Lessons from decompiling an embodied cognitive model

Audrey Girouard[1], Noah W. Smith[1], Frank E. Ritter[2]
audrey.girouard@tufts.edu, noah.smith@tufts.edu, frank.ritter@psu.edu
[1]Tufts University (United States)
Department of Computer Science
[2]The Pennsylvania State University (United States)
College of Information Sciences and Technology
Applied Cognitive Science Lab

Abstract

Cognitive models and intelligent agents are becoming more complex and pervasive. It is time again to consider high-level behavior representation languages and development environments that make it easier to create, share, and reuse cognitive models. One of these languages is Herbal, a high-level behavior representation language. Users represent knowledge in Protégé, an ontology editor. Herbal compiles this knowledge into cognitive models in Soar, a rule-based cognitive architecture. Herbal includes the ability to automatically link the resulting models to dTank, a simple, distributed tank game co-developed with Herbal.

To understand the theoretical implications of the process of compiling cognitive models from high-level descriptions more clearly, we generated by hand the Herbal high-level description of a well-written, medium-sized (50 rule) Soar model that plays dTank. This process is a type of decompilation process, of going from low- to high-level language, that yields lessons for both the compiler and the process of modeling. Many of the constructs in the model were supported by Herbal, particularly elaborations and simple and regular actions. In some cases, Herbal's representation prevents the user from generating incomplete, incorrect or atheoretical code—we saw hand written code that cannot be generated by Herbal because it is incorrect or overgeneral. This process also highlights problems with Herbal. Certain types of theoretically sound hand-written rules do not yet possess an exact translation to the high-level language (mostly knowledge about which action to chose, and links across known representations). We have several suggestions for constructs to be added.

This process of decompilation illustrates how users are creating models and could do so more easily and less error prone with more appropriate languages, in addition to helping develop Herbal, and, if automated, this decompilation process done by hand could lead to a decompilation feature in Herbal to help explain raw Soar code.

Introduction

Cognitive models and intelligent agents are becoming more complex and pervasive. It is time again to consider high-level behavior representation languages and development environments that make it easier to create, share, and reuse cognitive models. High-level languages provide a way to understand architectures of cognitive models.

One of these languages is Herbal (Cohen, Ritter et al. 2005 2005). As a collection of tools, Herbal is a development environment for producing cognitive models in Soar (Lehman, Laird et al. 2005), a rule-based cognitive architecture. Users represent knowledge in Protégé, an ontology editor (Stanford Medical Informatics 2004), and Herbal compiles this knowledge into cognitive models in Soar, a rule-based cognitive architecture. Herbal includes the ability to automatically link the resulting models to the dTank environment (Morgan, Ritter et al. 2005), a simple, distributed tank game co-developed with Herbal.

We are exploring the theoretical implications of the process of compiling cognitive models from high-level descriptions. This knowledge is important because it is a methodology for studying models and cognition, as well as a way to understand agent and cognitive architectures.

Decompilation

One way to understand a high-level language is to use a low-level one and "decompile it." In other words, it is necessary for a high-level language to retain as much of the expressiveness of the low-level language as possible, all the while providing the power that another level of abstraction brings. Thus, a high-level language must be able to reproduce at least the behavior of a program written in the lower-level language, if not the exact code. Furthermore, the high-level language should ensure that it compiles to well-formed constructs in the low-level language, and in the process eliminate both common bugs and poor programming practice.

To evaluate the Herbal high-level language on these criteria, our first step was to construct a Herbal model that reproduced (as closely as possible) the functional elements of a well-written, medium-sized (50 rule), Soar model called basic_tank (Councill 2003), included with the dTank environment. Our main objective with this exercise was to determine how well Herbal retains the expressiveness of the Soar language. By working from actual Soar programs rather than from the Soar language specification, we sought to focus our efforts on the commonly used constructs, rather than the full extent of Soar 8.5.2.

After noting the details of how this works with Soar, we will attempt to draw some general lessons for all high-level modeling languages. Those not interested in the details might wish to skip to the conclusions.

Operator and State Decompilation

The most basic construct in the Soar modeler's toolbox is the operator, which (in effect) modified working memory as the model is run. When the pre-specified conditions for its application are present, the operator attempts to create changes in working memory, or by means of the output mechanism, allows an model to act on its environment.

