Architectures and Idioms: Making Progress in Agent Design

Joanna Bryson and Lynn Andrea Stein[1]

Artificial Intelligence Laboratory, MIT
545 Technology Square, Cambridge MA 02139,USA
joanna@ai.mit.edu and las@ai.mit.edu

Abstract. This chapter addresses the problem of producing and maintaining progress in agent design. New architectures often hold important insights into the problems of designing intelligence. Unfortunately, these ideas can be difficult to harness, because on established projects switching between architectures and languages carries high cost. We propose a solution whereby the research community takes responsibility for re-expressing innovations as idioms or extensions of one or more standard architectures. We describe the process and provide an example — the concept of a Basic Reactive Plan. This idiom occurs in several influential agent architectures, yet in others is difficult to express. We also discuss our proposal's relation to the the roles of architectures, methodologies and toolkits in the design of agents.

1 Introduction

Design is the central problem of developing artificial intelligence. Even systems that learn or reason, learn or reason better when enabled by good design. Both experience and formal arguments have shown that structure and bias are necessary to these processes, in order to reduce the search space of their learning or reasoning algorithms to a tractable size [8, 29, 34–36]. Agent technology is itself a major design innovation which harnesses the innate ability of human designers to reason about societies of actors [19].

The importance of design explains the focus of our community on agent architectures. Agent architectures are essentially design methodologies: they are technological frameworks and scaffolding for developing agents. Of course, viewed as methodology, no architecture can be called complete. A complete architecture could automatically generate an intelligent agent from a specification.

Because the development of production-quality agents to date has always required the employment of human designers, there is a high cost associated with switching architectures. In fact, there is a high cost even for making changes to an architecture. The engineers responsible for building systems in an upgraded architecture require time to learn new structures and paradigms, and their libraries of existing solutions must be ported to or rewritten under the new version. These problems alone deter the adoption of new architectures. They are further exacerbated by the cost, for the architect, of

[1] LAS: also Computers and Cognition Group, Franklin W. Olin College of Engineering, 1735 Great Plain Avenue, Needham, MA 02492 las@olin.edu

creating documentation and maintaining a production-level architecture, and for the project manager, of evaluating new architectures. Nevertheless, new architectures often hold important insights into the problems of designing intelligence.

In this chapter we propose a meta-methodological strategy for the problem of incorporating the advances of new architectures into established development efforts. Our proposal is simple: a researcher, after having developed a new architecture, should express its major contributions in terms of one or more of the current "standard" architectures. The result of this process is a set of differences that can be rapidly understood by and absorbed into established user communities.

The next section presents this meta-methodological contribution in more detail. The bulk of the chapter then consists of an extended example of one such contribution, the Basic Reactive Plan (BRP), drawn not only from our own work [5, 6] but also from other architectures [10, 12, 26], We describe implementing it in three different architectures — Ymir [30], PRS-CL [23] and JAM [18], a Java-based extension of UM-PRS; we also discuss such an implementation in Soar. We conclude with a discussion of the roles of architecture, methodology, and toolkit in the problem of intelligent agent design.

2 Architecture Analysis by Reduction to Idiom

Consider the problem of expressing a feature of one architecture in another. There are two possible outcomes. A feature f_1, of architecture A_1 may be completely expressible in A_2. Assuming that this expression is not trivial (e.g. one line of code) then A_1 *constrains* A_2 in some way. On the other hand, if f_1 cannot be expressed in A_2 without altering the latter architecture, then A_1 *extends* A_2. These conditions are not mutually exclusive — two architectures generally both constrain and extend each other, often in multiple ways. Identifying these points of difference allows one architecture to be described in terms of another.

When we speak of the relative expressive power of two architectures, we are not really comparing their linguistic expressibility in the classical sense. Almost all agent architectures are Turing-complete; that is, a universal computing machine can be constructed within almost any agent architecture. This universal computing machine can then be used as an implementation substrate for another agent architecture. So, in the formal sense, all agent architectures are inter-reducible. We are concerned instead with the kinds of computational idioms that are *efficaciously expressible*[2] in a particular architecture. In this sense, an architecture A_1 may be considered to extend A_2 when there is no way to express reasonably succinctly the attributes of A_1 in A_2.

If, on the other hand, a feature f_1 of A_1 can be translated into a coding i^{f_1} of A_2 with reasonable efficiency, then we call that coding i^{f_1} an *idiom*. As we explained above, the existence of such an idiom means A_1 constrains A_2. This notion of constraint may

[2] In computational complexity theory, the notion of reducibility is augmented with the asymptotic worst case complexity of the reduction. So, for example, in the theory of NP-completeness, polynomial-time reducibility plays a crucial role. Our notion of efficacious expressibility does not rely on any criterion so sharply defined as the computational complexity of the reduction computation, but is intended to evoke a similar spectrum of reduction complexity.

seem counterintuitive, because new features of an architecture are usually thought of as extensions. However, as we argued in the introduction, extending the capabilities of the developer often means reducing the expressibility of the architecture in order to biases the search for the correct solution to the problem of designing an agent.

