The Architecture of the Crow Compiler and Runtime System

Taybin Rutkin

May 3, 2000

1 Introduction

Crow is a mixture of Icon[1] and MDC-90. Icon is a very high-level procedural language with logical features such as generators and backtracking [2]. MDC-90 is a superset of C with additional features to give it concurrent messaging abilities [1]. Crow gives to Icon the features that MDC-90 gave to C.

The two, while separate designs with their own architectures and purposes are very closely linked to each other. The Crow compiler has to generate the runtime system at compile time. This led to an interesting development as I had to design each one concurrently. I decided that the best way to do this was to take a Crow program and compile it by hand into Icon. Then I wrote a program that would do that for me. After that, I generalized the compiler so that it would work for other Crow programs. Of course, I foresaw most of the places where the compiler would need to be generic, so the steps were not quite as clean cut as it may appear.

1.1 A Very Brief Overview of the Crow Language

Messaging abilities allow a programmer to break an algorithm into smaller parts. These smaller parts are spread over several *nodes* and are processed concurrently. The nodes communicate to each other through *messages*. The nodes check which messages they have received and whether they fit a *pattern*. If a pattern fits, its corresponding *behavior* is launched. A behavior can have multiple patterns which all must be true for the behavior to launch.

[1] Icon is also the language the compiler was written in.

A behavior is equivalent to a procedure only instead of being given a name, it is identified by its pattern. Each behavior belongs to a *class*. Inheritance and even multiple inheritance are both allowed. A pattern is made up of assignments, comparisons, and *precedence*. Assignments fit if a message is available to be assigned. Comparisons use relational operators to compare the number of messages to an integer. Since multiple patterns can fit, each pattern can have a precedence, which is an integer from 0 to 255. The behavior with the highest precedence, in case of multiple matches, is launched.

Nodes are referenced through the `<S,X>` syntax. The `S` is a unique *symbol* that points to the node. The `X` is a non-unique integer that augments the symbol. So a program could have one symbol and just increment the `X` argument. The syntax `<S>` is also allowed. In this case, `X` is zero.

A symbol is created by the `symbol()` function call which returns a unique symbol. The class type that the symbol should represent is passed to the function. The returned symbol is not only unique among all the other symbols, it also represents the class of a node.

The best way to think of the semantics is to imagine each node having a queue for each message type. When a certain number of messages are in each queue, a behavior is launched and the messages are processed by that behavior. For a more in depth explanation, please refer to [4].

1.2 Goals of the Crow Compiler and Runtime System

I had several goals which influenced the design of the Crow compiler:

1. The compiler had to be correct. Even if a certain input was uncommon, it had be treated correctly.

2. The runtime system had to correctly simulate concurrency.

3. I wanted as much of the runtime system to be as generic as possible. In other words, I wanted to design a system that would require very little modification for each separate Crow program.

4. The output of the compiler was to be a self-contained Icon program. This program would not have to link with any libraries that were not distributed with Icon.

5. The design of the compiler had to be modular. This was not to make adding and deleting units easier. Once the compiler is correct, I do not foresee any necessary changes. By giving each procedure a certain task, if that task needed to be rewritten I could do that without worrying about breaking other parts of the program. The modularization also helped by using divide and conquer to reduce the intimidation of writing a compiler. Breaking the compiler into smaller pieces let me work on one procedure at a time.

All the goals were successful except for the correctness of the compiler, which I consider to be a partial success.

2 The Crow Compiler

As figure 1 shows, the compiler is actually made of several sections. I have grouped the filters into units according to what they do.[2] The compiler uses a pipes and filter architecture for the input group. Since data is not passed between the filters in the output group, the arrows show the order of operation more than anything else.

The compiler reads in a file and then sends each line to the input group one at a time. The line is then sent to all the filters in that group. So there is no true parsing in the sense of `lex` and `yacc`. This is why I consider the correctness of the compiler to be a partial success. Most inputs are accounted for and are parsed correctly. But since the input is not split into tokens and parsed using a grammar, if a Crow expression[3] is split over two lines, it will not be detected. The reason for not implementing a correct parser was that Crow and Icon have an incredibly complex grammar. Icon's compiler, which has existed for twenty years, does not use a lexer either.

2.1 The Preprocessor Unit

The two filters in the preprocessor unit translate text to text. I decided that each location could be represented as a unique integer[4]. So every location is

[2]The actual procedures are not necessarily in this order. But they should be. They are in the order that they were implemented.

[3]As opposed to a vanilla Icon expression.

[4]I had already decided to represent symbols as integers.

