An Improved Language for High Level Control Flow Semantics Definition

Richard Gankema
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.gankema@student.utwente.nl

ABSTRACT

This paper presents CFSL+, which is based on CFSL, a visual language for the specification of control flow in programming languages. In contrast to CFSL, the notation of CFSL+ was designed specifically with readability in mind. To achieve this, a set of design principles called the "Physics of Notations" was used. These principles, based on theory and empirical evidence, provide a framework that can aid visual language designers in developing notations that are cognitively effective. A preliminary evaluation using these principles shows that CFSL+ is more effective at communicating information than CFSL, although a user evaluation is still required to confirm this. A mapping from CFSL+ to CFSL is also presented.

Keywords

CFSL, Control Flow Semantics, Visual Language, Programming Language, Cognitive Effectiveness, "Physics" of Notations

1. INTRODUCTION

Like any language, a programming language has a grammar. The grammar defines the syntactical structure of the language, and is usually formally specified using a language such as Extended Backus-Naur Form (EBNF). Besides syntax, a language also has semantics, which defines the actual meaning of language constructs.

Unlike EBNF for syntax, no standard language exists to express the semantics of a programming language. Instead, the semantics of programming languages is usually described with natural language. This is problematic for several reasons: because natural language is inherently ambiguous, it can be hard to give a non-ambiguous explanation of more complex elements. But more importantly, it makes automated reasoning or correctness proving more difficult.

A part of the semantics of many programming languages is the control flow semantics. This defines in what order statements are executed, which is an important aspect of any imperative language. A first attempt to define a language for the specification of the control flow semantics of a programming language was presented by [7]. This language is called Control Flow Specification Language (CFSL). In CFSL every statement of a programming language is modelled by a *control flow specification graph* (CFSG). When all statements in a programming language have been modelled, flow graphs for programs written in that language can be automatically generated. Both the specification of CFSGs and the generation of flow graphs are implemented using a graph transformation tool called GROOVE.

CFSL is successful in that it can be used to specify the control flow semantics of a real programming language [8]. However, like most other visual DSLs, little attention has been paid to the visual syntax of the language [5]. As a result, graphs for non-trivial statements quickly become hard to read. Because of this, CFSL in its current state is not suited for explaining the control flow of statements to a human developer. But more significantly, specifying control flow semantics using CFSL is cumbersome, making it unattractive to use for language designers. This still leaves us without a commonly agreed language for the specification of control flow semantics.

This paper presents a visual language, CFSL+, that aims to solve this problem. CFSL+ can be used to specify control flow in the same way as CFSL. Unlike CFSL however, the language was designed specifically with readability in mind. To achieve this, a set of design principles called the "Physics of Notations" [5] was used. These principles are based on theory and empirical evidence rather than common sense. Surprisingly, most software engineering (SE) notations are designed using only the latter [5]. Using them should result in notations that are more cognitively effective. The principles are also used to evaluate CFSL+.

CFSL+ is used only to specify control flow of statements, it cannot generate flow graphs of programs. However, control flow specifications made in CFSL+ can be compiled to CFSGs, after which GROOVE can generate flow graphs in the CFSL format. By specifying a new language that compiles to CFSL, rather than improving CFSL itself, we hope to increase the usability of CFSL, without compromising its correctness. A tool was made for both the specification of control flow in CFSL+, and for the compilation of those specifications to CFSL.

The remainder of the paper is structured as follows: section 2 gives a brief overview of how CFSL works, and the elements that are required to specify control flow in CFSL. Section 3 gives an insight in previous research on cognitive effectiveness of visual languages. Section 4 presents CFSL+ and the design rationale behind each visual element, as well as a preliminary evaluation of the language. Section 5 shows how CFSL+ diagrams are compiled to CFSGs, and section 6 concludes this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
24th *Twente Student Conference on IT* January 22st, 2016, Enschede, The Netherlands.
Copyright 2016, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

2. CFSL

This section will provide a quick overview of how CFSL works, and the various elements the language consists of. The basis of a control flow specification for a programming language is the abstract syntax graph (ASG) of that language, which is derived from the syntax grammar. For every rule in the abstract grammar, control flow has to be specified, which is done using control flow specification graphs (CFSGs). From these CFSGs graph transformation rules are generated, which can be used to transform any ASG into a flow graph. Thus, when CFSGs have been specified for every grammar rule in a programming language, flow graphs for any program in that language can be generated from the ASG of that program.

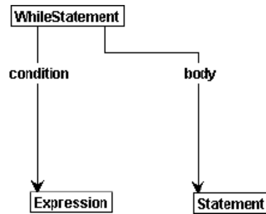

Figure 1. An abstract syntax graph of the `while` statement [7]

The actual value of CFSL is its ability to generate flow graphs (FG) after CFSGs have been specified for the statements in a programming language. Note however that this research does not aim to improve the readability of those FGs. Instead, the goal is to make it easier to specify and read CFSGs.

2.1 Control Flow Specification Graphs

Figure 2 shows the meta-model to which every CFSG should adhere. The remaining section will cover each of the elements defined in this model, and give examples for them using the CFSG for the `while` statement shown in figure 3.