This basic construct must be preserved in Herbal, at least in this very general sense. In Soar models, well-formed operators are usually programmed with corresponding linked similar *propose* and *apply* productions, where the *propose* production matches the conditions required for the operator to be executed to completion, and the matching *apply* production actually finalizes the action. Thus, multiple operators may be proposed for any memory state, allowing Soar to choose the most appropriate one (based on its conflict resolution scheme for choosing operators and additional knowledge).

Herbal follows the same general pattern to provide the same functionality to the high-level modeler. Operators[1] are compiled into matching *propose* and *apply* productions, sharing a common name prefixed by the type. These Operators may be composed of multiple Conditions and Actions, and Conditions and Actions may be used in multiple Operators within the same project. In Soar, operators must all access the current state (the current state of working memory). Similarly, in Herbal, Operators must be associated with a State, which is inserted by the compiler into the condition of the *propose*. Herbal supports multiple, named sub-states of the TopState, but in our experience this is not a feature that is often utilized in Herbal models.

Herbal compiles an Operator into two productions: matching *apply* and *propose* rules that are named after the Operator being compiled (Table 1: lines 1, 10). The name of the State associated with the Operator is affixed to the **state <s>** statement of both productions (lines 2, 11). Any associated Conditions are collected and, for every conditional attribute of the state, a StateMatch entry is created in the *propose* production (line 3). The Exp field of the Conditions are included below the state match section of the *propose* production (line 4). Lines from the OperatorMemory section of the Conditions are copied into the operator memory section of the *propose*'s action (lines 7-8). For all Actions associated with the Operator, entries in the OperatorMemory sections are similarly copied into the analogous section of the *apply* (lines 14-15). The Exp of the Action is copied into the consequence of the *apply* rule (lines 19-20). Actions do not have StateMatch fields. Note that the OperatorMemory sections of both productions are identical (in this example), and also the lines annotated as exceptions.

Table 1 : Two productions from the water-jug example in the Soar Tutorial (Laird 2004)

```
1    sp {water-jug*propose*fill                Operator name: fill
2        (state <s> ^name water-jug            Name of state associated
                                               with Operator
3                   ^jug <j>)                  StateMatch of Condition
4        (<j> ^empty > 0)                      Expression of Condition
5    -->
6        (<s> ^operator <o> +)                 Operator preference
7        (<o> ^name fill
8             ^fill-jug <j>) }                 OperatorMemory of Condition
9
10   sp {water-jug*apply*fill                  Operator name: fill, again
11       (state <s> ^name water-jug            Associated state name,
                                               again
12                  ^operator <o>
13                  ^jug <j>)                  ← Exception 1
14       (<o> ^name fill                       OperatorMemory of Action
15            ^fill-jug <j>)
16       (<j> ^volume <volume>                 ← Exception 2
17            ^contents <contents>)
18   -->
19       (<j> ^contents <volume>)              Expression of Action
20        <j> ^contents <contents> -)}
```

[1] For clarity, Herbal concepts are distinguished here by a capital letter: Operator, Condition, Action, StateMatches, OperatorMemory, Exp.

Herbal makes several assumptions about the nature of *propose*/*apply* pairs. If a modeler desires to have the operator memory of the *apply* match the operator memory of the *propose* and does not need to elaborate elements from the state match, then this compilation strategy will hold.

Unfortunately, there are two common Soar constructions that are not supported. Exception 1 (Table 1: line 13) shows that there is no way to insert additional elements into the state match section of the *apply*. In the example in Table 1, the element required had been specified in the *propose*, so perhaps that knowledge should be carried down into the apply during compilation, if necessary. The high-level ideal of a single Operator for which there are Conditions and Actions is very powerful, but perhaps in this case it would benefit from either more communication between the reference requirements of the Action and the original matching memory items of the Conditions, or else better ways to further elaborate the Actions.

Exception 2 (lines 16-17) further illustrates the benefits that the latter option would bring, as there is currently no way to add such code which is external to both the state match (which is automatically generated) and the operator memory (which often is identical to the *propose*). The solution to this issue would seem to be a third field for Actions, which would act in much the same way as the Exp of the Conditions (see line 4) acts for the *propose*.

