9

Production Systems

New Techniques in AAA Games

*Andrea Schiel**

9.1 Introduction

Production systems have been around since the 1940s and are now applied in a wide array of applications and ongoing research. AAA games bring a unique set of challenges to production systems; they require that AI systems be runtime efficient, deterministic, memory lean, and above all, implementable within the development cycle. Over the course of many of our titles, production systems have developed along different lines. This chapter tries to describe the majority of our more unique production systems, assuming that the reader has a basic knowledge of production systems. For readers who want to code their first production system, there is a list of references that describe basic production systems in more detail [Luger 93, Millington 09, Laird 12, Bourg 04] and these

* The author has worked at Electronic Arts for over 18 years on a variety of titles, in particular sports games. This chapter contains the insights from applying production systems to multiple AAA over many generations of consoles.

Table 9.1 Terms in Use in This Chapter

Term	Alternate
AI	AI agent or AI opponent system
Rule	Production, statement
LHS (left-hand side)	Precondition, conditional statement, if side
RHS (right-hand side)	Then side, action, postcondition
Rules database	Rules set, working set
Variable	Operator, assertion symbol, datum, working memory element, fact
Working memory	Input, assertion set, perceptions, knowledge, set of facts
Scripting language	Predicate logic, symbolic script
Matching stage	Rule binding, LHS evaluation, variable unification
Selection stage	Rule selection, conflict resolution
Execution stage	Act, RHS evaluation

should get you started. This chapter will step through some of the design choices you might make for a more advanced or specific system and presents some of the innovations in our AAA titles.

9.1.1 Terminology

Terminology surrounding production systems can be confusing since it varies depending on the source of the material. For clarification, Table 9.1 lists the terms in use in this chapter and some common alternate terms.

9.1.2 Design Considerations

Coding your own production system can be very rewarding but there are some choices that need to be made beforehand. Skipping one of these decisions has caused problems in development in the past and many of these can be found in postmortems on production systems:

- What decisions is the system trying to make? (scope and domain choice)
- How are rules represented? (rules representation)
- How will rules be authored?
- How does the left-hand side (LHS) evaluate? (matching systems)
- What happens if the AI fails to find a rule?
- What happens if there are multiple rules? (selection algorithms)
- Execution and the design of the right-hand side (RHS)
- How will the system be tuned?

The following sections go into more detail about each one of these points.

9.2 What Decisions Is the System Trying to Make?

Like any other architecture, production systems are good at certain tasks and not at others. If the decisions are particularly simple, a rules-based system is probably overkill. However, if the game doesn't break down into nice discrete states and the decisions need to reflect

different scenarios, a production system (or variation thereof) would be appropriate. At a minimum, a production system can be used when

- AI decisions are based on a variety of factors and are not easily quantified
- The system needs to respond quickly to changes in state
- There is an *expert* with knowledge of the game that needs encoding and that knowledge doesn't break down into a nice algorithm
- The actions for the system are independent of each other

The key decision is which part of the game the production system should be applied to. For example, production systems could make decisions for only part of the AI's process: when to do a trick or execute specific plays—though even these more narrow scenarios still need to fit the aforementioned criteria. A more specific AI can help limit when the system runs, which will help contain the runtime cost, though spikes in performance can still be an issue.

Alternatively, the production system could run all of the AI's decision logic. This sort of system requires more extensive optimization, and memory may be an issue since the scope of the AI is broader. The same can be true if the AI is being used for all of the simulation. Production systems can also be used for non-AI logic—such as for a game analysis system like color commentary for a sports game or for a contextual help system. The scope of these applications are harder to determine and handling the case when no rule triggers can be critical.

Performance-wise, production systems have a heavier update cost than most decision trees or state machines. Memory is required, but not necessarily more than what might be needed for another AI system. The rules themselves tend to have a small footprint, but they do add up quickly as a *typical* AI agent can require 100–500 rules. Different applications or more narrow applications of the system may require fewer rules, of course. There is also a hidden workflow and tool development cost in that an expert needs to author the rules, be able to iterate on those rules, and debug behaviors.