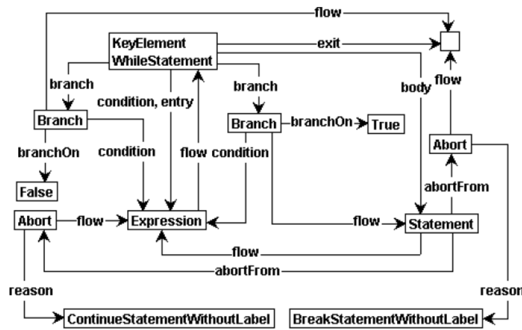

Figure 3. A control flow specification graph of the `while` statement [8]

2.1.1 *Abstract Syntax Graph*

At the core of every CFSG is the ASG of its grammar rule. The root of such an ASG is the left hand side of the rule, whereas its children are the non-terminals on the right hand side of the rule. For example, the grammar rule for Java's while-statement is as follows:

`WhileStatement = while ( Expression ) Statement.`

The corresponding ASG can be seen in figure 1. Note that the edges are labeled to denote the relation between the non-terminals. This label is optional. By default, this label is simply `child`. The same ASG can be found in figure 3

2.1.2 *Entry and exit*

To specify where control flow starts, an edge labeled `entry` is used, pointing towards the syntax element where control flow starts. Similarly, an edge labeled `exit` may point towards a node indicating that this node is the exit of a syntax element. No flow can originate from an exit node. These entry and exit nodes are used by CFSL to connect all the CFSGs for a single programming language. A CFSG can define only one entry, but multiple exits.

The `entry` edge in figure 3 indicates that control flow starts by evaluating the condition. There is only one `exit` for a `while` statement.

2.1.3 *Sequential control flow*

CFSL identifies three different kinds of control flow: sequential control flow, conditional branching control flow and abrupt completion control flow. Sequential control flow is specified by an edge labeled `flow` from one syntax element to another. Sequential flow does not branch, and as such a syntax element can not have more than one outgoing `flow` edge.

For example, the `flow` edge from `Statement` to `Expression` indicates that after the body of the `while` statement finished, control the condition is evaluated again.

2.1.4 *Conditional branching control flow*

Conditional branching is more complex, and requires additional nodes and edges. Each possible branch is denoted by an auxiliary node labeled `Branch`. This node has an incoming edge labeled `branch` which indicates the origin of the conditional branching, an outgoing `flow` edge towards the element where control flow is to be transferred to, and an outgoing edge labeled `condition` towards the element that results in the value that is to be evaluated for the branching operation. Furthermore, it may have an edge labeled `branchOn` towards a node that indicates the literal value the condition has to evaluate to for this branch to be taken, or a self-edge labeled `branchDefault` to indicate that this branch is taken when no other options are available.

Figure 3 shows the two branches that can be taken by `WhileStatement` after `Expression` has been evaluated. If it evaluates to `True`, control flows to the body of the `while` statement. If it evaluates to `False`, control flows to the exit.

2.1.5 *Abrupt completion control flow*

To specify abrupt completion an auxiliary node labeled `Abort` is required. This node has an edge labeled `reason` towards a syntax element that indicates the reason for the abrupt completion. CFSL identifies four different kinds of abrupt completion, each with different edges that are added to the `Abort` node:

- **Introduction**: an incoming edge labeled `abort`, and no outgoing `flow` edge. An example for this would be a `throw` or `return` statement in Java.
- **Resolution**: an incoming edge labeled `abortFrom`, and an outgoing `flow` edge. One example for this is a `catch` statement in Java.
- **Immediate resolution**: an incoming edge labeled `abort`, and an outgoing `flow` edge. An example for this is the `GoTo` statement in Visual Basic.

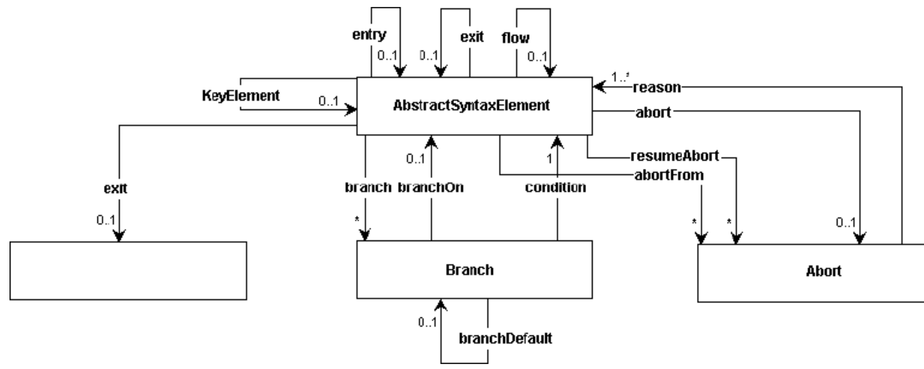

Figure 2. CFSL meta-model [7]

- **Resumption**: an incoming edge labeled `resumeAbort`. An example for this is when a `finally` block in Java resumes abrupt control flow after it has completed execution.

The `while` statement has two cases of control flow resolution, both originating in the body of the `while` statement. It resolves a `break` statement by passing control flow to the exit of `WhileStatement`. Abrupt control flow introduced by a `continue` statement is resolved by passing control flow to `Expression`.

2.1.6 KeyElement

CFSL defines a label `KeyElement` that can be set on an abstract syntax element to indicate that the CFSG specifies control flow for that element. Every CFSG must have 1 key element. A CFSG can contain more syntax elements than just the ones in the grammar rule corresponding to the key element. This can be useful to indicate a relation between control flow for the key element and other syntax elements.

Because figure 3 denotes control flow for the `while` statement, the `WhileStatement` syntax element is labeled `KeyElement`.

3. PHYSICS OF NOTATIONS

Although the use of visual notations is widespread in the field of software engineering (SE), little research is done on what makes them cognitively effective. When designing notations, most effort is generally spent on the constructs of the language, and only little on the visual representation of those constructs [5]. Even UML [6], the de facto industry standard for modeling in SE, does not provide any rationale for the design of the language. Instead, the language elements appear to be designed using best practices and common sense, rather than theory and empirical evidence. This is surprising, seeing that the manner in which information is represented highly affects the human ability to recognize that information [1].

