Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

2 Cδ programming language: specification and goals

In this section, the desired feature set and design choices of Cδ will be discussed. Cδ is a programming language ~~which~~ that extends the C♯ language and transcompiles into it. This means that (unlike C♯) it does not compile into the *Common Intermediate Language (CIL)* [1] but rather into C♯ code, ~~T~~thus passing the responsibility to compile into the *CIL*.

For the naming of this language, the character *C* of C♯ has been kept. The *δ* (lowercase delta from the Greek alphabet) is a reference to the mathematical model of *deterministic finite-state machines*. In the quintuple $(\Sigma, S, s_0, \delta, F)$ the *δ* stands for the *state-transition function* ~~which~~ that decides the next state for the given current state and input. This hints at the language paradigm that Cδ supports. In places where the Unicode character *δ* is not supported, the alternative notation *C delta* is used instead.

While this section shows what Cδ *should* have been, the section 6.2 Design goals vs. ~~final result~~results shows what it *has* become. ~~In s~~Section 3 Cδ programming language: implementation describes the developmental work of Cδ ~~is described~~.

2.1 Features

This subsection names and explains the planned feature set of the Cδ transcompiler.

2.1.1 Finite-state machines

The main motivation for Cδ is offering programming language constructs to create *finite-state machines*. Instead of writing highly complex

[1] An object-oriented assembly language into which .NET languages are compiled. This language will be compiled into executable machine code in the final step. Its purpose is similar to that of *Java bytecode*.

Commented [V1]: Active voice would be: This section provides a discussion of ….

Commented [V2]: Earlier, you used bolding for emphasis; here you use italics. Please check your style guide(s) and be consistent.

Commented [V3]: Please check your style guide(s) for instructions on whether to capitalise *section* when it refers to a specific section.

Commented [V4]: ***Final** result* is redundant.

Formatted: Font color: Auto

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

implementations, the developer can define state machines in Cδ by naming the available states and linking these with transitions. It is the Cδ transcompiler's responsibility to create valid C♯ code for the given state machine definition.

Determinism in Cδ

A *finite-state machine* defined in Cδ must follow the rule $\delta \rightarrow Q \times \Sigma \rightarrow Q$. This means that every transition δ (in the context of current state $q \in Q$ and current input $a \in \Sigma$) must have one single target state $q \in Q$. Even so, the *finite-state machines* in Cδ are **not** considered *deterministic*.

There are two reasons why *determinism* cannot be promised with Cδ:

1. **Single target state rule is not enforced**

   Defining a transition in Cδ allows one single source state and target state for a given condition only. However, there are no safety checks that multiple transitions are defining the **same condition** and **same source state** but with **different target states**.

   At *compile-time*[2] it is not possible to parse conditions and check ~~if~~ whether they are equal. That is because there are unlimited possibilities to write two conditions with different implementations ~~which~~ that evaluate to the same result. For example, these two Cδ condition blocks **condition** { **return true**; } and **condition** { **return** getTrueValue(); } might evaluate to the same but cannot be detected at *compile-time*.

[2] This describes the timespan in which the compiler translates the code and does static analysis in the search for issues.

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

The translated Cδ code does not check for it And at *runtime-time*[3] the translated Cδ code does not check for it either. Ideally, the state machine would iterate through each available transition and check for their conditions. When multiple transitions evaluate to be true, then an error should halt the program as the determinism rule was broken. Cδ will not check for this at *runtime-time* and choose the first transition elevating to true instead. This decision for *lazyevaluation* was made because each evaluation costs computing time. And in cConsiderationing between *formal definition check* and *performance*, the latter was preferred. Future versions of the Cδ transcompiler might change this behaviour.

Commented [V5]: Does this change your meaning?

2. **Programming can cause non-deterministic results.** While a developer would try to write code for deterministic results, there are multiple ways to break this behaviour. C♯ allows the developer to write *concurrent code* but Cδ is not implemented to be *thread-safe*[4] yet. *Reflection*[5] can be used to break into the private fields and methods of the state machine and thus circumvent safety checks.

Commented [V6]: You formatted this differently from #1 above. Please choose and be consistent.

In summary, the *finite-state machines* in Cδ cannot be considered *deterministic*. But they are not called *non-deterministic*, either, because this implies multiple target states would be allowed,. And thiswhich is prohibited in Cδ by definition (even if not enforced).
For this reason, the *finite-state machines* in Cδ remain untyped and mentioning the *determinism* is avoided.

---

3 This dDescribes the timespan in which the translated programme is running.

4 A C♯ component is considered *thread-safe* when multiple threads can access this component without causing *race conditions*.[Mic16b]

5 GThis gives C♯ code the ability to examine and modify its own code at runtime-time.[Mic15]

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

State machine example and Cδ pseudocode

To showcase the language extensions introduced by Cδ, we consider the following example: we want to write a console application ~~which~~ that checks ~~if~~ whether a given sequence of characters is in **lower camel case** notation.