Although in our example A_1 is constrained relative to A_2 due to feature f_1 of A_1, adding the idiom i^{f_1} is unlikely to constrain A_2. A_2 retains its full expressive power so long as the use of i^{f_1} is not mandatory. For an example, consider object oriented programming. In a strictly object-based language such as smalltalk, OOP is a considerable constraint, which can consequently lead to effective and elegant program design. In contrast, C++ has added the features of objects, but still allows the full expression of C. Thus, for the C++ programmer, the elegance of OOP is an option, not a requirement.

An idiom is a compact, regularized way of expressing a frequently useful set of ideas or functionality. We borrow the notion of idiom both from natural language and computer science, though in computer science, the term ‘idiom’ (or ‘design pattern’) is sometimes used for a less rigorous mapping than we mean to imply. An architecture can be expressed as a set of idioms, either on programming languages or sometimes on other architectures. Researchers seeking to demonstrate that their architecture makes a contribution to agent design might do well to express their architecture in terms of idioms in familiar architectures. In this way, the architecture can be both readily understood and examined. We demonstrate this approach in the following two sections.

It is important to observe that this meta-methodology is different from though related to the practice of publishing extensions of architectures. First, we do not discourage the practice of building entirely new architectures. If an architecture has been built as an entity, it is more likely to have significant variation from standard architectures, potentially including vastly different emphases and specializations for particular tasks. These specializations may turn out to be generally useful contributions, or to be critical to a particular set of problems. Second, the idiomatic approach emphasizes the search for generally applicable strategies. Generality here does not necessarily mean across all possible problems, but it should mean an idiomatic solution relevant across a number of different underlying architectures. Thus even if an idiom is developed in the context of a well known architecture, it would be useful if, on publication, the researcher describes it in terms of general applicability.

3 Discovering an Idiom: the Basic Reactive Plan

We now illustrate our suggestions with an idiom that we will call a *Basic Reactive Plan* (BRP). In this section we will introduce the idiom, its functionality and the reasons we consider it to be a useful idiom. In the next we demonstrate its idiomaticity by expressing it in several agent architectures. We begin by situating the BRP within reactive planning and reactive AI.

3.1 Reactive AI, Reactive Planning, and Reactive Plans

The terms ‘reactive intelligence’, ‘reactive planning’ and ‘reactive plan’ appear to be closely related, but actually signify the development of several very different ideas.

Reactive intelligence is what controls a reactive agent — one that can respond very quickly to changes in its situation. Reactive intelligence has sometimes been equated with statelessness, but that association is exaggerated. Intelligence of any kind requires state: for learning, perception, and complex control [4]. Reactive intelligence is however associated with minimal representations and the lack of deliberation.

Reactive planning is something of an oxymoron. It describes the way reactive systems handle the problem traditionally addressed by conventional planning — that is, action selection. Action selection is the ongoing problem for an autonomous agent of deciding what to do next. Conventional planning assumes the segmentation of intelligent behavior into the achievement of discrete goals. A conventional planner constructs a sequence of steps guaranteed to move an agent from its present state to a goal state. Reactive planning, in contrast, chooses only the immediate next action, and bases this choice on the current context. In most architectures utilizing this technique, reactive planning is facilitated by the presence of *reactive plans.* Reactive plans are stored structures which, given the current context, determine the next act.

3.2 Basic Reactive Plans

The BRP is an idiom relating to the form of the structure for a reactive plan. The simplest reactive plan possible is a simple sequence of primitive actions $\iota_1, \iota_2, \dots \iota_n$. Executing a sequential plan involves priming or activating the sequence, then releasing for execution the first primitive act ι_1. The completion of any ι_i releases the following ι_{i+1} until no active elements remain. Notice that this is *not* equivalent to the process of *chaining*, where each element is essentially an independent production, with a precondition set to the firing of the prior element. A sequence is an additional piece of control state; its elements may also occur in different orders in other sequences. Depending on implementation, the fact that sequence elements are released by the *termination* of prior elements can be significant in real time environments, and the fact that they are actively repressed by the existence of their prior element can increase plan robustness (see [39] for further discussion on time). This definition of sequence is derived from biological models of serial ordering (e.g. [14]).