9.3 Choice of Rules Representation

The way rules are described is probably the biggest difference between the way these systems are implemented in games, compared to those used in research and other industries. The actual scripting language varies between systems: in CLIPS [Riley 13], it's C; SOAR has its own symbolic language [Laird 12]; and in our games, the language is very specific for the domain that the rule is being used for. In larger production systems, the scripting language supports predicate logic and unification to allow more flexibility in deciding which rules are applicable. This is often expensive both to develop (if you're coding your own) and to run. Due to the cost of unification, there are algorithms like Rete [Schneider 02, Millington 09] that optimize the matching. The advantage of a full predicate calculus (like SOAR) is that the language is very flexible and can describe a large variety of scenarios.

In deterministic games, however, unification is often difficult to take advantage of, and when it is removed, the development of a rules representation can be simplified. Unification can make it difficult to optimize for the scale of the AI. In the majority of our

games, a more custom, smaller, and specific language was used. The elements of the LHS are designed to reflect the game's perception and state variables and the RHS are called into the animation or lower systems.

Another detail we discovered is a little unusual; sometimes, designers wanted the ability to create temporary variables. These variables may not map to a variable in the perception system but are to represent some value the designers want to check, the most common being some type of counter. In systems with a working memory, this is trivial to implement, but if there isn't a working memory, a form of scratch memory or blackboard is needed to carry these temporary variables.

One caution here is that designers might attempt to use temporary variables to introduce state into the decision process itself. This should be discouraged and the solution is usually to sit down with the designer to assist them with building a better rule. Temporary variables have also been used to introduce randomness into the system—which could be another sign that the designer may need assistance with the development of the LHS of a rule (i.e., it is being selected too often or too little). Having useful operators for the LHS of rules or building in a weighting system for rule selection can help here. The types of operators (usually logical) will help determine how extensive the rules language will need to be.

9.4 Method of Rules Authoring

Full proprietary systems often support their own editor and in these cases, the authors are effectively scripting their rules. In addition, with the ability to read definitions at runtime, the evaluation of the rules can be a fantastic asset for debugging purposes. Live editing could also be possible but in one variant system we developed, the rules are recorded.

9.4.1 Recorded Rule System

Recording systems allow for rapid iteration on rules and for very quick authoring of rules. The author runs the game, puts the system into record mode, and the system will automatically record the state of the game. When the specific type of action occurs, the state of the game is associated with the action and becomes the LHS of the rule (the action is the RHS). The disadvantage of this system is that manual tweaking of the rules is limited.

For the LHS (the recorded part), the following are necessary considerations:

- What's being recorded: which game state variables are important?
- How important each game state variable is: a weight for each variable that modifies how much that variable applies when being matched. Lower weighted variables are less necessary for a match.
- The tightness of the matching for each variable: the range of values that would be tolerated for each variable.
- How far back in time is needed to be recorded to form the LHS—is it a snapshot right before the action or a time range before the action.

In our implementation, the LHS rules produce a score that is considered a match if it surpasses a given threshold value. The rules that match form the matched rules set and the rest of the system is a classic production system. That is, the actions (RHS) go through a stage of conflict resolution and whichever rules are left after selection then run their actions.

The disadvantage is that the system works best if the rules share a lot of the same LHS variables. By the same token, sharing a lot of LHS elements does make the Rete algorithm a good choice. However, since predicate calculus wasn't in use, a custom matching system was implemented.

Determining which game state variables are needed for the LHS is a challenge. Once this step is completed, the actual authoring is very quick. A useful aid here is the ability for the system to rewind and playback the rule and to identify, at runtime, which rules are being triggered, and specific for a rule, how each of the elements of the LHS matched.

9.4.2 More Typical Systems

Nonrecorded authoring utilizes a custom editor—though the best systems seem to be the ones that read in text files and effectively compile them into rules. However, this may allow for authors to create malformed rules (which they will have to correct), but it gives them the most freedom to use whichever text editor they prefer. As such, this is one of the more accessible approaches.

Alternatively, if the grammar is quite specific, a more symbolic or graphical editor can be used. In retrospect, many designers seem to dislike graphical editors but depending on the complexity, this is an option. How easy it is for designs to author the rules will have a direct impact on the efficiency of the workflow.