In a reaction to the lack of theory on notation design, Moody presents a framework called "The Physics of Notations" in [5]. The framework comprises nine principles based on theory and empirical evidence that can aid with the design and evaluation of visual languages. This paper uses that framework to identify the shortcomings in CFSL, and to evaluate whether CFSL+ has reached its goal of being cognitive effective. This section will give a short explanation of each principle, as well as an evaluation of CFSL using that principle. It is important to note that some principles conflict with others. Therefore, the goal is not to get a maximum score on each principle (which would not be possible), but rather to find an optimal balance between the principles.

3.1 Semiotic Clarity

This principle states that there must be a one-to-one relation between semantic constructs and graphical symbols. This prevents ambiguity and unnecessary complexity.

CFSL scores poorly on this principle. Although there are many semantical constructs that CFSL visualises (such as sequential control flow, conditional branching, four types of abrupt completion, and so on), there are only two symbols: nodes and edges. This is called symbol overload: the same symbol is used to express multiple different constructs. Symbol overload is problematic because it can cause ambiguity. An example of ambiguity in CFSL is that a child edge between two nodes may be interpreted as control flow between two nodes.

3.2 Perceptual Discriminability

This principle states that different symbols should be easily distinguishable from one another. The extent to which symbols are different from each other is called the visual distance. This is defined as the amount of visual variables in which two symbols differ, and the extent to which these variables are different. Moody identifies eight visual variables: horizontal position, vertical position, shape, brightness, size, orientation, colour and texture. Symbols that have a unique value for one of the variables are discriminated more effectively than symbols that are only unique in a combination of variables [9]. For example, in a diagram where all symbols are black, a red symbol will appear to "pop out" of the diagram, quickly drawing the attention of the reader.

In principle, CFSL scores well on this principle. Despite the fact that the symbols only differ in shape, the two symbols (nodes and edges), are easily distinguished. However, two symbols are by no means sufficient, and if more symbols are to be added, the use of more visual variables may be required.

3.3 Semantic Transparency

This principle states that visual representations should be designed in such a way that, when possible, their representation suggests their meaning.

The only symbol that adheres to this principle in CFSL is the arrow when used for expressing control flow. Arrows are a very common symbol for expressing direction and as such even novices are likely to understand the meaning behind the symbol. However, when used for expressing parent-child relations in the syntax graph, it actually sug-

gests a meaning that it does not have, which is even worse than simply not knowing what it means. The latter is the case for syntax elements, which is expressed using rectangles. However, due to the highly abstract nature of the concept of syntax trees, there may not be a better representation for them.

3.4 Complexity Management

This principle states that visual notations should include explicit mechanisms to support complexity. This means that if a diagram becomes too complex to easily understand, it should be possible to decompose it into smaller parts.

CFSL supports complexity management by its very nature: rather than specify control flow for the entire programming language or even statements, control flow is specified for each grammar rule. This is however enforced, and not just optional. For statements that are defined by a large number of grammar rules this can in fact make matters more complex. Furthermore, it is not possible to decompose a CFSG of a grammar rule into smaller CFSGs in the case it becomes too complex.

3.5 Cognitive Integration

This principle states that explicit mechanisms should be included to support integration of information from different diagrams. It only applies when multiple diagrams are used to represent a system.

CFSL provides no visual clues to indicate a relation between graphs.

3.6 Visual Expressiveness

This principle states that symbols should use the full range of visual variables. One of the most effective of these variables is colour, because the human visual system is highly effective at discriminating between a limited number of colours [2]. However, colour is challenging for certain persons (the color blind) and printing in black and white only is still very common.

CFSL scores poorly on visual expressiveness, with only shape being used. Although arguably sufficient for a language with only two symbols, there is much room for improvement here.

3.7 Dual Coding

This principle states that text should be used to support graphics. Research has shown that the combination of graphics and text is more powerful than only text or graphics [3].

CFSL makes no use of dual coding, despite the abundance of text. Text does not support graphics, but is used as the only way to distinguish between different symbols.

3.8 Graphic Economy

This principle states that the number of different symbols should be cognitively manageable. When a diagram uses too many different symbols, it becomes hard for a reader to understand and remember all of them. According to [4], humans are able to discriminate between up to seven different symbols, plus or minus two.

With only two different symbols, CFSL passes this principle. This is a good example of a principle that conflicts with other principles (in this case Semiotic Clarity).

3.9 Cognitive Fit

The last principle states that a language should have different "dialects" for different audiences and media. For instance, novices may need a simplified version of the same language. There could be a dialect that uses simple shapes and little colour for hand drawn diagrams, and a more complex dialect that uses the full spectrum of colours for use on screen.

CFSL does not have different dialects.

4. CFSL+

This section will present CFSL+, the visual language that is the result of this research. First, the overall design rationale behind CFSL+ will be explained. Then, each of the elements in the language will be presented, along with the design rationale behind them. The section will conclude with an evaluation of CFSL+ using Moody's principles.

4.1 Design philosophy

CFSL+ has been specifically designed with cognitive effectiveness in mind. The main idea behind CFSL+ is that the user should first and foremost be able to identify the general control flow as fast as possible. All other information, such as the syntactic relation between elements, is secondary. This is perhaps where CFSL is lacking the most. All relations between elements are expressed by arrows, and all elements (be they abstract syntax elements or auxiliary branch nodes) are expressed by the same rectangles. Because of this, the reader has to put a relatively great deal of effort in identifying which edges are for control flow, and which are not. This makes CFSGs hard to read.