A BRP is a more complicated plan structure for circumstances when the exact ordering of steps cannot be predetermined. For example, a BRP is useful in cases where some requirements of a goal might have already been met, might become unset during the procedure, or might require arbitrary repetitions of steps or sets of steps.

A *BRP step* is a triple $\langle \pi, \rho, \alpha \rangle$, where π is a priority, ρ is a releaser, and α is an action. A *BRP* is a small set (typically 3–7) of plan steps $\{\langle \pi_i, \rho_i, \alpha_i \rangle *\}$ associated with achieving a particular goal condition. The releaser ρ_i is a conjunction of boolean perceptual primitives which determine whether the step can execute. Each priority π_i is drawn from a total order. Each action α_i may be either another BRP or a sequence as described above. Hierarchies are controversial in reactive control because they generally involve maintaining a stack and can consequently become a control bottleneck. In this chapter we discuss hierarchical BRPs only briefly (see Section 4.1).

The order of expression of plan steps is determined by two means: the releaser and the priority. If more than one step is currently operable, then the priority determines which step's α is executed. If more than one step is released with the same priority, then the winner is determined arbitrarily. Normally the releasers ρ_i on steps with the same priority are mutually exclusive. If no step can fire, then the BRP terminates. The top priority step of a BRP is often, though not necessarily a goal condition. In that case, its releaser, ρ_1, recognizes that the BRP has succeeded, and its action, α_1 terminates the BRP.

3.3 An Example BRP

The details of the operation of a BRP are best explained through an example. BRPs occur in a number of architectures, and have been used to control such complex systems as flight simulators and mobile robots [1, 6, 25]. However, for clarity we draw this example from blocks world. Assume that the world consists of stacks of colored blocks, and that we want to enable an agent to meet the goal of holding a blue block[3]. A possible plan would be:

Priority	Releaser	Action
4	(holding block) (block blue)	*goal*
3	(holding block)	drop-held-object, lose-fixation
2	(fixated-on blue)	grasp-top-of-stack
1	(blue-in-scene)	fixate-blue

For this plan, we will assume that priority is strictly ordered and represented by position, with the highest priority step at the top. We refer to the steps by their priority.

Consider the case where the world consists of a stack with a red block sitting on the blue block. If the agent has not already fixated on the blue block before this plan is activated (and it is not holding anything), then the first operation to be performed would be element **1** because it is the only one whose releaser is satisfied. If, as a part of some previous plan, the agent has already fixated on blue, **1** would be skipped because the higher priority step **2** has its releaser satisfied. Once a fixation is established, element **2** will trigger. If the grasp is successful, this will be followed by element **3**, otherwise **2** will be repeated. Assuming that the red block is eventually grasped and discarded, the next successful operation of element **2** will result in the blue block being held, at which point element **4** should recognize that the goal has been achieved, and terminate the plan.

This single reactive plan can generate a large number of expressed sequential plans. In the context of a red block on a blue block, we might expect the plan 1–2–3–1–2–4 to execute. But if the agent is already fixated on blue and fails to grasp the red block successfully on first attempt, the expressed plan would look like 2–1–2–3–1–2–4. If the unsuccessful grasp knocked the red block off the blue, the expressed plan might be 2–1–2–4. A basic reactive plan is both robust and opportunistic.

If a grasp fails repeatedly, the above construction might lead to an infinite number of retries. This can be prevented through several means: either through habituation at the step level, timeouts at the level of the BRP, or through a separate attentional mechanism which is triggered by repeated attempts or absence of change. This last mechanism

[3] This problem is from [33]. The perceptual operations in this plan are based on the visual routine theory of [32], as implemented by [17].

requires another part of the architecture to provide for controlling attention; as is discussed immediately below, this is a common feature of reactive architectures.

3.4 Identifying a Valuable Idiom

How does one find a useful architectural feature, and how does one distinguish whether it is worth expressing as an idiom? Features are distinguished by the methodology described in Section 2, by comparison and reduction of an architecture to one or more others. Features can be idioms if they can be expressed in other architectures. Idioms are valuable within an architecture if they perform useful functions; they are valuable to the community if they are not yet regular features of existing architectures or methodologies. In this section, we illustrate the process of identification by going through the history of the identification of the BRP in our own research. We also give two counter-examples from the same source.

We first developed a BRP as one of three elements of the reactive planning system for the architecture Edmund [5, 6]. Edmund was initially conceived in the early nineties in response to the Subsumption Architecture [3]. Subsumption is a strictly behavior-based architecture with minimal internal state. Because the individual behaviors are intended to act primarily in response to their environment, only very limited communication is allowed between them. Developing action selection within this framework is very complicated. In Edmund, we attempt to maintain the advantages of the behavior-based system while simplifying the design of behavior arbitration. This is done with reactive plans.