9.5 Choice of Matching System

On some platforms where we are constrained by runtime performance, and unification isn't being supported, optimization of the system's matching stage is required. In one case, this led to the development of a new system that is a departure from a true production system. In this variant, with a very large rules database, the matching system stops at the first rule matched. This turned the production system into a specialized rules-based system but it did open up some unique applications. This type of selection, a type of greedy selection, is described in Section 9.5.1.

9.5.1 Greedy Matching/Selection

As an example, let's start with 500 rules in our database, numbered 1–500. In any given update, the rule matching is capped at 1 ms. For up to 1 ms, process the LHS of the rules in order. If a match is made, exit early. If a match is not made, continue to the next rule. Once a match is made, selection has been effectively completed as well since there is only one rule. That rule's RHS is then executed.

This approach created the complication that the AI would always fire the same rule for the same state. This was solved by randomizing the order of the rules in the rules database. Figure 9.1 demonstrates how this works.

- *Step 1*: Rule 1's LHS is evaluated but it doesn't match. The system tries rule 2 and so forth until it matches rule 4. Rule 4's RHS is then executed. Before the next update, the rules database is randomly shuffled.
- *Step 2*: Several updates later when the same state occurs again, the database is now in a different order. The system starts with rule 300 (which doesn't match in this example). Finally, rule 87 matches/is selected. Its RHS is then executed. (This example assumes rule 4 was shuffled below rule 87).

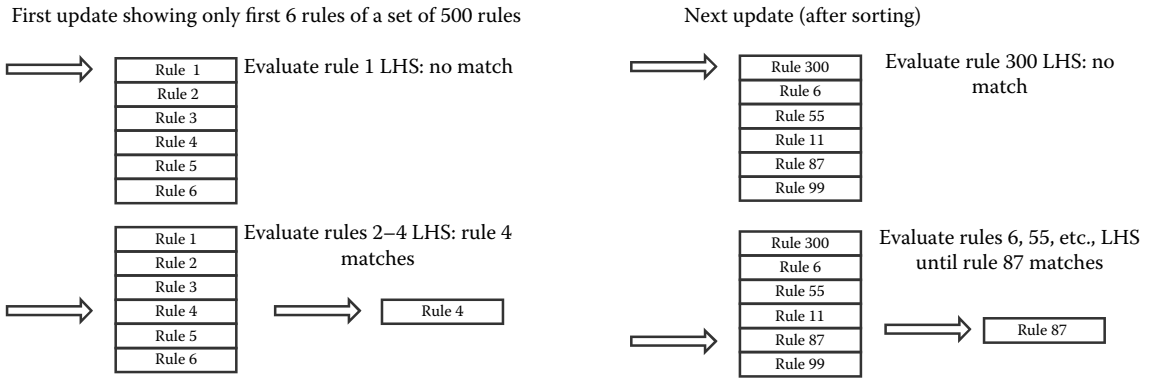

Figure 9.1

Greedy-based selection's first two updates (read left top to bottom, then right top to bottom).

This made the matching/selection processing extremely fast, allowing it to run very large rules databases on very limited platforms. In addition, we capped the matching so that only n numbers of rules were actually checked. This allows for the capping of a spike if all rules were not going to match. The capping is used because, like any list, the worst-case runtime is O(*n*), which, again, can be prohibitive if *n* is large or a rule has an expensive LHS. It's worth noting that spikes can occur during matching (even with a cap) if a rule's LHS is very expensive to evaluate. This is mitigated by optimizing the LHS variables and by limiting the number of variables allowed for the LHS of a rule.

9.6 What Happens If the AI Fails to Find a Rule?

This may seem like a simple problem, but in practice, we've run into issues with the AI stalling. The first challenge is ensuring that it's obvious when the system fails to find a matching rule. The second challenge is to determine if that's acceptable. In some games, as long as the AI already has an action, it is fine if the AI doesn't find a new rule. In other games, this can lead to an AI standing absolutely still, or a factory not producing anything, or no tricks being performed, etc. As long as this is identified, a default rule could be the solution or additional rules to fill in the missing scenarios could be authored. In one unique case, backward chaining was used. This allowed the system to work out a sequence of steps to get to a matchable state. Backward chaining is beyond the scope of this chapter, but it is possible so long as the amount of back chaining is constrained.