To aid the reader in identifying control flow, elements that denote control flow are designed in such a manner that they will appear to be more in the 'foreground', whereas secondary information will be placed in the 'background'. This is achieved by the use of color saturation and size. Smaller objects are generally perceived to be further in the background than bigger objects. Similarly, low color saturation has the effect of an object to appear in the background, whereas high color saturation will make an object appear to be in the foreground.

Another important aspect of CFSL+ is that it does not try to capture every semantical construct in a visual symbol. An example for this is the absence of a `reason` edge for abrupt control flow. Instead, the reason for abrupt control flow is only expressed using text. The reasoning behind this is that trying to express everything visually can make diagrams so complex that this becomes counter productive. Again, important aspects of control flow are all expressed visually, while the details are expressed using text only.

4.2 Language elements

This section will give an overview of all elements that are defined by CFSL+, as well as a design rationale for them.

4.2.1 Abstract Syntax Graph

Just like CFSL, the core of a specification diagram in CFSL+ is the abstract syntax graph. The syntax elements are, like in CFSL, expressed by rectangles with a black border, labeled with the name of the non-terminal. The key element of the graph is represented by a much bolder border, rather than by another label, as well as an underlined label. This should allow the reader to instantly identify that this element is more 'important' than the other elements. A syntax element may have a circle in the right top corner with a single letter, which is used to uniquely identify the syntax element. This identifier can be used for branching and abort flow, as will be explained

further on.

The relation between non-terminals in their ASG is expressed using a child edge. A child edge has a thin light grey stroke, and dark grey rectangles at each side. Grey was chosen because this way the edge appears to be farther in the background than the solid black flow edges.The rectangle at the parent side is larger than at the child side. This is a metaphor for the fact that parents are larger than children. The rectangles are also deliberately placed inside the syntax element, to indicate that a child is a part of a parent. A child edge is labeled 'child' by default, but may have a custom label like in CFSL. An example of the ASG for the `while` statement in CFSL+ is shown in figure 4. Note that `WhileStatement` is they key element, and the custom labels for the child edges.

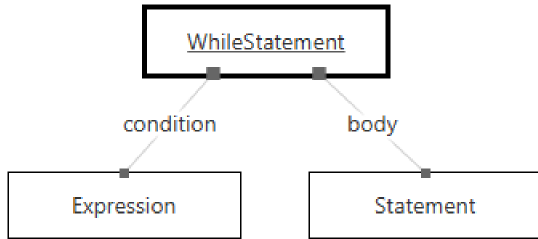

Figure 4. An abstract syntax graph of the `while` statement in CFSL+

4.2.2 *Sequential control flow*

A flow edge between two syntax elements indicates regular control flow between them. A flow edge is a bold, blue arrow, pointing in the direction of the flow. Arrows intuitively represent direction making the shape of the symbol an obvious choice. The thickness and colour of the symbol help it appear in the foreground. Flow edges are labeled 'flow'.

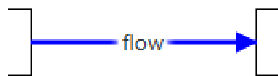

Figure 5. A flow edge.

4.2.3 *Branching control flow*

Conditional branching is indicated by a diamond shaped node. This branch node, has an incoming flow edge labeled 'branch' to indicate the point of branching. The syntax element that results in the actual value that is evaluated for the branching operation is referred to by the label of the branch node. This label has the format `[id] = ?` where `[id]` identifies a syntax element as explained in the previous section.

The actual branches that can be taken are expressed with outgoing flow edges from the branch node. The label of the edge indicates the value that the condition will have to evaluate to for that branch to be taken. There are three kind of labels:

- A label can be the name of the literal value that the condition should evaluate to, such as 'True' or 'False'.
- A label can be the identifier of another syntax element defined in the diagram. If the condition evaluates to the same value as that syntax element, this branch will be taken.
- A label can be 'default'. This indicates that this branch is taken if no other branches can be taken.

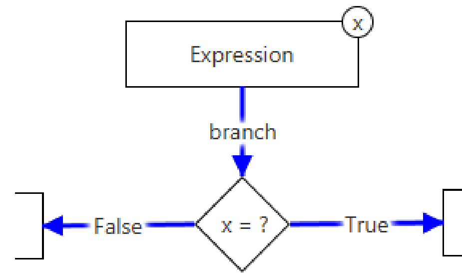

Figure 6. Conditional branching.

Using only text to indicate the condition and values for the branching prevents the diagram from becoming overly complex. The diamond shape for the branch node has been chosen because it is a common shape for branching operations in SE notations, with the most well known example being UML flow diagrams. It also looks sufficiently different from the rectangular syntax elements so that the two are easily distinguished.

4.2.4 *Abrupt control flow*

All kinds of abrupt control flow are indicated by a red arrow, with a red thunderbolt icon at the start of the edge. The thunderbolt icon and the red colour were chosen because in many cultures this is interpreted as something going wrong. Of course, abrupt control flow does not always imply error handling, but because it often does it may still be a useful metaphor. It is also sufficiently different from sequential and branching control flow, to emphasize the fact that abrupt control flow describe alternate routes of flow. Finally, red makes the symbol pop out, as blue makes the other flow edges pop out.

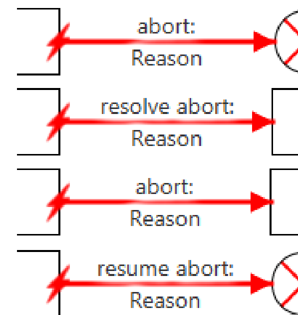

Figure 7. From top to bottom: abrupt control flow introduction, resolution, immediate resolution and resumption.

Like in CFSL, there are four kinds of abrupt control flow. Abrupt control flow introduction is specified by an edge labeled 'abort' flowing from a syntax element to an abort state (which will be introduced in the next section). Resolution is specified by a 'resolve abort' edge flowing to a syntax element where control is transferred to. Similar to CFSL, immediate resolution is an 'abort' edge flowing to a syntax element rather than an abort state. Lastly, resumption is modelled by an edge labeled 'resume abort' flowing to an abort state.