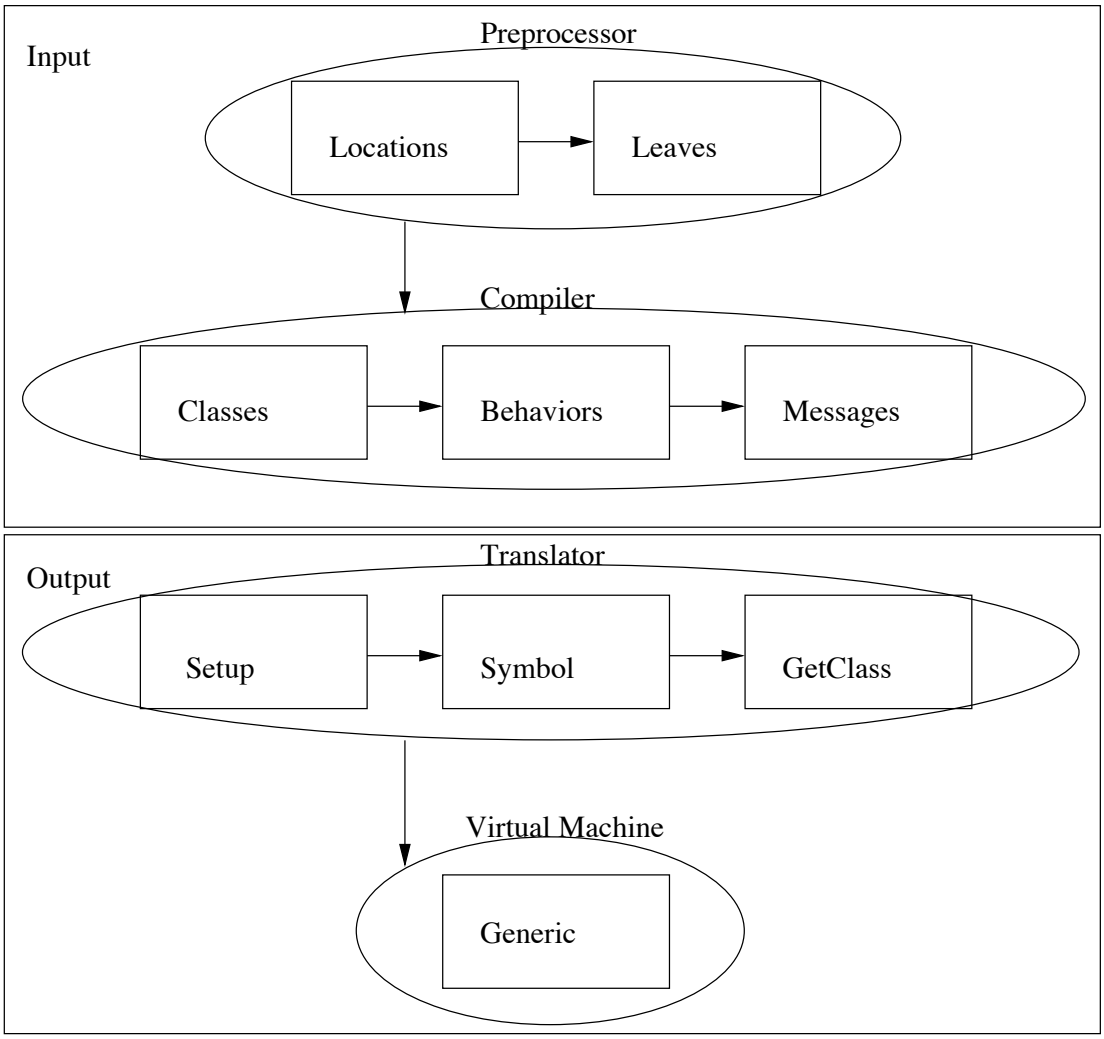

Figure 1: The filters of the Crow compiler.

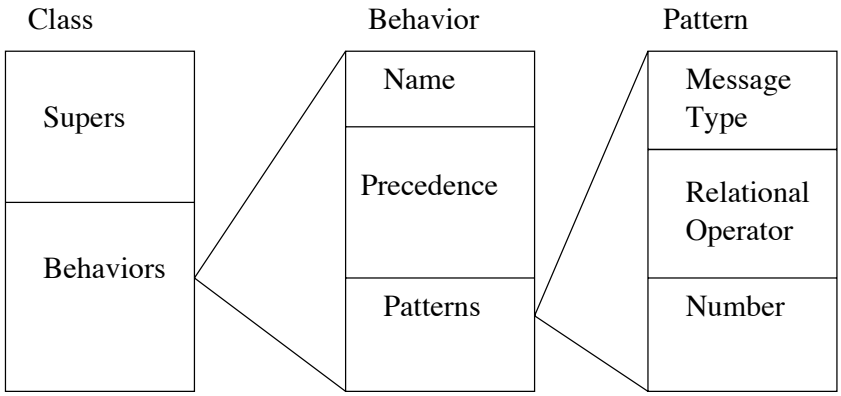

Figure 2: The compiler creates this data structure for each class during compile-time.

translated into a bit-or function. So, `<2,4>` is turned into `loci(2,4)`, which in turn calls `ior()`. This indirection is necessary because I also wanted to allow a *location* data type to be passed to `loci()`. `loci()` accepts both integers and the *location* data type and then calls `ior()` internally.