The three structural elements of Edmund's reactive plans are action patterns, competences and drive collections. Action patterns are the simple sequences described in Section 3.2. Competences are BRPs. Drive collections are special forms of the BRP with extra state. In Edmund, a drive collection serves as the root of a BRP hierarchy, providing both motive force and an "alarm" system [28] for switching attention to urgent needs or salient events. To illustrate the search for valuable idioms, we consider a reduction of each of these three features of Edmund's reactive plans in the context of Subsumption Architecture (SA) [3], the Agent Network Architecture (ANA) [21], the Procedural Reasoning System (PRS) [13] and Soar [24].

Although deceptively simple, action patterns actually required extensions to the original versions of each of the above architectures except PRS. Within SA, a sequence can only be expressed within a single behavior, as part of its FSM. When the need for behavior sequencing was discovered, a mechanism for suppressing all but the currently active behavior was developed [9]. ANA explicitly represents the links of plans through chains of pre- and post-conditions, but with no privileged activation of a particular plan's elements. This sequencing strategy is inadequate [31], and has been improved in more recent derivative architectures [2, 27]. Soar initially represented sequences only as production chains. As mentioned in Section 3.2, this mechanism is insufficient in real-time applications. This problem has now been addressed with a dedicated sequencing mechanism that monitors durations [20]. PRS has a reactive plan structure, the Act, which allows for the coding not only of sequences, but of partial plans. Although an action pattern could therefore be seen as an idiom on an Act, we have no strong reason to argue that this particular reduction in power is useful. In conclusion, we have evidence from the history of multiple architectures that an action pattern is an important feature. However, it is not one that can easily be implemented as an idiom, because it generally extends rather than constrains architectures that lack a trivial way to express it.

As for a parallel mechanism for allowing attention shifts, some implementation of this feature is ubiquitous in reactive architectures. SA assumes that all behaviors operate in continuous parallel, and can always grasp attention. The difficulty in controlling this mechanism was the main motivation for Edmund. ANA has similar features and control problems: each behavior is always evaluated as a possible next act. PRS addresses both control and reactivity on each cycle: it first persists on the currently active plan, then engages meta-reasoning to check whether a different plan deserves top priority. Soar also seems to have struck a balance between persistence and reactivity. Being production based, it is naturally distributed and reactive, similarly to SA and ANA. Persistence is encouraged not only by the new seriating mechanism mentioned above, but primarily by clustering productions into *problem spaces*. A problem space is actually somewhat like a BRP in that it focuses attention on a subset of possible productions. Because all of these architectures have means for monitoring the environment and switching attention, introducing drive collections on top of these mechanisms does not have demonstrable utility.

The BRP is a different matter. First, there are several examples of structures like it in the planning literature, yet it is not present as a single feature in any of these four architectures. BRPs are similar in effect, though not in construction, to the "triangle tables" built to make STRIPS plans more robust [12]. They are also similar to teleo-reactive plans [26] and what Correia and Steiger-Garção [10] call "fixed action patterns" (a term that seems inappropriate given their flexibility in ordering expressed behavior). We therefore have evidence not only from our own experience, but also from several other architecture research communities of the utility of the BRP. Nevertheless, the BRP, unlike the simple sequence, is not in general use. In fact, to implement them in SA or ANA would require extensions, for much the same reason as the implementation of sequences requires extensions. There is no intrinsic way to favor or order a set of expressed actions in either architecture except by manipulating the environment. PRS and Soar, on the other hand, contain sufficient ordering mechanisms that implementing a BRP idiom should be tractable.

In summary, the value of an idiom is dependent on two things. It must be expressible but not trivially present in some interesting set of architectures, and it must be useful. Utility may be indicated by one's own experience, but also by the existence of similar features in other architectures. With respect to the BRP, it is present in several architectures in the literature, and we have independently found its programming advantages sufficient to lead us to implement it in several architectures besides our own. The next section documents these efforts.

4 Expressing BRPs in Other Architectures

In the previous section, we introduced an architectural feature we called the Basic Reactive Plan. In this section, we document the implementation of this feature as an idiom on a number of architectures. In Section 5 we will discuss how to best exploit this sort of advance in agent design.

