes in the reactive control architecture.

6.1 Reactive versus Deliberative Monitoring

The Reflecs architecture raises the issue of trade-offs between deliberative and reactive monitoring. In the former mode, monitoring and failure detection is done in the deliberative reasoner, and the monitoring process and the reactive controller run in series. In the latter, the monitoring process is in the reactive controller itself, with the monitoring and reactive processes running in parallel. The model-based deliberative reasoner is invoked only when a failure is detected. In either case, Reflecs' uses the same model-based methods for reconfiguration. We conducted five additional experiments to compare the efficiency of reactive and deliberative monitoring enabled by Reflecs' model-based method. Table 1 summarizes the results. Steps in the table refers to the total number perception-action cycles in completing the robot task; time refers to the time taken to complete the task (and is registered using `time()` function in the simulation program).

As expected, reactive monitoring leads to much better performance. Deliberative monitoring may be useful in the reactive system design, development and debugging phases.

6.2 Limitations

Local minima are a formidable problem for many AI methods including reactive control methods. We know of no general solution to the problem, and expect all candidate solutions to be restricted to specific situations. Reflecs' current method for reconfiguration can become costly if several schemas in the reactive architecture are potential candidates for mode switching. One solution might be to complement the model-based method with other methods. We have augmented the model-based method with a memory of past failures and corresponding reconfigurations. Also, Reflec's current method is limited in the kinds of modifications it can make to the schema-based reactive controller. At present, the method can only reconfigure the reactive architecture by mode switching which results in the replacement of one motor schema for realizing an action in a class of actions by another motor schema for realizing an action in the same action class. The model-based method cannot modify the mappings between the perceptual and the motor schemas.

7 Related Research

Recovery from failures has been an important topic in AI research on planning and design. Here we will compare our work only with related research on reconfiguration of reactive control architectures, and, in particular, with model-based approaches to reactive architectural reconfiguration. In the context of reactive agents, Howe and Cohen [Howe and Cohen, 1991] describe a method for recovery from failure. Their method not only monitors agent performance and detects failures, but also keeps statistics of failures. It uses heuristic knowledge to recover from recurrent failures and to modify the reactive controller. The domain-specific heuristics directly map specific kinds of recurrent failures to specific kinds of modifications to the reactive controller. In contrast, Reflecs maintains a model of the reactive controller, and, when a failure occurs, it uses the model to analyze the failure and recover from it by reconfiguring the reactive architecture.

Williams and Nayak [Williams and Nayak, 1996] describe a different model-based technique for reconfiguring reactive systems and its implementation in a system called Livingstone. The goal of their work is the same as ours: model-based agent autonomy. But our work differs from theirs in the dimensions of domain, task, knowledge, and method. Livingstone works in the domain of "immobile robots." In contrast, Reflecs operates in the domain of reactive controllers on mobile robots. In this domain, local minima present a formidable problem. Livingstone performs a form of state-based diagnosis, while Reflecs performs function-based analysis. In Livingstone, the states pertain to the physical system, but in Reflecs, the system states are "information states" of the reactive controller. In addition, while Livingstone's model of the physical system is "flat," Reflecs' knowledge of the information system is organized hierarchically in a SBF model. Livingstone uses the method of conflict-directed best-first search to identify components whose modes need to switched `on/off`. Reflecs uses function-directed top-down search for mode identification and system reconfiguration. The search is localized by the organization of the SBF model.

As mentioned earlier, Reflecs' model-based method for reactive control reconfiguration derives from Autognostic's general theory of self-redesign in adaptive systems. Reactive controllers aboard mobile robots are hardware-embedded software systems. Autognostic's theory of modeling hardware-embedded software systems arises from an earlier theory of model-based failure analysis and redesign of hardware systems [Goel and Stroulia, 1996].

8 Conclusions

Local minima present a major problem for methods of reactive control. Since schema-based reactive systems contain built-in redundancy that enables multiple configurations with different modes, Reflecs casts recovery from failure due to local minima as a redesign problem. Reflecs uses a model-based method for monitoring reactive processing, detecting and analyzing failures, and reconfiguring the reactive architecture. The method works by switching the schema modes, provided that the number of participating schemas is small. The Reflecs architecture enables both deliberative and reactive monitoring.

Reflec's model-based method suggests that four kinds of qualitative knowledge about reactive systems enable architectural reconfiguration through mode switching. First, when a reactive agent gets caught in a local minima, its behavior initially becomes cyclic. This knowledge enables detection of failures due to local minima. Second, Reflec's model of the reactive architecture specifies the functions and modes of the perceptual and motor schemas. This knowledge enables mode identification. Third, the model represents the task structure of reactive processing in the structure-behavior-function (SBF) language. This knowledge localizes the reasoning. Fourth, in each perception-action cycle, the deliberative reasoner uses knowledge of the trace of reactive processing that leads to a specific behavior. Together with the SBF model, this knowledge enables continuous monitoring, failure detection and analysis.

References

[Arbib, 1992] M. Arbib. Schema theory. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1427–1443. Wiley, 2nd edition, 1992.

[Arkin, 1989] Ronald Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112, August 1989.

[Arkin, 1990] Ronald Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[Goel and Stroulia, 1996] Ashok Goel and Eleni Stroulia. Functional device models and model-based diagnosis in adaptive design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 10:355–370, 1996.

[Howe and Cohen, 1991] Adele E. Howe and Paul R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 801–808, 1991.

[Stroulia and Goel, 1996] Eleni Stroulia and Ashok K. Goel. A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. In *Proc. Thirteenth National Conference on Artificial Intelligence*, pages 959–964, 1996.

[Stroulia and Goel, 1994] Eleni Stroulia and Ashok K. Goel. Learning problem-solving concepts by reflecting on problem solving. In *Proc. 1994 European Conference on Machine Learning*, pages 287–306, 1994.

[Williams and Nayak, 1996] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proc. Thirteenth National Conference on Artificial Intelligence*, pages 971–978, 1996.