It is possible to give multiple reasons for control flow, in which case the reasons are to be separated by comma's. It is even possible to specify an entire abstract syntax tree as the reason, rather than just one element. The downside of

this is that labels can become quite complex. However, the majority of the times only one syntax element is the reason for abrupt control flow, and expressing it visually may pollute the diagram to such an extent that the diagram as a whole becomes harder to read.

4.2.5 *Start, stop and abort nodes*

The start, stop and abort nodes are used to specify where the control flow starts and ends. Only one start node can appear in the diagram. This node has a single flow edge leading to an abstract syntax element. Multiple stop and abort nodes are possible in a diagram. Stop nodes can have any number of any kind of control flow incoming. No control flow can originate from a stop node. Abort nodes can have any number of incoming abort and resume abort edges. No control flow can originate from the abort node. Control flow that ends in an abort node indicates that control flow will not pass to the next statement, but will have to be resolved by another statement.

The start, stop and abort nodes are all circles containing an icon. A circle was chosen to make it sufficiently different from other nodes in the graph. The start and stop nodes have play and stop icons as commonly used in media players. This is a metaphor for the start and end of control flow. The icons are black, because they denote regular control flow. The abort node has a red cross. Red was chosen to link it to the abort edges.

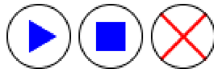

Figure 8. From left to right: start, stop and abort nodes

4.2.6 *Syntax element exit*

When flow enters a stop node, it is by default assumed to take the exit of the key element. In some cases it can be desirable to change this behavior. The exit edge connects a stop node and a syntax element to indicate that any flow going to that stop node takes the exit of that syntax element.

Like the child edge, the exit edge has a light grey stroke, because it is considered secondary information. To emphasize the fact that the relation between a syntax element and a stop node is more abstract than a parent-child relation, the line is also dashed rather than solid.

4.2.7 *Example diagram*

Figure 9 shows an example diagram for the `while` statement. Note that only 6 nodes are required to specify control flow for the `while` statement in CFSL+, as opposed to 12 nodes in CFSL. By focussing on the solid blue arrows, one can quickly identify the looping behavior of the `while` statement. This is not as easy to see in CFSL, where all edges are visually the same.

4.3 Evaluation

To determine whether CFSL+ is cognitively effective, it was validated using Moody's principles. For each principle CFSL+ is first evaluated, and then compared to CFSL.

4.3.1 *Semiotic Clarity*

Out of the 9 elements that CFSL+ has, there is one (11%) case of symbol overload: all three types of abrupt control flow (introduction, resolution and resumption) are represented by the same symbol. There are also three cases of symbol deficit: there are no equivalent symbols for the `condition`, `branchOn`, and `reason` edges in CFSL.

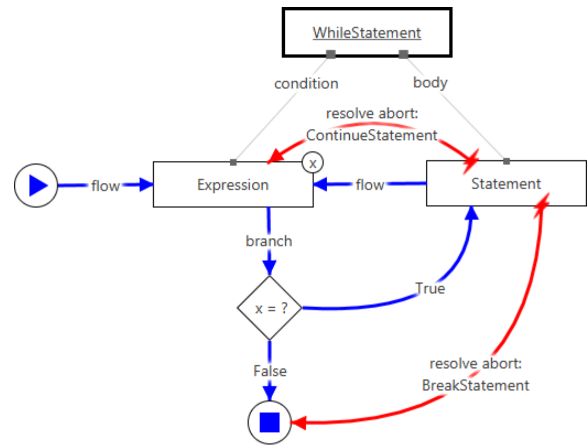

Figure 9. Specification diagram for the `while` statement

Symbol deficit is considered to be desirable, to reduce complexity and thus score better on the Graphic Economy principle. Although symbol overload is generally considered less desirable, in this case it could be considered as a special case of symbol deficit. CFSL+ simply does not define symbols for types of abrupt control flow, but just control flow in general. Again, this is done to reduce complexity.

This is a clear improvement to CFSL, where all symbols are overloaded.

4.3.2 *Perceptual Discriminability*

Attention has been paid to make sure all elements can be easily distinguished. Nodes only differ in one or two visual variables (shape and border width), but the difference between a circle and a rectangle or diamond is fairly big. The difference between the rectangle and diamond shape is a bit smaller however. Nodes (rectangles, diamonds and circles) and edges (arrows and lines) have an even larger visual distance. Although abort and flow edges are both in essence arrows, their color and the extra thunderbolt shape for abort edges make them considerably different. To make sure the child and exit edges are not misinterpreted as control flow information, they are simply lines rather than arrows.

Because CFSL only has two symbols it is hard to make a real comparison with CFSL+. However, because there are far less cases of symbol overload, it is much easier to distinguish between different constructs in CFSL+.

4.3.3 *Semantic Transparency*

Control flow in programming languages is a rather abstract concept, and as such not all elements in CFSL+ are semantically transparent. The best example is the syntax element, which is represented by a rectangle. A rectangle could in essence mean anything, and a total stranger to CFSL+ would not likely understand that it represents a non-terminal. The same can be said about the diamond shape for the branch node, and the circle with the cross for the abort node.

Other elements are more transparent however. Start and stop nodes have clear start and stop icons, and control flow is denoted by arrows, which are very common icons to represent direction. Exit and child edges connect elements to denote a relation between them, although the exact relation between them is likely not as easily guessed without prior knowledge.