9.7 What Happens If There Are Multiple Rules?

When there are multiple rules—that is, when greedy matching isn't in use—the matched rules need to undergo selection. The purpose of selection or conflict resolution is to ensure that the RHS of rules do not produce actions that conflict with each other but in some cases, it is also to scope down the number of rules. For example, in one system, when the set of matched rules is very large, only the first *n* rules are selected and this limits the

number of RHS that execute and scales down the possible conflicts. If the RHS are kept as unique as possible, then the number of conflicting rules should be reduced since all rules would be able to run. We discovered it was important to track the number of times that a rule was executed (selected) since the selection algorithm was sometimes filtering out rules to the extent that they never were selected.

One system had the problem where for a few game states, a very large number of rules were always matched. The solution was to randomly select a subset, process these for conflicts (which were minimal), and then execute the subset. The random selection prevented a bias from creeping into the system from the selection algorithm (constraining the matching in this particular situation wasn't possible).

By contrast, in another system where a large number of matches occurred on a regular basis for many rules, a form of partially supervised training was the solution. The system supports a weighting of the LHS. When a rule is successful, its LHS weight is increased. If it fails, the LHS weight is decreased. This requires a selection algorithm that selects biased on a weight, a system that monitors the results of the RHS execution, and a method for the authors to tweak the amount of negative/positive feedback for the training system. The AI is then run many times in training mode against both itself and human players. After many games, the resulting database is locked and the result is a self-tuned rules database. The rules subset selected are the first *n* number of rules that matched—but not a random subset. Instead, since the subset reflects the result of training, the subset is the set of higher-performing rules.

This proves to be highly successful and it allows for the weeding out of bad rules. However, you will need to manually check for rules that are too highly rated and remove low-frequency rules. Low-frequency rules won't have as much reinforcement applied to their weights and will drift to the bottom. Likewise, you can boost the weight of a rule to increase its probability of selection if the expert feels a rule is being unfairly penalized.

9.8 Execution and the Design of the RHS

The RHS is the part of the system where an action can be taken as a result. It can be a flag, a setting of a variable, a message, or an event or any other form of implementation. In general:

- The RHS should be cheap as possible to execute
- As much as possible, the RHS should not preclude other RHS actions. That is, minimize the number of conflicts between the RHS if possible
- If implementing backward chaining, the RHS should be discretely identifiable and be reusable as a variable on the LHS
- The RHS should be able to support being called multiple times without restarting on every call or the LHS will need to check if a RHS action is already running
- There should be minimal dependency between RHS. If an RHS requires that another action runs before it, the rules need to enforce this, which adds additional checks to the LHS of the rules

Authoring of the RHS can be done in the same manner as the LHS or it can be distinct. In the example that follows, the RHS is generated during a recording session.

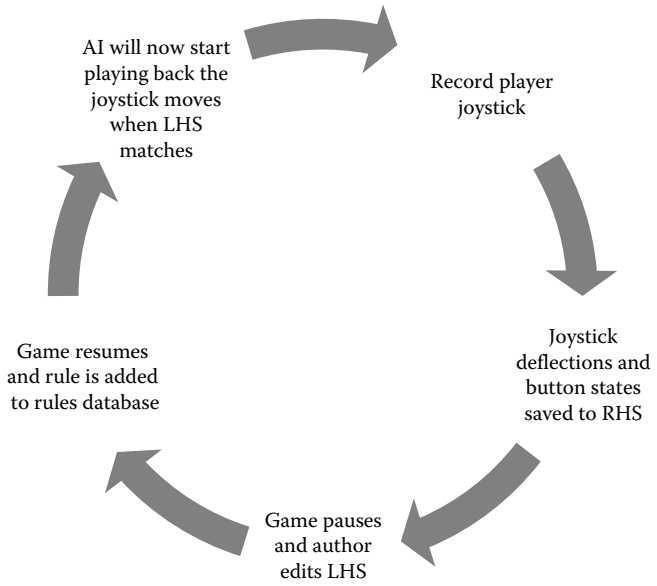

Figure 9.2

Recording the RHS.