4.1 Ymir

Our first effort to generalize the benefits of Edmund's action selection was not in a widely-used standard architecture, but was rather in another relatively recent one, Ymir ([30] see also [7]). Ymir is designed to build complex agents capable of engaging in multi-modal dialog. A typical Ymir agent can both hear a human conversant and observe their gestures. The agent both speaks and provides non-verbal feedback via an animated character interface with a large number of degrees of freedom. Ymir is a reactive and behavior-based architecture. Its technical emphasis is on supporting interpretation of and responses to the human conversant on a number of different levels of time and abstraction. These levels are the following:

- a "reactive layer", for process-related back-channel feedback and low-level functional analysis. To be effective, this layer must be able operate within 100 millisecond constraints,
- a "process control layer", which deals with the reconstruction of dialogue structure and monitoring of process-related behaviors by the user, and
- a "content layer", for choosing, recognizing, and determining the success of content level dialogue goals.

Ymir also contains a key feature, the *action scheduler*, that autonomously determines the exact expression of behaviors chosen by the various layers. This serves to reduce the cognitive load, accelerate the response rate, and ensure that expressed behavior is smooth and coherent.

Although Ymir excels at handling the complexity of multimodality and human conversations, it does not have a built in capacity for motivation or long-term planning. Ymir is purely reactive, forming sentences for turn taking when prompted by a human user.

Because of Ymir's action scheduler, the implementation of drives, action patterns and BRPs was significantly different from that in Edmund. The scheduler could be relied on to "clean up" behaviors that had been triggered but were not expressed after a timeout, but it could also be signaled to allow their lifetimes to be renewed. In the case of action patterns, all of the elements were posted to the schedule, each with a unique tag, and all but the first with a precondition requiring that its predecessor complete before it began operating.

The BRP is implemented as an Ymir behavior object which is posted to the action-scheduler. When executed, the BRP selects a step (as per Section 3.2) and adds the step to the scheduler. The BRP then adds itself to the scheduler with the termination of its child as a precondition. The original copy of the BRP then terminates and is cleaned up by the scheduler. If the child or its descendents maintain control for any length of time, the 'new' parent BRP will also be cleaned up (see further [5] on reactive hierarchies). Otherwise, the BRP persists in selecting plan elements until it either terminates or is terminated by another decision process.

4.2 PRS-CL

Our next implementation of BRPs came during a project exploring the use of reactive planning in dialogue management. Because this was a relatively large-scale project, a well-established architecture, PRS, was chosen for the reactive planning. Because of other legacy code, the language of the project was Lisp. Consequently, we used the SRI implementation of PRS, PRS-CL [23]. PRS-CL provides not only an implementation of PRS, but also documentation and a set of GUI tools for developing and debugging PRS-CL agent systems. These tools are useful both for creating and debugging the main plan elements, the Act graphs.

Acts are roughly equivalent to action patterns described above, but significantly more powerful, allowing for parallel or alternative routes through the plan space and for cycles. We initially thought that a BRP would be best expressed within a single Act. However, there is no elegant way to express the inhibition of lower priority elements on an Act choice node. Instead, we implemented the BRP as a collection of Acts which are activated in response to the BRP's name being asserted as a goal. This results in the activation of all the Acts (steps) whose preconditions have been met.

PRS-CL has no built-in priority attribute for selecting between Acts. Selection is handled by meta-rules, which operate during the second half of the PRS control cycle (as mentioned in Section 3.4). We created a special function for the meta-rule that selects which of the Acts that have been triggered on a cycle is allowed to persist. This function is shown in Figure 1.

```
(defun BRP (list-of-ACTs)
  (let* ((comp-list (consult-db '(prs::speaker-competence prs::x.1)))
         (current-BRP (BRP-name (first comp-list)))
         (current-priorities (priorities-from-name current-BRP)))

    ; loop over priorities in order, terminate on first one available
    ; to fire (as indicated by presence in list-of-ACTs)
    (do ((priorities current-priorities (rest priorities))
         (result))
        ; this is the 'until' condition in a lisp 'do' loop ---
        ; if it is true, the 'do' returns a list containing ''result''
        ((setf result (BRP-find-ACT (first priorities) list-of-ACTs))
         (list result))
      ; if we have no priorities, we return something random
      (unless (and priorities list-of-ACTs)
        (return (set-randomly list-of-ACTs))))
                     ))
    )) ; defun BRP
```

Fig. 1. BRP prioritization implemented as a function for PRS-CL meta-reasoning. Since relative priority is situation dependent, the BRP function must query the database to determine the current competence context. Priorities are maintained as a list of Act names, each associated with a BRP name.

The BRP function we have built for PRS-CL depends on a list of priority lists, where each priority list is associated with the name of the BRP. This is somewhat unfortunate, because it creates redundant information. The Act graphs contain similar information implicitly. Any such replication often leads to bugs caused by inconsistencies in long-term maintenance. Ideally, the priority lists would be edited and maintained within the same framework as the Acts are edited and maintained, so that consistency could be checked automatically.