Again, CFSL+ scores better on this principle. Not all symbols in CFSL+ are semantically transparent, but in contrast to CFSL there are no symbols that suggest a different meaning than the intended meaning. Recall that CFSL uses the arrow symbol to represent a parent-child relationship.

4.3.4 Complexity Management

CFSL+ is modular, just like CFSL. Control flow is not specified for the entire programming language in one diagram, but rather one diagram for each grammar rule. In the case that a diagram for one rule becomes too complex there are no options to break it up in more diagrams.

Because CFSL+ uses the same kind of modularity as CFSL there is no difference with CFSL on this principle.

4.3.5 Cognitive Integration

CFSL+ does not provide additional visual clues to integrate different diagrams. Again, there is no difference with CFSL.

4.3.6 Visual Expressiveness

A total of four visual variables are used in CFSL+: shape, colour, brightness (colour saturation) and texture. All nodes only use on visual variable however, which is shape. Brightness is used to create visual precedence between different types of edges. The bold lines of control flow arrows make them appear more important than the thin, grey lines of child and exit edges. Texture is only used for the dashed lines of the exit edges. Colour is used to distinguish the control flow edges from the other symbols.

CFSL+ scores much better than CFSL on this principle. The only variable used by CFSL+ is shape, whereas CFSL+ also uses colour, brightness and texture.

4.3.7 Dual Coding

Dual coding means that text is used to supplement graphics. This is used in the case of flow, branch, exit and child edges. The other texts (such as the name of a syntax element or the reason for an abort flow) do not complement graphics, but rather replace them. Although more dual coding would be possible by adding text to for instance the start and stop nodes, it could also clutter the already text-heavy diagram too much.

CFSL+ scores slightly better than CFSL, due to the fact that in CFSL the only way to distinguish between symbols is by text (which gives text a deciding rather than supporting role), whereas CFSL+ has dedicated symbols for structures.

4.3.8 Graphic Economy

Although there are significantly more symbols in CFSL+ than in CFSL, it is still relatively low with nine elements. These nine elements can be classified into five different categories:

- Syntax elements (syntax element)
- Branch nodes (branch node)
- State nodes (start node, stop node, abort node)
- Flow edges (flow edge, abort edge)
- Relation edges (child edge, exit edge)

CFSL+ scores lower than CFSL in this regard, since CFSL has only two symbols to remember. This is however unavoidable if we are to score better on Semiotic Clarity.

4.3.9 Cognitive Fit

CFSL+ does not define different dialects for different audiences or media. The reasoning behind this is that CFSL+ is not intended for a wide variety of audiences. Also, CFSL+ diagrams are still readable when printed black and white, and the shapes are simple enough for them to be hand drawn. Therefore no additional dialects seem to be relevant for other media.

CFSL+ scores the same as CFSL on this principle, as both languages do not feature different dialects.

4.3.10 Discussion

Table 4.3.10 summarizes the results of the evaluation. Evaluating CFSL+ and CFSL using Moody's principles suggests that CFSL+ is cognitively more effective than CFSL. However, because this is a strictly qualitative evaluation there is no way to say by how much it is more effective. To get a more decisive answer to this question, a user evaluation is required to measure how much easier it is to understand diagrams in CFSL+ than it is to understand CFSGs. Still, it is safe to say that CFSL+ is at least an improvement over CFSL with regards to readability.

Table 1. CFSL compared to CFSL+ using Moody's principles

Language	1	2	3	4	5	6	7	8	9
CFSL	--	0	-	-	-	--	-	++	0
CFSL+	+	+	+	-	-	+	0	+	0

5. SUPPORT FOR CFSL+

To support the use of CFSL+, a tool was developed that allows users to draw diagrams in CFSL+, and export them to CFSGs in CADP automata format, which can be imported by GROOVE. This section gives an overview of the structure of the program, as well as how it was validated that it correctly compiles CFSL+ diagrams to CFSGs.

5.1 Technology

The tool is implemented using Java, in order to make it available for all major PC platforms. For the interface and the binding between model and interface JavaFX is used. JavaFX is Java's latest framework for developing interfaces, and allows for a separation between application logic, layout and styling by using Java, FXML and CSS respectively. FXML is an XML-based language in which the structure of interfaces is defined. CSS as used in JavaFX is similar to CSS used on the web, although somewhat limited.

5.2 Interface

The tool follows the Model-View-Controller (MVC) pattern to separate application logic from presentation logic. This is implemented by a single abstract class `Controller<T extends Node>`. This class abstracts much of the boilerplate code away that is needed for loading views using FXML files. Note that `Node` refers to a JavaFX node, not a node in a CFSL or CFSL+ graph. `Controller` has three ways of loading and attaching to a view:

1. If no arguments are provided to the constructor, a default FXML file is loaded, and the `Controller` is specified as the controller object for that FXML file. The FXML file that is loaded must have the same name as the `Controller` implementation, minus the word controller, and then suffixed with `View.fxml`.

For example, calling the default constructor of `MainController` will load the FXML file named `MainView.fxml`. This method requires that the name of the controller ends with `Controller`.

2. If a string is provided as argument, `Controller` will load an FXML by that string instead. This can be used to share the same FXML file with multiple controllers.

3. If a JavaFX `Node` is provided as argument, `Controller` will not load an FXML file, but attach to the provided `Node` instead. This is useful in cases where views are more naturally defined using Java than FXML.

All methods of loading views require that the view is of type `T` as specified by the `Controller` implementation. It is up to implementations of `Controller` to implement the actual binding to views and models. Generally, the model will be injected in the constructor of the controller, which loads the view as described above.

Controllers and views defined using Java code reside in the `gui` package. Views defined using FXML reside in a separate resources folder. This folder also contains the CSS stylesheet that defines the styling for the entire application.