This was an unexpected departure from my original design. At first, `<S,X>` was translated into `ior(S,X)` directly. Then I found that additional functionality was necessary. If my program had been less modular, it would not have been so easy to extend. As it was, all that was involved was modifying a single filter to output `loci(S,X)` instead of `ior(S,X)`.[5]

The other filter just converts `leave()` function calls to `send()` function calls. The `leave()` function accepts one argument, a message, and sends it to the current location. Since Icon has no sense of *here*[6], the function has to be converted to a call that passes the current location. If `leave(foo)` is found in the line, it is converted to `send(foo, here)`.

2.2 The Compiler Unit

The filters in the compiler unit are responsible for reading the message, class, and behavior information and saving it in the data structures indicated in figure 2. I would have preferred to use an object oriented design. I think encapsulation leads to better design. Unfortunately, I was not sure how to

[5] And the `loci()` procedure itself, of course.

[6] `here` is analogous to C++'s `this` keyword. It allows for a kind of self reference.

proceed, so I wrote the filters one at a time as I figured out their respective designs. Since I used a pipe and filter design, I could implement the compiler one part at a time. This allowed me to have a vague notion of how the filters would interact while avoiding having to detail the entire architecture. If I had used an object oriented design, I would have had to be more precise in the design at an earlier stage.

I found, compared to the rest of the project, designing the data structures in figure 2 took the greatest amount of effort. Before I had decided what structures to use, it was impossible to write any significant code. Even though I did not use an object-oriented architecture, I knew from previous experiences that it is better to make the algorithms fit the data structures instead of vice versa.

The compiler unit also provides some help for the virtual machine unit later on. For instance, when a new behavior is added to the behavior list, instead of just appending it to the end, a special procedure, `insertbehavior()`, inserts it in order of precedence. Later on, when the runtime system is looking for behaviors that match, if one matches, it is known that it has the highest precedence and more searching is unnecessary. Not only is this more efficient, but it also simplifies the design of the virtual machine.

When a behavior is found, its pattern is sent to the `scanbehavior()` procedure, which parses the string and creates the appropriate data structures. The logic for this procedure is very complicated. When I first wrote it, I left an important feature out. I found that I was unable to modify the original and instead I had to write a new procedure from scratch. The current form is also very complicated and if it had to be modified, it would have to be rewritten as well.

This was not a problem. The complexity is contained within one procedure with a defined input, output, and side effect. Given these constraints, there should not be a problem with the procedure being rewritten as long as it is within the same constraints.

The side effect of `scanbehavior()` procedure is that, after scanning the behavior, it calls `insertbehavior()` to add the behavior to the behavior list. This was necessary to maintain a consistent logic across procedures. I would have preferred a cleaner approach, but since the side effect is clearly shown and documented, I do not see this as a problem.

2.3 The Translator Unit

The translator unit is where the switch from processing input to formatting output is located. The major part of this is the `setup()` filter. The `setup()` procedure creates the data structure shown in figure 3. This procedure shows how the translator unit departs from the pipe and filter style.

Instead of having a single input which the procedure acts upon and a single output which the procedure writes to, there is a data structure which the procedure performs some action on. This makes this procedure use an object oriented style. The difference is that in a pipe and filter style, if the input and output data structures were changed, the procedure would need very little change. In an object oriented style, if the data structure is changed, then the rewritten procedure could be radically different.[7]

The `symbol()` and `getclass()` procedures also showed this change in architecture. They, too, were dependent on a data structure for their form. The only input they accepted was a list which they did not act on. The list was only necessary for their output, which they achieved through a side effect on the list.

2.4 The Virtual Machine Unit

This one filter adds the generic virtual machine procedures to the output. I think I was successful in making the majority of the virtual machine generic. Five procedures are generic compared to three that need to be generated at compile time. These five the `main()` procedure, the `send()` function, the `fitspattern()` procedure, the `loci()` procedure, and the `decodelocus()` procedure. Since these procedures *are* the runtime system, the interesting ones will be described in the section on the runtime system.

I am especially proud of my success with the virtual machine. It was the result of sitting down and figuring out what procedures were necessary and what they should do. Once they were defined, it was just a matter of writing them all.

I had considered putting the generic procedures in another file which would be linked to the compiler output at compile time. But I realized that this would actually be more difficult. It was just easier to append them to the output and then send the output to the Icon compiler.

[7]Depending on the amount of change, of course.

