Engagement-Reflection in Software Construction

Quinten Rosseel and Geraint A. Wiggins
Artificial Intelligence Lab, Vrije Universiteit Brussel, Pleinlaan 9, 1050 Brussels, Belgium
quinten.rosseel@vub.be, geraint@ai.vub.ac.be

Abstract

This paper introduces experimental research in progress concerning a novel creative software construction method that uses the Engagement-Reflection model and Floyd-Hoare logic. By considering software construction as a creative writing task, we demonstrate how principles from story generation can be applied in software construction.

Introduction

In this paper, we present work in progress on the automated construction of programs. Our approach differs from most such activities because we view program construction as a creative process, rather than a deductive one, and we apply technology from computational creativity to address it. The Engagement-Reflection model (Sharples, 1995) is a well-known approach to narrative generation. Here, we metaphorically replace the narrative with a program, and the well-formedness constraints on narratives with program semantics specified using a standard method of program analysis.

The paper is laid out as follows. In the background section, we illustrate prior research in creative story writing and story generation that inspired this work. More specifically, the Engagement-Reflection model (Sharples, 1995) and an application of it, named MEXICA (Pérez ý Pérez and Sharples, 2001).

In the preamble, we look more closely at Floyd-Hoare (FH) logic (Hoare, 1969), the principal formalism for reasoning about programs in this work. We employ FH-logic for describing assignments with boolean conditions over program variables and extend it with distance measures and transformations that enable construction of simple programs within the ER-model, given user-defined specifications.

After a technical description of the construction process, we provide an outlook for the most important challenges that are to be tackled in the research following this paper.

In essence, the conditions set out by the user define the conceptual space (Boden, 1992) of programs that the system can explore. Exploration of this conceptual space is done by virtue of the ER-cycle that drives program composition forward. After each ER-cycle, the conditions bounding the conceptual space can be transformed by applying transformational rules, yielding new conceptual spaces that can in turn be explored by additional ER-cycles.

Background

The Engagement-Reflection (ER) Model

The ER-model of Sharples (1995) is a cognitive model that describes the creative process of authors writing stories. The idea is to decompose the writing task into two phases called engagement and reflection. This composition is based on the observation that a writer cannot simultaneously enact a writing procedure and re-represent it at the same time (Karmiloff-Smith 1990, quoted by Sharples 1995). This means that reflecting on a text requires the writer to stop writing, resulting in a cycle of engaged knowledge telling, interleaved with periods of reflection (Sharples, 1995).

Cycles can have a short period, as when a writer looks back over each sentence as it is written, or a longer period, as when a writer looks back over a paragraph as it is written. The interaction between engagement and reflection pushes the composition of material forward, with engagement providing new material for consideration and reflection offering a re-interpretation of the material, together with new plans to be enacted (Sharples, 1995).

The ER-model has been successfully applied in story generator MEXICA (Pérez ý Pérez and Sharples, 2001), that produces story frameworks about the Mexicas, old inhabitants of modern México City. MEXICA shows that it is possible to generate coherent content that satisfies an initial set of user-defined constraints using the ER-model.

From Stories to Programs

Cognitive models of creative writing and story generators are of interest because stories and programs share properties that suggest how ideas from creative writing procedures can be applied in the domain of software construction.

Characters and variables are arguably the most straightforward correspondence. Just as characters may take different forms and personalities, variables may be bound to different types and values. Characters in the story are subject to actions and events, pushing the emerging story in a particular direction. Likewise, variables are subject to statements calls that modify the variable state, causing the program to behave accordingly.

In most stories, the plot builds up to one or more protagonists that partake in some final activity or event that ends the story. In the same way, the goal of the program is to assign a set of variables to a desired value or mutual relation. In some cases, story actions and events require that characters are in a particular state or mutual relation before they can be executed. This accords with conditional statements of the programming language.

Preamble

Floyd-Hoare (FH) Logic

FH logic is the principal formalism behind the software construction processes in this work. The central component of the logic is the Hoare triple, that represents how a set of statements $\mathcal{S}$ changes the variable state. This state is captured in a set of conditions on the values of the variables.

Condition A condition c is a boolean assertion on the value of a variable, before or after execution of a set of statements $\mathcal{S}$. Conditions are stored in a global set of conditions $\mathscr{C}$, statements are stored in a global set of statements $\mathscr{S}$.