The fact that PRS-CL and its associated tool set emphasize the construction of very complex plan elements in the form of Acts, but provide relatively little support for the construction of metarules or the manipulation of plans as hierarchies, would seem to reflect an expectation that switching attention during plans is an unusual exception. Normal behavior is based on the execution of the elaborate Act plans. This puts PRS-CL near the opposite end of the reactive planning spectrum from architectures such as Subsumption (SA). As described in Section 3.4, SA assumes that unpredictability in action scheduling is the norm, and predictably sequenced actions are the exception. The BRP reflects a moderation between these two extremes. The BRP expects and handles the unexpected, but provides for the specification of solutions that require multiple, ordered steps.

4.3 JAM / UM-PRS

We have not been entirely happy with PRS-CL, so have been exploring other architectures for our dialogue project. JAM is a Java based extension of UM-PRS, which is in turn a C++ version of PRS that is more recently developed than PRS-CL. The control cycle in all three languages is similar. JAM and UM-PRS have somewhat simplified their analog of the Act so that it no longer allows cycles, but it is still more powerful than Edmund's action patterns. The JAM Act analog is called simply a "plan"; for clarity, we will refer to these as JAM-plans.

JAM-plans do have a notion of priority built in, which is then used by the default meta-reasoner to select between the JAM-plans that have been activated on any particular cycle. Our current implementation of BRPs in JAM is consequently a simplified version of the BRP in PRS-CL. A JAM BRP also consists primarily of a set of JAM-plans which respond to an "achieve" goal with the name of the BRP. However, in JAM, the priority of a step within the BRP is specified by hand-coding priority values into the JAM-plans. This is simpler and neater than the PRS-CL solution described above (and works more reliably). On the other hand, losing the list structure results in the loss of a single edit point for all of the priorities of a particular competence. This again creates exposure to potential software bugs if a competence needs to be rescaled and some element's priority is accidently omitted.

Both PRS implementations lack the elegance of the Ymir and Edmund solutions in that Acts or JAM-plans contain both local intelligence in their plan contents, and information about their parent's intelligence, in the priority and goal activation. In Edmund, all local information can be reused in a number of different BRPs, potentially with different relative priorities. The Ymir BRP implementation also allows for this, because the BRP (and sequence) information is present in wrapper objects, rather than in the plans themselves. We have not yet added this extra level of complexity in either PRS-CL or JAM, but such an improvement should be possible in principle.

4.4 Soar

We have not actually implemented a BRP in Soar yet, but for completeness with relation to the previous section, we will make a short description of the expected mechanism. Much as in PRS, we would expect each currently operable member element of the BRP to trigger in response to their mutual goal. This could be achieved either by preconditions, or exploiting the problem space mechanism. In Soar, if more than one procedure triggers, this results in an impasse which can be solved via meta-level reasoning. We assume it would be relatively simple to add a meta-level reasoning system that could recognize the highest priority element operable, since Soar is intended to be easily extendible to adapt various reasoning systems. This should operate correctly with or without chunking.

The Soar impasse mechanism is also already set for monitoring lack of progress in plans, a useful feature in BRPs mentioned in Section 3.3. In Edmund, retries are limited by setting "habituation" limits on the number of times a particular plan step will fire during a single episode. Ymir also supplies its own monitoring system; we have not yet addressed this problem in our PRS-CL or JAM implementations.

5 Discussion: Architecture, Methodology, or Tool?

An agent architecture has been defined as a methodology by which an agent can be constructed [37]. However, for the purpose of this discussion, we will narrow this definition to be closer to what seems to be the more common usage of the term. For this discussion, an *architecture* is a piece of software that allows the specification of an agent in an executable format. This actually moves the definition of architecture closer to the original definition of agent language, as a collection of "the right primitives for programming an intelligent agent" [38]. A *methodology* is a set of practices which is appropriate for constructing an agent. A *tool* is a GUI or other software device which creates code suitable for an architecture (as defined above), but code which may still be edited. In other words, the output of an architecture is an agent, while the output of a tool is code for an agent. A methodology has no output, but governs the use of architectures and tools.

In this chapter, we are emphasizing the use of idioms to communicate new concepts throughout the community regardless of architecture. In natural language, an idiom can be recognized as a phrase whose meaning cannot be deduced from the meanings of the individual words. If an idiom is built directly into an architecture, as a feature, there may be an analogous loss. Some features may be impossible to express in the same architecture, such as the BRP and fully autonomous behavior modules. Features implemented directly as part of an architecture reduce its flexibility. However, if a feature is implemented as an idiom, that can be overridden by direct access to the underlying code, then the problem of conflicting idioms can be dealt with at a project management level, rather than through architectural revision.