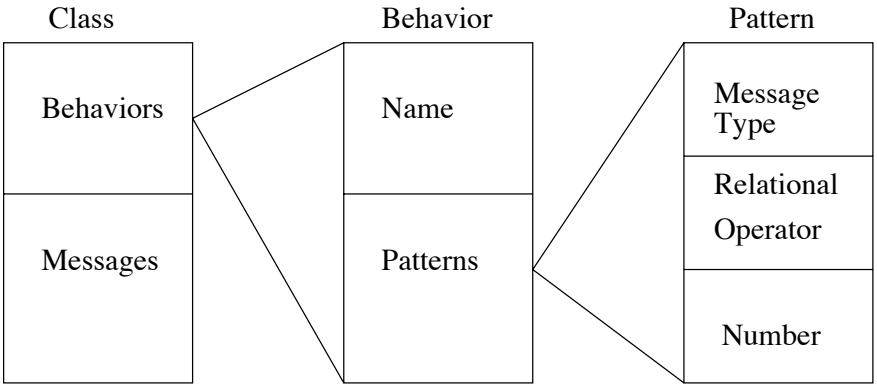

Figure 3: The runtime system's data structures.

3 The Crow Runtime System

The overriding goal of the runtime system's design was to simulate concurrency correctly. This was successfully accomplished by using a loop. The loop iterated over a list of locations that had messages sent to them. Each location is checked to see if a behavior is ready to be launched.

Most of the runtime system is devoted to checking patterns and sending messages. Since the size of the runtime system is so small[8], it was hard to give it a consistent architecture. But if I had to try and categorize it, it is fairly close to an object oriented design. This is because there are a few data structures and most of the procedures deal with those structures. The program was not written with an object oriented style in mind, but if I were to clean up the design, that would be one of the easiest switches. It is so close already; it would only involve renaming some procedures and changing some parameters.

The biggest problem that came up during the design of the runtime system was confusing the compiler's data structures, shown in figure 2, with the runtime system's data structures, shown in figure 3. As one can see, the two were almost identical except for a couple differences. These differences were:

1. A class description did not contain a list of superclasses. This was because after the translation, the superclass' behaviors had been integrated into the regular behavior list.

[8]The runtime system only has eight procedures. This does not include the various data structures.

2. A *messages* list had been added. This was a list containing a node for each message type. Each node contained a queue for the messages.

3. The precedence of a behavior was not listed. Listing the precedence was no longer necessary because the behaviors were listed in descending order of precedence.

The structures might have been designed better, but the important thing is that they were designed. This allowed me to write the procedures. Any problems with the structures that occurred later could be fixed.

The `send()` procedure was one of the key procedures in the runtime system.[9] But it did not work with *portals*, which were an additional layer of indirection for symbols. Because `send()` was a single procedure with a fixed behavior, I could modify it, and as long as it had the same behavior, it did not matter how I changed it. This was a weaker type of design by contract since it was not enforced but relied on the programmer reading the specifications for each procedure [3].

4 Conclusion

I would have been more comfortable with Icon if it had a procedure overloading feature similar to Java or C++. There were several procedures that accepted more than one datatype for the same variable.[10] While I certainly understand why procedure overloading would be impossible in a type-less language such as Icon, I think it would have emphasized the division between the multiply functionalities. The only work around was to start those procedures with an `if`-statement branching mechanism to check for each expected datatype. It works, but it led to some unclear procedures.

It seems to me that the supremacy of the data structure over the procedure is the most subtle but important benefit of object-orientation. The reason for that is that data structures give the algorithms a form and a purpose. If procedures are written without the data structures established, it leads to spaghetti code as the procedures will tend to rely on each others' internals. With a fixed data structure, a procedure[11] can concentrate on doing one thing and doing it well.

[9] It is the main function call of the Crow library.

[10] Most notably the `send()` procedure, which checks for three datatypes.

[11] And programmer!

All the procedures in the translator and virtual machine units only performed output through a side effect on the list that was passed to them. This seems to safely remove it from conforming to the pipes and filters architecture. I think that it is important to be flexible when it comes to architectures. When using different styles, make sure that their separation is distinct. But I have no problem with mixing different styles in a project this size.[12] If the input section is separated from the output section, it should not matter what architectures they use, as long as they are internally consistent.

References

[1] Thomas W. Christopher. *Experience With Message Driven Computing and the Language LLMDC/C*, chapter 2, pages 15–28. Illinois Institute of Technology, 1990. `http://www.cs.iit.edu/~mdc/docs/mdc/exllmdc.ps`.

[2] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-To-Peer Communications, 3rd edition, 1997.

[3] Bertrand Meyer. *Interactive Software Engineering*, chapter Applying "Design by Contract", pages 40–51. IEEE, October 1992.

[4] Taybin Rutkin. The Crow programming language. `http://www.clarku.edu/~trutkin/Crow/spec/spec.pdf`, March 2000.

[12] 455 lines at the last count.