Hoare Triple $(\mathcal{Q})\ \mathcal{S}\ (\mathcal{R})$ is a Hoare triple that asserts that an ordered set of program statements $\mathcal{S} \subseteq \mathcal{P}(\mathscr{S})$, satisfying a set of preconditions $\mathcal{Q} \subseteq \mathcal{P}(\mathscr{C})$ before running, will satisfy a set of post-conditions $\mathcal{R} \subseteq \mathcal{P}(\mathscr{C})$ after running. All Hoare triples are stored in the global set of Hoare triples $\mathscr{T}$. $\mathcal{P}$ is the standard power set operation over a set.

In practice, we use triple constructions backwards when generating programs. We know that the result of a set of statements $\mathcal{S}$ satisfies a program state $\mathcal{R}$, and we need to infer an assertion $\mathcal{Q}$ on the state before $\mathcal{S}$, provided that $\mathcal{S}$ terminates.

Axiom of Assignment When we know that a set of conditions $\mathcal{Q}$ is true after assigning an expression E to variable V, then this means that the substitution of V by E in the set of conditions $\mathcal{Q}$ must hold before the assignment. Note that $\mathcal{Q} \subseteq \mathcal{P}(\mathscr{C})$ and $\{(V := E)\} \subseteq \mathcal{P}(\mathscr{S})$.

$$\frac{}{(\mathcal{Q}_{[V \mapsto E]})\ \ \{(V := E)\}\ \ (\mathcal{Q})}$$

The assignment axiom has no premises and generates the weakest set of preconditions that is needed for the execution of the assignment to result in a state that satisfies the set of post-conditions (Bridge, 2003).

Example 1 Proof that $x = 1$, given that $x = 2$ after the assignment $x := x + x$ using the assignment axiom.

$$\dfrac{\dfrac{\dfrac{}{(\{x = 2\}_{[x \mapsto (x+x)]})\ \ x := (x + x)\ \ (\{x = 2\})}}{(\{(x + x) = 2\})\ \ x := (x + x)\ \ (\{x = 2\})}}{(\{x = 1\})\ \ x := (x + x)\ \ (\{x = 2\})}$$

□

Extensions to Floyd-Hoare logic

If we know how to transform a program state $\mathcal{R}$ to another state $\mathcal{Q}$ with a set of statements $\mathcal{S}$, example 1 suggests that program construction is possible with a set of start conditions $\mathcal{Q} \subseteq \mathcal{P}(\mathscr{C})$, a set of goal conditions $\mathcal{R} \subseteq \mathcal{P}(\mathscr{C})$ and an ordered set of statements $\mathcal{S} \subseteq \mathcal{P}(\mathscr{S})$.

Valued Condition Distance Inserting relevant statements requires a distance metric that quantifies how a set of statements influences the state of a program with respect to its goal. When conditions are real-valued, a distance δ between two conditions sets $\mathcal{Q} \subseteq \mathcal{P}(\mathscr{C})$ and $\mathcal{R} \subseteq \mathcal{P}(\mathscr{C})$ can be determined by a pairwise difference of each variable v, asserted in both $\mathcal{Q}$ and $\mathcal{R}$, where $E(c)$ evaluates a condition $c \in \mathscr{C}$ to the asserted value of its variable.

$$\delta(\mathcal{Q}, \mathcal{R}) = \sum_{\forall v:\ q_v \in \mathcal{Q}, r_v \in \mathcal{R}} \mid E(q_v) - E(r_v) \mid$$

Condition Variables In order to specify abstract relations over triple conditions, we introduce condition variables, stored in a global dictionary $\mathscr{V}$. Each entry (C, p) in $\mathscr{V}$ is a one-to-one mapping between a condition variable C, denoted with an uppercase letter, to a program variable p, denoted with a lowercase letter. Condition variables enable us to specify abstract relations that allow conditions to satisfy a broader set of program states.

Abstract Condition Distance Conditions that involve abstract expressions cannot be evaluated by value but are compared with respect to an abstract distance function. In this context, we look at the minimum tree edit distance (TED) of the abstract syntax tree (AST) associated with the expression in the condition. Hence, insertions, deletions and substitutions are executed on the level of operators, function applications, variables and values.