While we are fairly sure that there is no way to exactly duplicate the code in the exception areas of Table 1, we concede that it may be possible to accomplish the same effect through other mechanisms in Herbal. However, since one of Herbal's objectives is to de-obfuscate the modeling process, those mechanisms should either be obvious or else well documented, and have found that both need to be improved in Herbal.

The example given in Table 1 raises an additional issue. If a modeler wishes to change in Soar the value stored under a specific name in working memory that arises, for example, in the code, it is not as simple as overwriting the old value with a new one. Soar requires removal of the previous working memory item and the creation of a new one. Observe line 19 in Table 1, in which an element **contents** associated with the object **j** (a jug found in the current state) is added to memory with a value corresponding to the capacity (**volume**) of that jug, thus "filling" the jug. Note also line 20, which removes the old value. This is a very common thing for an Action rule in Soar to do, and we suggest that Herbal make such actions as easy as possible for the programmer. Instead of filling in the appropriate Soar code to accomplish the task in the fields of the Action (which we have shown above to be difficult, if not impossible for this example), perhaps the modeler could be presented with a dialog which would ask which elements of what object should be added to working memory under what name, and whether the old value should be removed. If the case of an overwrite command, it could also add an extra condition clause with this object as well as the relevant attributes and their current values, eliminating the specific problem of lines 16-17. This and related wizards (covering mathematical and logical operations on objects in memory, for example) would make Herbal feel much more like a genuine high-level language with a state of the art IDE, allowing manipulation of objects and variables rather than simply a high-level interface frequently requiring the modeler to lapse back into writing low-level code. When programmers write in a high level language such as Java, they never have to write byte-code to specify exactly how objects and variables should be handled in memory, and it would seem that high-level modeling languages should grant their users the same support.

This analysis makes another suggestion as well: for the development of Herbal. In trying to replicate the code in Table 1 with a Herbal model, we found unexpected behavior in the compilation of the OperatorMemory (lines 6-7). Given the code `^fill-jug <j> in a Condition`, the Herbal compiler does a little extra (unexpected) work, producing the code `^j ^fill-jug <j>`. This is a bug in the Herbal compiler, for this sort of statement is compiled as expected in StateMatch expressions.

To conclude this discussion of operators, let us examine the abstraction of States in Herbal. In a Soar model built from a Herbal model containing multiple, named States, the conceptual switch between states happens when the old **<s> ^name** is removed from memory and a new one is added, in a similar process to that discussed above, pertaining to the action of lines 19-20 of Table 1. Herbal should therefore support an easy and straightforward way to instruct an Action to transition between States, perhaps with the same "wizard" approach advocated above.

Elaboration Decompilation

Elaborations are high-level concepts in which a production recognizes when a certain condition is truein working memory and creates appropriate working memory elements as a consequence. This is useful for creating memory objects that are conditional on ones currently available, initializing memory elements before they are used, linking states, changing selection preferences, and simply reporting on the state of the model as it is running. These concepts are usually be encoded in a single Soar production (as opposed to operators, which require two). They are designed to support faster, but more ephemeral changes to WW - when the condition no longer is true, the changes are removed, whereas with operators, the changes stay. The default Herbal Ontology supports the concept of a basic elaboration, in which a single production (prefixed by **elaboration***) is compiled from a set of Conditions and a set of Actions, and must be associated with a single, particular state. Since there is no need to transfer operator memory across two productions, the compilation is very simple: the conditions and actions are copied verbatim into the Soar production. This implementation is simple and elegant, although the modeler does have to write low-level code components. In the following paragraphs, we suggest several different categories of Elaborations that we feel should be available to the Herbal programmer.

State Initialization elaborations should be able to be compiled in such a manner that they ared not associate with any named state, but instead are more general in their implementation. We suggest adding a distinct entity called "Initialization" that is a subclass of Elaboration. As per Soar conventions, this new construct should generate productions with the prefix **initialize***.

Initializations are also sometimes accompanied by "link" rules. These rules are used to pull information from one state into another. Currently, it is impossible to create these links across known representations, between states. A new construct should be available to perform this duty, with the prefix **link***. The following example was taken from the basic_tank dTank agent, and illustrates usage.