Accessibility to different idioms may explain why some architectures, such as SA or ANA, despite wide interest, have not established communities of industrial users, while others, such as Soar and PRS, have. Soar and PRS are sufficiently general to allow for the expression of a number of methodologies. However, as we said earlier, generality is not necessarily the most desirable characteristic of an agent development approach. If it were, the dominant agent "architectures" would be lisp and C. Bias towards development practices that have proven useful accelerates the development process.

We believe GUI toolkits are therefore one of the more useful ways to communicate information. They are essentially encoded methodologies: their output can be generalized to a variety of architectures (see further [11]). A toolkit might actually be an assemblage of tools chosen by a project manager. Each tool might be seen as supporting a particular idiom or related set of idioms. A GUI tool that would support the BRP would need to be able to parse files listing primitive functions, and existing sequential plans and BRPs. A new BRP could then be created by assembling these items into a prioritized list with preconditions. This assemblage can then be named, encoded and stored as a new BRP. Such a tool might also facilitate the editing of new primitive elements and preconditions in the native architecture.

Of course, not all idioms will necessarily support or require GUI interfaces. Ymir's action scheduler, discussed in Section 4.1, is a structure that might easily be a useful idiom in any number of reactive architectures if they are employed in handling a large numbers of degrees of freedom. In this case, the "tool" is likely to be a stand-alone module that serves as an API to the agent's body. Its function would be to simplify control by smoothing the output of the system, much as the cerebellum intercedes between the mammalian forebrain and the signals sent to the muscular system.

What then belongs in an architecture? We believe architectures should only contain structures of extremely general utility. Program structures which might be best expressed as architectural attributes are those where professional coding of an attribute assists in the efficiency of the produced agents. This follows the discussion of agent languages given in [22]. Examples of such general structures are the interpreter cycle in PRS or the production system and RETE algorithm in Soar. Other structures, such as the BRP, should be implemented via idioms, and tools developed to facilitate the correct generation of those idioms.

Again, we do not discourage the development of novel architectures. An architecture may be a useful level of abstraction for developing specialized ideas and applications. However, when distributing these inventions and discoveries to the wider community, tools and idioms may be a more useful device. Note that a specialist in the use of a particular tool could be employed on a number of projects in different languages or architectures with no learning overhead, provided the tool's underlying idioms have already been expressed in those languages or architectures.

6 Conclusion

In this chapter we have argued that methodology is the main currency of agent design. Novel architectures are useful platforms for developing methodology, but they are not very useful for communicating those advances to the community at large. Instead, the features of the architecture should be distilled through a process of reduction to more standard architectures. This allows for the discovery of both extensions and idioms. Idioms are particularly useful, because they allow for methodological advances to be absorbed into established communities of developers. Given that this is the aim, we consider the development of tools for efficiently composing these idioms to often be a better use of time than attempting to bring an architecture to production quality.

As an ancillary point, our discussion of reactivity in Section 4.2 above demonstrates that this process of reduction is a good way to analyze and describe differences in architectures. This process is analogous to the process of "embedding" described in [15] (see also [16]). We have elsewhere used this approach to do a rough cross-paradigm analysis of useful features for agent architectures [4]. The reductions in that article were not particularly rigorous. Doing such work with the precision of [15] might be very illuminating, particularly if the reductions were fully implemented and tested. A particularly valuable unification might be one between a BDI architecture such as UM-PRS or JAM and Soar, since these are two large communities of agent researchers with little overlapping work.

Our community's search for agent methodology is analogous to evolution's search for the genome. When we find a strategy set which is sufficiently powerful, we can expect an explosion in the complexity and utility of our agents. While we are searching, we need both a large variety of novel innovations, and powerful methods of recombination of the solutions we have already found. In this chapter, we have demonstrated a mechanism for recombining the attributes of various architectures. We have also contributed some material in the form of the Basic Reactive Plan (BRP). It is our hope that contributions such as these can help the community as a whole develop, and discover through automated development, the agents we wish to build.

Acknowledgements

Thanks to Greg Sullivan and our anonymous reviewers for many excellent suggestions on this chapter. The work on Ymir was conducted with Kris Thórisson, who also contributed to this chapter with early discussions on the role of alternative architectures. The dialogue work was conducted at The Human Communication Research Centre in the University Edinburgh Division of Informatics under Prof. Johanna Moore.

References

[1] Scott Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Stanford University, December 1996. Department of Computer Science.

[2] Bruce Mitchell Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, MIT, September 1996. Media Laboratory, Learning and Common Sense Section.

[3] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.

[4] Joanna Bryson. Cross-paradigm analysis of autonomous agent architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(2):165–190, 2000.