~~For the purpose of~~For this example, we define *camel case* as a notation in which multiple words are concatenated without blank spaces in between. Each word in the characters sequence must start with a capital letter. No two capital letters can be next to each other and the last character ~~has to~~must be lower case. An empty sequence is not accepted.

The **lower** *camel case* uses the above rules with the distinction~~,~~ that the first word[7] in the sequence must start with a~~n~~ lower character. To further simplify the example, we only work with the Latin alphabet and do not accept any other characters (e.g., digits). These are accepted sequences:

- cat
- dogOwner
- lowerCamelCase

The following sequences are **not** accepted:

- ϵ[8]
- Cat
- CamelCase
- favoriteDVD[9]

Commented [V7]: I can't put a comment in the footnotes. Here and in several places, you refer to C]. Do you mean C#? Please check footnotes throughout and correct if needed.

For the detection of *lower camel case*, this *regular expression* [Aho90] can be used:

[a-z]([A-Z]?[a-z])*

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

~~And t~~This *regular expression* could be easily implemented in C♯ using the Regex[Mic16a] class included in the *Base Class Library*. But this approach will only return true if a match in the given sequence was found or not. The developer cannot intervene with the reading process and, e.g., in what context or ~~/~~at which character the matching failed.

Commented [V8]: Does this change your meaning? https://www.scribbr.com/academic-writing/transition-words-phrases-list-misuses/

Regular expressions are notations for *regular languages*. The *Chomsky hierarchy* classifies this as a *type-3 grammar*.[Cho59] Type-3 grammars can be

[7]In this context, *word* means a segment in the sequence. E.g., goodExample consists of the words good and Example.

[8]The small epsilon denotes an empty sequence. There are no characters in this sequence.

[9]Note that the suffix DVD ~~are~~ is three capital letters in a row. This is not accepted.

Commented [V9]: Shouldn't these be footnotes?

detected with finite-state machines (both *deterministic* and *non-deterministic*). This is the where the Cδ language will be used for this example.

The following *state machine diagram* (Figure 1) detects the *lower camel case* notation and is equivalent to the *regular expression* from above. A given sequence is accepted~~,~~ when the state machine halts in a *final state* (which is Lower Char in this machine). Halting in a normal *state* will reject the given sequence. If none of the available *transitions* can be traversed, ~~then~~ we consider the sequence rejected.

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

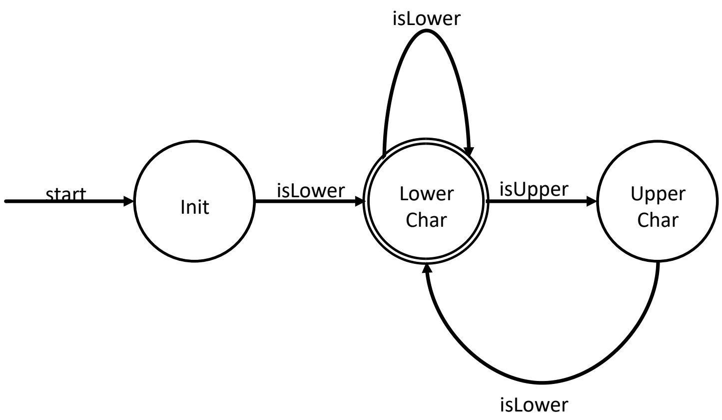

Figure 1: A finite-state machine detecting strings in lower camel case.

With the first state, Init, ~~it is enforced that~~ an empty sequence will not be accepted. The transition labels isLower and isUpper represent either *lower case* or *capital* characters. The states Lower Char and Upper Char ensure the other rules stated before.

Commented [V10]: Does this change your meaning?

Commented [V11]: Are what?

This example state machine will now be defined in the Cδ language. The following code is a mix of Cδ code and C♯ pseudo-code. The latter is done to focus on the new Cδ language features and to reduce the overall needed space in this thesis.

Basic finite-state machine definition in Cδ

```
public automaton<char> LowerCamelCaseMachine
{
```

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

```
// available states
start state Init; state
UpperChar;
end state LowerChar;

// available transitions
transition Init LowerChar
   { return char.IsLower(value); } transition
LowerChar ~~LowerChar~~
   { return char.IsLower(value); } transition
LowerChar UpperChar
   { return char.IsUpper(value); } transition
UpperChar LowerChar
   { return char.IsLower(value); }
}
```

Listing 1: C*δ* pseudocode for detection of lower camel case.

In the first line, an optional *access modifier*[6] (public) is used. It is followed by the new C*δ* keyword automaton. This tells the C*δ* transcompiler that the next code block, which is marked with ~~the~~ curly brackets { and }, defines a finite-state machine. A C♯ compiler would fail at this point and state~~,~~ that automaton is not a valid keyword.