Table 2 : A link production from the basic_tank agent example

```
sp {link*directions*to*substates
    "Links direction knowledge to substates."

    (state <s> ^superstate <ss>)
    (<ss> ^direction-knowledge <dk>)
-->
    (<s> ^direction-knowledge <dk>)}
```

Selection elaborations are productions facilitating a selection between currently co-eligible operators. Selections are typically ignorant of state name information, so they should be able to be created without association with any specific Herbal State. We suggest that selection rules could be created as a distinct entity in Herbal, generating productions with the prefix **select***.

Finally, Monitor elaborations do not significantly modify working memory, but rather output information relevant to the current situation for the user to monitor the current status. Because this is a standard debugging tool for Soar modelers, this abstraction should certainly be added to in Herbal. The compiled production should have the prefix **monitor***.

Compound Production Decompilation

There are times in writing a Soar model when the modeler wishes to create a series of linked productions, where (for example) a single *propose* can have multiple *apply*s. One reason this is done is in the case where one action is required the first time an operator fires, and a different action is required the second time in basic Tank. Because Soar does not support logical program control statements such as **IF**, this is usually accomplished with two *apply* productions, named (for example) **apply*first*time** and one **apply*rest*of*the*time**. Due to Herbal's strict interpretation of the operator concept as being composed of two productions, it is not straightforward to encode such an advanced operator. We suggest that there should be some sort of support for this functionality more directly than having two operators.

Conclusion

The decompilation process we have employed has proven to be a useful methodology for studying both low- and high-level modeling languages. This approach of decompiling provides a way to see how these two levels are related, and has served to highlight the strengths and weaknesses of both modeling environments. A breakdown of how Herbal was able to re-model the basic_tank dTank program is below, in Appendix A. In analyzing actual models we feel that we have made comments and suggestions that will most help the development of Herbal, as we have tried to be very mindful of the needs of the modeler. These suggestions are summarized in Table 3.

Table 3 : Summary of suggestions for Herbal

1. Expression of the condition of an Action	Provide the possibility to add code external to both the state match and the operator memory
2. High-level libraries	Support a high-level library for mathematical and logical operation on objects in memory

3. Construct for replacing a working memory object	Support of a dialog to make working memory object replacement easier
4. Initialization elaborations	Creation of a new Herbal object similar to Elaborations, but not associated with any particular state
5. Link elaborations	Creation of a new Herbal object to support passing information between different states
6. Selection elaborations	Creation of a new Herbal object to select between actions the agent can take
7. Monitor elaborations	Creation of a new Herbal object to support output of information relevant to the current situation
8. Compound Productions	Support the functionality of linked productions

In conclusion, we would note that it seems possible to automate the process of decompilation on well-written Soar code-bases, assuming that Herbal supports a broad enough level of expressiveness. Such a script could help foster reuse of Soar code currently in use and promote the use of the Herbal platform and apply to other high-level behavior representation development efforts.

Acknowledgement

???

References

Cohen, M. A., F. E. Ritter, et al. (2005). Herbal: A high-level language and development environment for developing cognitive models in Soar. 05-BRIMS-044, Orlando, FL, Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation.

Councill, I. (2003). Basic_tank, Penn State University.

Laird, J. (2004). The Soar 8 Tutorial, University of Michigan.

Lehman, J. F., J. Laird, et al. (2005). "A Gentle Introduction to Soar, An Architecture for Human Cognition: 2006 Update."

Morgan, G. P., F. E. Ritter, et al. (2005). dTank: An environment for architectural comparisons of competitive agents. 05-BRIMS-043, Orlando, FL, Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation.

Stanford Medical Informatics (2004). Protégé (Version 3.1.1).

Appendix A: Analysis of Herbal's support for the constructs contained in the basic_tank program

An analysis of the productions rules from the basic_tank program shows that almost all of the *propose* operators can be decompiled in Herbal, with the exception of the bug found in the compiler for OperatorMemory objects. Most of the *apply* operators cannot be modeled, generally because of a missing feature in Herbal, mostly because of the the inability to add state matches and/or operator memory in the condition of the rule, but also sometimes because of compound productions. All elaborations can be reproduced exactly. Other constructs such as initialization, link and select elaborations have not been exactly reproduced, mostly because of association with states. Finally, a few rules were not analyzed.

In the basic_tank program, *proposes* represent 26% of the productions, *applys* 34%, elaborations, initializations, link and selection productions 32%, with a 8% other productions. Most of those could be reproduced with the suggestions proposed.