[5] Joanna Bryson. Hierarchy and sequence vs. full parallelism in reactive action selection architectures. In *From Animals to Animats 6 (SAB00)*, Cambridge, MA, 2000. MIT Press.

[6] Joanna Bryson and Brendan McGonigle. Agent architecture as object oriented design. In Munindar P. Singh, Anand S. Rao, and Michael J. Wooldridge, editors, *The Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL97)*, pages 15–30, Providence, RI, 1998. Springer.

[7] Joanna Bryson and Kristinn R. Thórisson. Dragons, bats & evil knights: A three-layer design approach to character based creative play. *Virtual Reality*, page in press, 2000.

[8] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

[9] Jonathan H. Connell. *Minimalist Mobile Robotics: A Colony-style Architecture for a Mobile Robot*. Academic Press, Cambridge, MA, 1990. also MIT TR-1151.

[10] Luis Correia and A. Steiger-Garção. A useful autonomous vehicle with a hierarchical behavior control. In F. Moran, A. Moreno, J.J. Merelo, and P. Chacon, editors, *Advances in Artificial Life (Third European Conference on Artificial Life)*, pages 625–639, Berlin, 1995. Springer.

[11] Scott A. DeLoach and Mark Wood. Developing multiagent systems with agentTool. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.

[12] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[13] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[14] R. N. A. Henson and N. Burgess. Representations of serial order. In J. A. Bullinaria, D. W. Glasspool, and G. Houghton, editors, *Proceedings of the Fourth Neural Computation and Psychology Workshop: Connectionist Representations*, London, 1997. Springer.

[15] K. Hindriks, F. De Boer, W. Van Der Hoek, and J.-J. C. Meyer. Control structures of rule-based agent languages. In J.P. Müller, M.P. Singh, and A.S. Rao, editors, *The Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL98)*, pages 381–396, 1999.

[16] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.

[17] Ian D. Horswill. Visual routines and visual search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[18] Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 236–243, Seattle, May 1999.

[19] Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.

[20] John E. Laird and Paul S. Rosenbloom. The evolution of the Soar cognitive architecture. Technical Report CSE-TR-219-94, Department of EE & CS, University of Michigan, Ann Arbor, September 1994. also in *Mind Matters*, Steier and Mitchell, eds.

[21] Pattie Maes. The agent network architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991.

[22] John-Jules Ch. Meyer. Agent languages and their relationship to other programming paradigms. In J.P. Müller, M.P. Singh, and A.S. Rao, editors, *The Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL98)*, pages 309–316, Paris, 1999. Springer.

[23] Karen L. Myers. *Procedural Reasoning System User's Guide*. Artificial Intelligence Center, SRI International, Menlo Park, CA, USA, 1.96 edition, 1997,1999.

[24] Alan Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

[25] Nils Nilsson. Shakey the robot. Technical note 323, SRI International, Menlo Park, California, April 1984.

[26] Nils Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[27] Bradley Rhodes. PHISH-nets: Planning heuristically in situated hybrid networks. Master's thesis, Media Lab, Learning and Common Sense, 1996.

[28] Aaron Sloman. Models of models of mind. In Aaron Sloman, editor, *AISB'00 Symposium on Designing a Functioning Mind*, 2000.

[29] Peter Stone and Manuela Veloso. A layered approach to learning client behaviors in the robocup soccer server. *International Journal of Applied Artificial Intelligence*, 12:165–188, 1998.

[30] Kristinn R. Thórisson. A mind model for multimodal communicative creatures & humanoids. *International Journal of Applied Artificial Intelligence*, 13(4/5): 519–538, 1999.

[31] Toby Tyrrell. *Computational Mechanisms for Action Selection*. PhD thesis, University of Edinburgh, 1993. Centre for Cognitive Science.

[32] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

[33] Steven D. Whitehead. Reinforcement learning for the adaptive control of perception and action. Technical Report 406, University of Rochester Computer Science, Rochester, NY, Feb 1992.

[34] David H. Wolpert. The existence of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1391–1420, 1996.

[35] David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

[36] Michael Wooldridge and Paul E. Dunne. Optimistic and disjunctive agent design problems. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.

[37] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[38] Michael J. Wooldridge and Nicholas R. Jennings, editors. *Intelligent Agents: the ECAI-94 workshop on Agent Theories, Architectures and Languages*, Amsterdam, 1994. Springer.

[39] Stephen Zimmerbaum and Richard Scherl. Sensing actions, time, and concurrency in the situation calculus. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII. Agent Theories, Architectures, and Languages — 7th. International Workshop, ATAL-2000, Boston, MA, USA, July 7–9, 2000, Proceedings*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, 2001. In this volume.