After automaton an optional *data type* (char) can be given. This tells the C*δ* transcompiler which *data type* will be used to match a *transition*. ~~By u~~Using the data type char in this example~~,~~ ~~it is~~ ensure~~d~~s that only characters can be passed to this state machine for execution. Omitting the *data type* tells the C*δ* transcompiler to use object instead, which forces the transitions to type-check themselves.

[6] C*J* uses the object-oriented programming paradigm and thus has the concept of *encapsulation*. State machines defined in C*δ* are translated into normal classes in C*J*. The *access modifiers* in C*δ* work exactly ~~like~~ as they do in C*J*.

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

This feature resembles the *input alphabet* for state machines, transferred and applied to *data types* in this programming language.
LowerCamelCaseMachine gives the C𝛿 state machine a name, just ~~like~~ <u>as</u> classes are named in C♯.
In line<u>s</u> 3 and 8<u>,</u> the C♯ syntax for source code comments ~~are~~ <u>is</u> used. These are ignored by the C𝛿 transcompiler and used to group the next statements visually.

Commented [V12]: Generally, a paragraph should be more than one sentence.

Lines 4-6 define the available states in this state machine. A state in C𝛿 is defined by the keyword state, followed by a name and a semicolon. State machines in C𝛿 must have exactly one *initial state* but can have any number of *final states* (even zero). It is possible to define a state being both *initial* and *final*. Each state must be named uniquely and only a finite count of states can be defined.
In this example<u>,</u> the initial state Init, the final state LowerChar and the state UpperChar are defined. C𝛿 introduces the keywords start and end[7] to qualify the type of a state.

With lines 9-16<u>,</u> the available transitions are defined in this state machine. A transition is defined with the keyword transition, a source state, a target state and a code block. The source/target state must match ~~with~~ the defined states. The order in which states and transitions are defined does not matter. Transitions ~~can~~ <u>may</u> reference states that are defined in subsequent source code lines. The code block must return a ~~*boolean*~~ <u>*Boolean*,</u> which decides~~,~~ ~~if~~ <u>whether</u> the transition will be used for the given input. The input is a parameter named value and has the *data type* ~~which~~ <u>that</u> was defined in line 1. The rules for the transition code block are the same as for methods in C♯.

Commented [V13]: See comment about / above—here and throughout.

[7] The keyword end was chosen as an easy to remember counterpart to start. It qualifies the following state as a *final state* (also called *accepting state*)<u>,</u>~~.~~ ~~N~~<u>n</u>ot to be confused with the state on which the state machine *ends* (which can be final or not).

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

In this example, the four transitions are defined ~~just like~~as they are in the state machine diagram (Figure 1). The statements char.IsLower and char.IsUpper are C♯ pseudo-code, which takes a single character and return whether it is lowercase or ~~/~~capitalis~~z~~ed.

The above source code (Listing 1) is enough to create a state machine ~~which~~ that checks for *lower camel case* notation. Before showing the usage of this state machine, however, more powerful constructs will be introduced. Next, the definition of code blocks for entering and leaving states will be shown.

Entry and exit code blocks for states in Cδ

```
public automaton<char> LowerCamelCaseMachine
{
    //   a v a i l a b l e     s t a t e s
    start state Init
    {
        entry { WriteLine("State machine started"); }
        exit { WriteLine("First lower-case char read"); }
    } state UpperChar
    {
                    entry { WriteLine("Last char was lower -case"); }
    }
    end state LowerChar
    {
```

1

2

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

```
        entry { WriteLine("Last char was                    lower_-case"); }
    }

    //   a v a i l a b l e     t r a n s i t i o n s transition
    Init LowerChar
        { return       char.IsLower(value); }

      transition       LowerChar LowerChar

        { return       char.IsLower(value); }

      transition       LowerChar UpperChar

        { return       char.IsUpper(value); }

      transition       UpperChar LowerChar

        { return       char.IsLower(value); }
}
```

Listing 2: Extended Cδ pseudocode for executed statements when entering or leaving states.

This time, the definition of states has been extended by code blocks. ~~The~~ ~~l~~Lines 4-16 still define the same states as in the previous code (Listing 1). Now a state in Cδ can be followed by curly brackets (the semicolon is omitted). Within these brackets an *entry* code block, an *exit* code block or *both* can be defined. The keywords entry and exit define the type of its following C♯ code block.

The state Init has both a code block for when this state is entered and ~~one~~ for when it is exited. WriteLine is pseudo method to print output into the console application. It is used here to help the user trace ~~in~~ which state the state machine is currently in. Any other C♯ code could be put in here as well. The states UpperChar and LowerChar define an entry code block only.