5.3 Model

The Model part of MVC is implemented by classes in the `model` package. The core classes are `Graph` and the abstract `GraphElement`. `Graph` is a collection of `GraphElements`. `GraphElement` is implemented by the abstract classes `Node` and `Edge`. `Node` has methods for connecting and disconnecting `Edges` to itself. It also has an abstract method `canConnect` that is used to determine whether a specific side (start or end) of a given `Edge` can connect with the given `Node`. This is used to maintain the model in a valid state, as well as provide feedback to the user when attempting to connect an edge to a node.

For each symbol in CFSL+ an implementation of either `Node` or `Edge` exists. Figure 10 shows these classes, as well as the relation between them. The boxes are implementation of `Node`, and the arrows are implementations of `Edge`. The direction of the arrows represents the direction in which they can be connected. The name of each class should be self-explanatory.

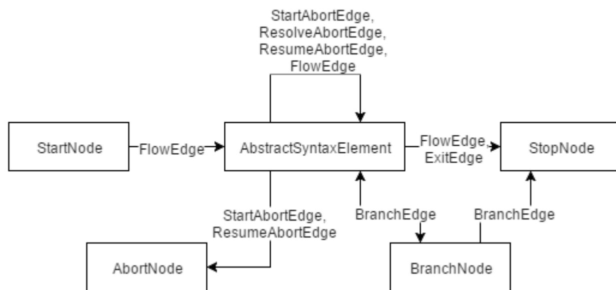

Figure 10. Model classes

Model classes generally use JavaFX properties to allow for data binding between controller and model object.

5.4 Compilation

Compilation of CFSL+ diagrams to CFSGs is implemented by the `CfslPlusChecker` and `CfslPlusCompiler` object. `CfslPlusChecker` is injected with a `Graph` object, and is responsible for checking whether the provided graph is in a valid state for compilation. If any of the checks fail, an exception is thrown. It also determines the type of each `Node` and `Edge` and copies it to a list of `Nodes` and `Edges` of that type, which is later used by `CfslPlusCompiler`.

`CfslPlusCompiler` performs the actual compilation operation. Because a specification diagram in CFSL is a graph, and generally not a tree, it cannot be walked in the same way that abstract syntax trees usually are in compilers of programming languages. Instead, the graph is walked in order of type. First, CFSL nodes are created for each `Node` object, and the mapping between source and target node is saved. Then, CFSL edges between those nodes are created for each `Edge` object.

5.5 Validation

This section briefly describes how it was validated that the compilation process indeed results in the intended CFSGs. Example CFSGs are given in [7] of most of the Java statements. By modeling those statements in CFSL+ and compiling them to CFSGs, the CFSGs can be compared to see whether they are equal.

Not all CFSGs defined in [7] were selected. Instead, a set of CFSGs was carefully selected so that each feature of CFSL was tested at least once. These features include all types of control flow, and features such as multiple exits for a single CFSG, among others. The equivalent CFSL+ statements proved to generate CFSGs that are equal to the CFSGs that were chosen for the test. Appending A shows two of the CFSGs that were chosen for the test, as well as the equivalent CFSL+ graph and the result of compiling it.

6. CONCLUSION

This paper presented a new visual language, CFSL+, for the specification of control flow in programming languages. Although visual languages are common in SE, few of them have been designed with cognitive effectiveness in mind. To achieve a cognitively effective language, CFSL+ has been designed and evaluated using Moody's principles. Although an evaluation using those principles alone is not enough, it does suggest that CFSL+ is at least on the right track of being cognitively effective, as opposed to CFSL. We have also presented how CFSL can be compiled to CFSL+, and validated that indeed everything that can be modeled using CFSL can be modeled using CFSL+.

One of the most valuable lessons learned is the importance of perceptual precedence. Like CFSL, CFSL+ contains a relatively large number of edges that do not directly represent control flow. If all those edges would appear to be equally important, it would be hard for the reader to identify the control flow in a diagram. By making control flow edges appear to "pop out" of the diagram, attention is automatically drawn to control flow information first, without having to compromise on the amount of secondary information shown. This does not just apply to control flow diagrams, but to any notation that needs to include a high variety of information.

As suggested earlier, there is still remaining work. A user evaluation is required to verify whether it is indeed more effective at communicating control flow information, as well as to identify potential points of improvement. Also, no work has yet been done on improving Complexity Management and Cognitive Integration. Improving on those principles could improve readability for specifications of statements that consist of a large number of grammar rules.

7. REFERENCES

[1] J. Larkin and H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987.

[2] J. Mackinlay. Automating the design of graphical presentations of relational information. volume 5, pages 110–141, 1986.

[3] R. Mayer and R. Anderson. Animations need narrations: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology*, 83(4):484–490, 1991.

[4] G. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.

[5] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

[6] Object Management Group. *Unified Modeling Language*, 3 2015. Version 2.5.

[7] R. Smelik. Specification and construction of control flow semantics. Master's thesis, University of Twente, 1 2006.

[8] R. Smelik, A. Rensink, and H. Kastenberg. Specification and construction of control flow semantics. *Proceedings - IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2006*, pages 65–72, 2006.

[9] A. Treisman and G. Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12(1):97–136, 1980.

APPENDIX

A. COMPILATIONS FROM CFSL+ TO CFSL

This section shows two test cases that were ran for the validation of the compilation process. Figure 11 shows the `LabeledStatement` and figure 12 shows the `TryFinally`. The first graphic in each figure shows the CFSL diagram as presented in [7]. The second graphic shows the CFSL+ equivalent, and the last graphic shows the result of compiling the CFSL+ diagram. It can be seen that both CFSL diagrams are equivalent.