9.8.1 More Complex RHS

One system records the joystick actions to produce the RHS of the rules, as shown in Figure 9.2. The execution for the RHS is simply a playback of the recorded joystick. This system allows a production system to be applied to a more complicated series of actions. It's very fast to author but it also requires a postediting stage so that authors can tweak the playback or discard bad recordings. Selection is also an issue since it isn't apparent when rules would be in conflict.

This is solved in two ways. In the first method, the rule is marked as unique so that if it matches, only that rule can be selected. In the second method, which applies to most of the rules, certain joystick maneuvers are identified and any rules that also hold these maneuvers are selected out. The authors could also watch the AI play, and if a rule is firing in the wrong scenario, the game can be paused, the rule opened for editing, and the LHS can be tweaked. The game then resumes but the production system reruns so that the author can ensure that their changes were appropriate. The final issue with this system is that if the player is what is being recorded, this will only work if the AI can use the data in the same way that the player uses it.

9.9 Debugging and Tuning

Several variants on how to author systems have been presented, but all of these need to be tuned and tested. Outside of the typical performance profiling, there are a couple of common key indicators that all of our systems have:

- Number of times a rule executes
- Success rate of executing the RHS

- Number of times a LHS variable matches
- List of rules that execute with a very high frequency
- List of rules that never execute

Another common debugging tool is to allow the RHS for a rule to be executed on demand—such that the behavior itself can be tuned or tweaked. Some systems support a complete reedit or rerecording of the RHS.

One implementation uses the selection step to train its AI. In training mode, the game is running with debug information available and the rules open for editing. If a rule is selected in training mode that the expert (designer) doesn't like, they can indicate that, and the system lowers the rule's weighting. Alternatively, the expert can pause the game and adjust the rule's LHS at runtime to ensure that the rule doesn't fire in that circumstance. They can likewise reward a given selection. This system requires a runtime editor for the rules and a way to reload the rules. Much more common is for production systems to log the results of matching and selection and have the *training* done offline.

In general, support for live editing can make iteration on the LHS of rules much easier. It can be difficult to author the constraints for the LHS of a rule for all possible scenarios—and having a way to edit and then rerun a scenario can help with this tuning immensely.

Logging and debug display of which rules are firing for a given AI is common. It helps to know which rules are creating the current behavior. Many systems support the graphical display for the test of the LHS variables. For example, if a variable is testing the range from one object to others, the debug might inscribe a circle to show what the radius/distance is set to.

9.10 Conclusion

Production systems have been used in a variety of published AAA titles and have proven themselves in many industries for some time now. All of the systems described are the results of the hard work of many engineers for different types of games over the course of many years. Some of these systems have evolved over time—almost all due to a practical constraint. Some are no longer production systems but all had their start with such a system and were usually developed initially over the course of 2 years.

A challenge all of these systems face is that they need to be very easy to iterate on and this has inspired new ways to author systems. Likewise, domain-specific systems have launched a variety of new ways to represent rules and to optimize matching outside of Rete and other classic algorithms. There is nothing inherently flawed with a more classic approach, but it is hoped that by presenting some of the techniques we use, readers might be inspired to think beyond the typical implementation and extend what is possible for these systems in their own games.

References

[Bourg 04] Bourg, D. M. and Seemann, G. 2004. *AI for Game Developers*. Sebastapol, CA: O'Reilly Media Inc.

[Laird 12] Laird, J. 2012. *The Soar Cognitive Architecture*. Cambridge, MA: Massachusetts Institute of Technology.

[Luger 93] Luger, G. F. and Stubblefield, W. A. 1993. *Artificial Intelligence Structures and Strategies for Complex Problem Solving*, 2nd edn. Redwood City, CA: The Benjamin/Cummings Publishing Company Inc.

[Millington 09] Millington, I. and Funge, J. 2009. *Artificial Intelligence for Games*. Boca Raton, FL: CRC Press.

[Riley 13] Riley, G. 2013. CLIPS: A tool for building expert systems. Sourceforge. http://clipsrules.sourceforge.net/ (accessed May 27, 2014).

[Schneider 02] Schneider, B. 2002. The Rete matching algorithm. *Dr. Dobbs Journal*. http://www.drdobbs.com/architecture-and-design/the-rete-matching-algorithm/184405218 (accessed May 27, 2014).