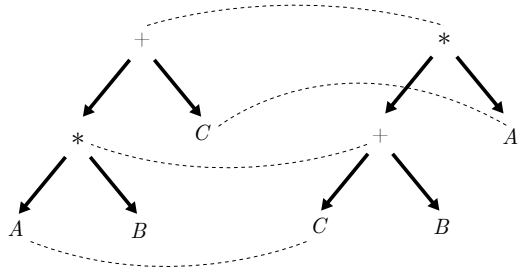

Figure 1: A mapping between $A * B + C$ and $(C + B) * A$ with 4 substitutions.

The TED problem has a well-known, dynamic programming solution (Zhang and Shasha, 1989) that allows for a customizable cost function γ to evaluate insertions, deletions and substitutions of nodes in trees. This is desirable as not all tree operations need the same associated weight in the distance metric.

Variable Dependency Transformation Being informed on condition variables in $\mathscr{V}$, enables the system to rewrite conditions from triples in $\mathscr{C}$ when given patterns arise.

The following rule is aimed at separating abstract variable dependencies. A Hoare triple with a set of statements $\mathcal{S} \in$

$\mathcal{P}(\mathscr{S})$, a set of post-conditions $\mathcal{R} \subseteq \mathcal{P}(\mathscr{C})$ and a singleton precondition of the form

$$\Big\{\big(p_0 = f_1(C_1)\ \circ_1\ \ldots\ \circ_{n-1}\ f_n(C_n)\big)\Big\} \subseteq \mathcal{P}(\mathscr{C})$$

with $f_1 \ldots f_n$ functions involving one and only one condition variable, $C_0 \ldots C_n$ mutually independent condition variables and $\circ_1 \ldots \circ_{n-1}$ binary operations $\in \{\oplus, \ominus, \otimes, \oslash\}$, can be transformed with the variable dependency transformation when $(C_0, p_0) \ldots (C_n, p_n) \in \mathscr{V}$.

$$\frac{\Big(\Big\{\big(p_0 = f_1(C_1)\ \circ_1\ \ldots\ \circ_{n-1}\ f_n(C_n)\big)\Big\}\Big)\ \mathcal{S}\ \Big(\mathcal{R}\Big)}{\Big(\Big\{\big(p_1 = f_1(C_1)\big) \ldots \big(p_n = f_n(C_n)\big)\Big\}\Big)\ p_0 = p_1 \circ_1 p_2 \ldots p_{n-1} \circ_{n-1} p_n\ ;\ \mathcal{S}\ \Big(\mathcal{R}\Big)}$$

Example 2 The following proof employs condition variables and a variable dependency transformation. The goal is to generate the Pythagoras relation between three program variables a, b and c. The start specifications t_1, t_2 and t_3 make sure that $(A, a), (B, b)$ and $(C, c) \in \mathscr{V}$.

$$t_1 :\ \Big(\emptyset\Big)\ \emptyset\ \Big(\Big\{\big(a = A\big)\Big\}\Big)$$

$$t_2 :\ \Big(\emptyset\Big)\ \emptyset\ \Big(\Big\{\big(b = B\big)\Big\}\Big)$$

$$t_3 :\ \Big(\emptyset\Big)\ \emptyset\ \Big(\Big\{\big(c = C\big)\Big\}\Big)$$

The goal specification t_4 is defined in terms of A and B and should be satisfied for all programs that are generated. By equaling c to both C and $\sqrt{A^2 + B^2}$ in the start and goal conditions, the system is told to find a way to make $C = \sqrt{A^2 + B^2}$.

$$t_4 :\ \Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)\Big\}\Big)\ \emptyset\ \Big(\emptyset\Big)$$

As the derivation process works backwards from t_4, the system might at some point infer a triple t_5 using the assignment axiom. Assume that c is positive in this example.

$$\frac{\Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)_{[c \to sqrt(c)]}\Big\}\Big)\ c := sqrt(c);\ \Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)\Big\}\Big)}{\Big(\Big\{\Big(\sqrt{c} = \sqrt{A^2 + B^2}\Big)\Big\}\Big)\ c := sqrt(c);\ \Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)\Big\}\Big)}$$
$$t_5 :\ \Big(\Big\{\Big(c = A^2 + B^2\Big)\Big\}\Big)\ c := sqrt(c);\ \Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)\Big\}\Big)$$