Note that the visual appearance of the CFSL+ diagrams is slightly different than presented in the rest of the paper, because some changes were made after validating the compilation process. However, those changes did not affect the semantics of CFSL+.

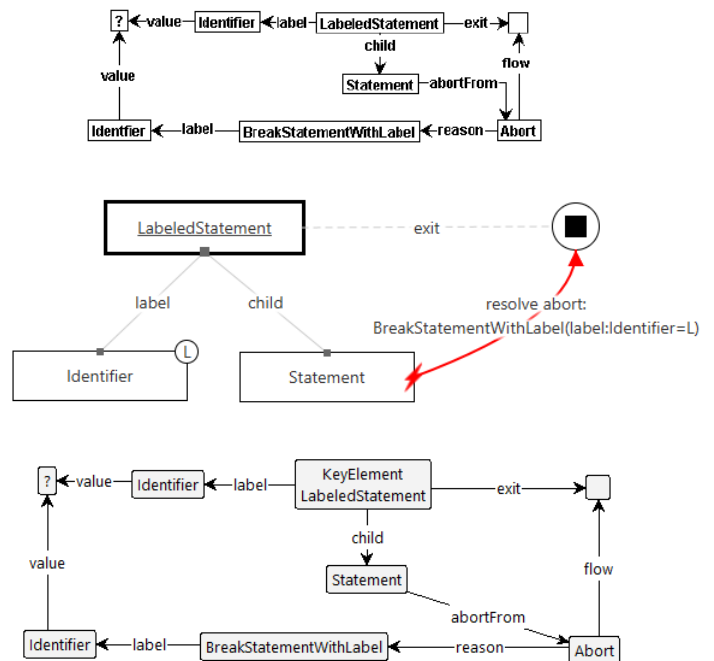

Figure 11. Compiling the `LabeledStatement` diagram. The original CFSL diagram is erroneously missing the `KeyElement` label, but is otherwise correct.

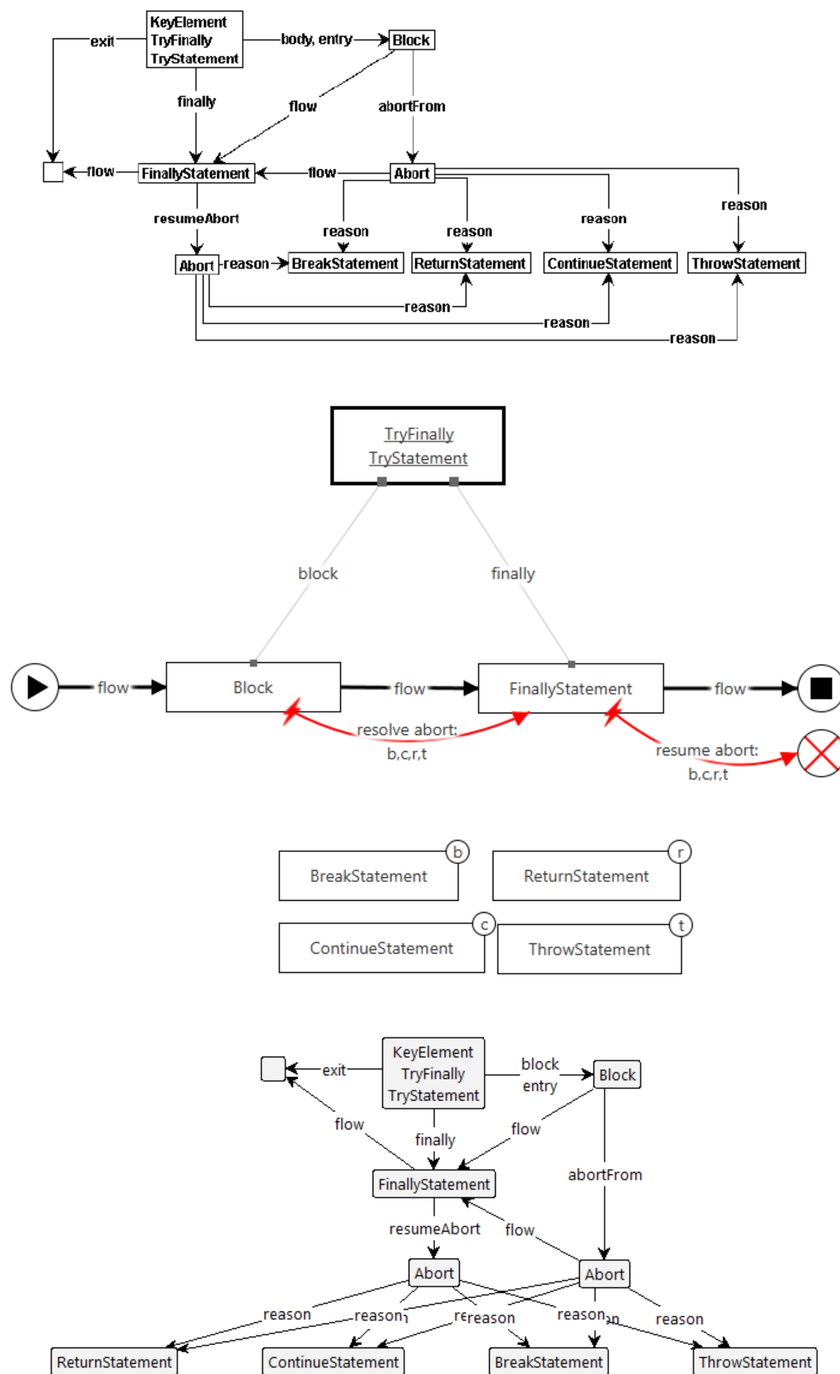

Figure 12. Compiling the `TryFinally` diagram.