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

In Cδ, states can be defined with optional entry and exit code blocks. There can only be one entry and one exit block **or** only one of them **or** neither of them. The next step is to allow code execution when transitions are traversed.

Traversal code blocks for transitions in Cδ

```
public automaton<char> LowerCamelCaseMachine
{
   //   available   states
   start state Init
   {
      entry { WriteLine("State machine started"); }
      exit { WriteLine("First lower-case char read"); }
   } state UpperChar
   {
```

1

2

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

```
                entry { WriteLine("Last char was upper case case"); }
    }
    end state LowerChar
    {
                entry { WriteLine("Last char was lower -case"); }
    }

    //    a v a i l a b l e    t r a n s i t i o n s transition
    Init LowerChar
       {  return  char.IsLower(value);  }  transition  LowerChar
    LowerChar
       {  return  char.IsLower(value);  }  transition  LowerChar
    UpperChar
    {
       condition { return char.IsUpper(value); } entry      {
       WriteLine("Read upper-case char"); }
    }
    transition UpperChar LowerChar
    {
       condition { return char.IsLower(value); } entry      {
       WriteLine("Read lower -case char"); }
    }
}
```

Listing 3: Further extended Cδ pseudocode for executing statements when traversing transitions.

Code execution can also be done while traversing transitions~~, too~~. Cδ allows ~~to~~ extending a transition with a code block ~~which~~ that is executed whenever the state machine decides to use this transition. In this example, the user will be informed which kind of character was read when switching between the states LowerChar and UpperChar.

Commented [V14]: Here and throughout, check placement for your captions.

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

To differentiate between the *traverse condition* code block and the *statements to execute when traversed* code block, the keywords condition and entry are used. The order in which the statements are executed is as follows:

1. exit code block of the **current state**
2. entry code block of the **traversed transition**
3. entry code block of the **reached state**

Commented [V15]: Good use of a numbered list. The rule for lists is to use a numbered list when the order of the list items matters and use a bulleted list when the order does not matter. Good job.

Using the state machine in a console application

Now the defined finite-state machine will be used in a sample console application. The idea is to write C# code ~~which~~ that iterates through a sequence of characters and puts each character one by one into the state machine. After that, the status of the state machine is checked. Depending on the results (e.g., the state machine accepts) the user is informed.

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

```
public static void Main(string[] args)
{
    const string textToCheck = "thisIsCamelCase"; var myMachine = new
    LowerCamelCaseMachine();

    foreach (char letter in textToCheck)
    { myMachine.Invoke(letter);
    }

    if (myMachine.IsEndState)
    {
        WriteLine("Text is in lower camel case!");
    } else
    {
        WriteLine("Text is NOT in lower camel case!");

        if (myMacine.CurrentState == "UpperChar")
        {
            WriteLine("Text has multiple upper-case letters in a row or ended with
                upper-case.");
        }
        else                if (myMacine.CurrentState == "Init")
        {
            WriteLine("Text is empty or started with upper case.");
        }
    }
}
```

Listing 4: A console application written in C♯ pseudocode. The state machine LowerCamelCaseMachine (written in Cδ) is used.

No detailed explanation will be given for this C♯ pseudo-code (Listing 4). The defined state machine LowerCamelCaseMachine is ~~instantiated~~ initiated like any other object in C♯. Line 5 creates a constant sequence of characters, which

Commented [V16]: Is this what you mean?

Vangie Stice-Israel
ACADEMIC ESL EDITING SAMPLE—TECHNOLOGY

5-star review:

I am very satisfied with the work of Vangie. He or she put really much effort into finding my mistakes and even found the best hidden ones. I even learned a lot when I transfered the suggestions back into my original document. Vangie's comments helped me to understand WHY something was wrong and how to improve from there. The delivery date was also sooner than expected. Great!

the state machine will read by the state machine. In lines 6-7 the application iterates through the constant and puts each character into the state machine.

This is done with the method Invoke. This method takes an argument as input and changes the current state accordingly. Code blocks defined for transitions and states will be executed. In this case, the user would be informed about the evaluation process defined in the state machine source code (Listing 3).

Finally, the application checks on the status of the state machine. If the state machine halted in a final state (IsEndState), then a success message is printed into the console. Otherwise, the state machine halted in a non-final state or failed to find a suitable transition. This means that the sequence was rejected. Additional information can be read, such as like which state is the current state. With this, more precise messages can be printed. E.g.For example, if the state machine has not accepted and the current state is Init, then the sequence input must either have been empty or started with anything but a lowercase character.

This elaborate example shows how finite-state machines can be defined in Cδ. The presented constructs presented are the basic feature set implemented in the Cδ transcompiler and not a comprehensive feature list. A detailed enumeration of the final feature set is discussed in the section 3, Cδ programming language: implementation.

Commented [V17]: Since it's the beginning of a sentence