Using the variable dependency transformation rule, the system is now able to rewrite triple t_5 as follows. This allows the triple to be further evaluated in terms of A and B separately.

$$\frac{t_5 :\ \Big(\Big\{\Big(c = A^2 + B^2\Big)\Big\}\Big)\ c := sqrt(c);\ \Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)\Big\}\Big)}{\Big(\Big\{\big(a = A^2\big), \big(b = B^2\big)\Big\}\Big)\ c := a + b;\ c := sqrt(c);\ \Big(\Big\{\Big(c = \sqrt{A^2 + B^2}\Big)\Big\}\Big)}$$

Finally, in the optimal case, the system might finish the program by applications of the assignment axiom, resulting in t_6 and t_7.

$$\frac{\Big(\Big\{\big(a = A^2\big)_{[a \to (a*a)]}, \big(b = B^2\big)_{[a \to (a*a)]}\Big\}\Big)\ a := a * a;\ \Big(\Big\{\big(a = A^2\big), \big(b = B^2\big)\Big\}\Big)}{t_6 :\ \Big(\Big\{\big(a = A\big), \big(b = B^2\big)\Big\}\Big)\ a := a * a;\ \Big(\Big\{\big(a = A^2\big), \big(b = B^2\big)\Big\}\Big)}$$

$$\frac{\Big(\Big\{\big(a = A\big)_{[b \to (b*b)]}, \big(b = B^2\big)_{[b \to (b*b)]}\Big\}\Big)\ b := b * b;\ \Big(\Big\{\big(a = A\big), \big(b = B^2\big)\Big\}\Big)}{t_7 :\ \Big(\Big\{\big(a = A\big), \big(b = B\big)\Big\}\Big)\ b := b * b;\ \Big(\Big\{\big(a = A\big), \big(b = B^2\big)\Big\}\Big)}$$

□

Construction of Programs

This section provides more background on how program construction employs previously described concepts to generate software in the C programming language. We distinguish three main processes at the highest level.

- Program Input Parsing: extract and generalize statements from the input programs to Hoare triples.
- Program Sequence Engagement: select Hoare triples and construct programs.
- Program Sequence Reflection: verify and edit engaged program sequences and its Hoare triples according to the specifications set out by the user.

In order to generate programs, the user provides an inventory of programs and specifies the start and goal conditions in the form of Hoare triples, like for example t_1, t_2, t_3 and t_4 in example 2. After parsing the inventory, the system uniformly draws statements from $\mathscr{S}$ and retains those that reduce the distance between the user-defined start and goal conditions.

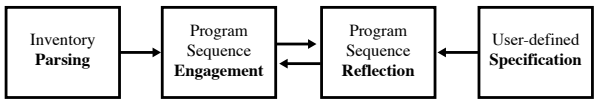

Figure 2: The program construction process.

Program Input Parsing

Before any meaningful program can be constructed, the system requires an inventory of valid C programs to serve as experience during engagement and reflection. This inventory is parsed and represented as Hoare triples in the triple store $\mathscr{T}$ together with statements in the statement store $\mathscr{S}$.

At parse-time, statements are generalized before storage in $\mathscr{S}$ so that they can be instantiated to a variable that is part of the user's specification. Additionally, user-defined goal specifications are not taken into account at parse-time as boolean assertions inside conditions are only initialized upon reaching reflection. However, each statement inside

a Hoare triple gets equipped with a Hoare transition rule that semantically corresponds with that statement, such that boolean assertions can get propagated through when they are used inside triples.

It is worth mentioning that there are many ways to extract triples from an inventory of programs. We choose to retain the semantics of the inventory as much as possible when storing triples inside the triple store $\mathscr{T}$ and do not bother to extract every possible sub-program within a parsed program, as this would cause a combinatorial explosion of triples in $\mathscr{T}$. Although we see that it can yield benefits for better and more diverse composition, this is not the current focus in this research.

Program Sequence Engagement

The engagement phase constructs programs from the knowledge stored in the triple store $\mathscr{T}$, the statement store $\mathscr{S}$ and the condition store $\mathscr{C}$. The selection process selects a random triple from $\mathscr{T}$ and instantiates the triple's statements with a random variable from the user's specification in $\mathscr{C}$. Subsequently, triples are inserted to the program sequence, a structure that represent the program under construction. This process repeats until the maximum number of engagement iterations is reached.

We are aware of the fact that this process has more potential beyond random selection methods and that more informed selection procedures can be explored in future work.

Program Sequence Reflection

Reflection is the most decisive factor in the software construction process. When engagement transmits a program sequence, it's up to reflection to determine what triples are relevant for the specification passed by the user. Reflection picks up from the last analyzed triple in the program sequence and propagates its initialized post-conditions upwards through the triple's associated statements, using the Hoare transition rules. When a condition has propagated through all statements inside the triple, the condition is part of the set of preconditions of that triple. The goal is to obtain a triple with the same preconditions as the start conditions set out by the user. Reflection continues until the maximum amount of reflection cycles is reached or when the user-defined start conditions are satisfied.

A triple under consideration is deemed relevant when the list of statements inside that triple transforms the preconditions of the last analyzed triple closer to the start conditions. This is done by initializing the post-conditions of the triple under consideration, applying its transition rules and comparing these preconditions with the start conditions. If relevant, the triple is kept in the program sequence, otherwise it is removed.

One might rightfully object that this can introduce local minima with respect to the distance metric in the trajectory of triple insertions. Inserting one triple might not improve the conditions with respect to the start conditions, but it might improve the conditions as a combination with other triples. This leads into territory of Hoare triple merging and block structures, the current focus of this research and one of the topics in the next section.

Finally, when the maximum number of engagement iterations is reached, the system checks if it is able to rewrite the triples in $\mathscr{T}$ by applying transformation rules that are triggered by predefined patterns in conditions, like the variable dependency transformation.

Conclusions

At this point, we have discussed the most significant mechanisms that underlie the program construction process of this work. This last section provides a short framing of the future challenges to be tackled.

Blocks and Control Flow

The system described up until now focuses on combinations of single statements, allowing it to compose simple programs in the C programming language. The next logical steps lie in the ability to work with compound statements or blocks, paving the way for control flow structures that specify the logical order in which computations are performed, like conditionals and loops.

Blocks are delimited by curly braces to group together declarations and statements, which makes them syntactically equivalent to single statements for the parser of a C compiler (Kernighan and Ritchie, 1978). All control-flow structures of interest employ block structures, so it is no surprise that Hoare triples need to be general and flexible in their representation for block structures. Additionally, block representations will need to be editable, such that reflection can evaluate and edit nested block statements too. Allowing block-structure flexibility will enable the system to combine and reuse different parts from the inventory of programs. These merging strategies will proof useful when local optima arise in the construction of programs.

On a more technical note, compound statements require the parser to pass additional information to the triple representation in $\mathscr{T}$ so that its transitional rules can be applied accordingly. As each block deals with its own scope, the engagement phase needs to employ more informed initializations for the variables in each block and take into account lexical scoping.

Software Construction as a Creative Activity

Before we can consider the system creative, it needs a component that evaluates generated concepts (Wiggins, 2006). Currently, system output is manually evaluated after the start conditions are satisfied by the ER-cycle. In future work, more automated evaluation processes based on software engineering metrics of size and complexity need to be incorporated (Fenton and Bieman, 2014). This information can be used as a feedback signal to adapt the conceptual space that determines program construction, similar to MEXICA's filter system.

References

Boden, M. 1992. *The Creative Mind*. London: Abacus.

Bridge, D. 2003. Lecture 17: Floyd-hoare logic for partial corectness.

Fenton, N., and Bieman, J. 2014. *Software metrics: a rigorous and practical approach*. CRC press.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12(10):576–580.

Karmiloff-Smith, A. 1990. Constraints on representational change: Evidence from children's drawing. *Cognition* 34(1):57–83.

Kernighan, B. W., and Ritchie, D. M. 1978. *The C programming language*. Prentice Hall.

Pérez ý Pérez, R., and Sharples, M. 2001. Mexica: A computer model of a cognitive account of creative writing. *Journal of Experimental & Theoretical Artificial Intelligence* 13(2):119–139.

Sharples, M. 1995. *An account of writing as creative design*. University of Sussex.

Wiggins, G. A. 2006. A preliminary framework for description, analysis and comparison of creative systems. *Knowledge-Based Systems* 19(7):449–458.

Zhang, K., and Shasha, D. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18(6):1245–1262.