Software Architecture and Engineering 2016
Project Part 1

Published: March 07, 2016
Submission: April 03, 2016 (midnight); this is a firm deadline

Your task is to design an abstract syntax tree of a simple programming language. The emphasis of your task is on designing the data model, expressing it in UML, formalizing it in Alloy, and checking whether certain properties hold for your Alloy model.

We intentionally made parts of the language description below ambiguous. It is your responsibility to clarify any ambiguities with the assistants (your "customers"). If you have questions for the customer, send an e-mail to sae-assistants@lists.inf.ethz.ch.

All people who signed up for the project will be signed up to the mailing list sae-students@lists.inf.ethz.ch, where you can discuss the project with other students, and where we will also post announcements.

Updates - 29th of March

- Clarification: Parameters do not have to be read
- Clarification: `p_isDeclared` only true for variables declared through a variable declaration, not for parameters.
- Clarification: Expressions are always associated with a type
- Clarification: Nodes in trees have at most one parent

1 The Linear language

A Linear program consists of functions and types. Functions consist of a linear sequence of statements, which may or may not contain expressions. Expressions are associated with a type.

Functions have a return type, as well as a set of formal parameters with types. A function can be called in a corresponding call expression, which is assiociated with a set of expressions that serve as actual parameters and are mapped to formal parameters. A return statement carries an expression that defines which value is returned by the function and terminates the execution of the function body. Each function execution must be terminated by a return

statement. A function may not contain unreachable statements. Recursion is not allowed. There is one main function that takes no parameters and from which all other functions are transitively called.

Variables are declared exactly once, either in a corresponding declaration statement or as function parameter. Declaration statements declare variables of a specific type and must appear in the same function before the first use. A variable that has been declared can be assigned to using an assignment statement, which has some expression on the right-hand side. Once the variable has been assigned to, it can be used in expressions in subsequent statements. We do not allow dead variables (variables that are not parameters and are never read) or dead assignments (assignments that are not followed by a read of the variable). Furthermore, parameters should never be assigned to.

Besides function calls and variable references, expressions can be literals, which also have a type. For the sake of simplicity, we do not model the value of literals. Expressions form trees, and nodes of expression trees are never shared, i.e. every node has at most one parent.

Each type can have up to one supertype. The usual typing rules apply: For example, if a variable is declared with type T_1 and then assigned to a second variable of declared type T_2, then T_1 must be a subtype of T_2. Similar rules apply to function calls and return statements.

2 UML Model (40 points)

Task A. Create a UML class diagram of a data structure representing valid LINEAR programs. Include all the relevant relations and details. In addition, document any detail that cannot be encoded in the UML class diagram. OCL specifications are not required. Use the best design practices you have seen in the lectures.

3 Alloy - Static Model

Task B. (30 points) Create an Alloy model of valid LINEAR programs. Include all the relevant details, relations and facts. Ask your TA in order to clarify any ambiguities. In addition, document any detail that cannot be encoded in the Alloy model.

In addition to satisfying the above requirements as precise as possible, the alloy model should satisfy the following formal requirements:

- The signature containing all expressions is called `Expr`
- The signature containing all variables is called `Variable`
- The signature containing all functions is called `Function`
- The signature containing all literal expressions is called `Literal`

- The signature containing all statements is called `Statement`
- The signature containing all local variable declarations is called `VarDecl`
- The signature containing all types is called `Type`
- The signature containing all formal parameters is called `FormalParameter`
- The model defines the following functions:
  - `p_numFunctionCalls[]:Int`
    Returns the number of function calls in the program.
  - `p_expressionTypes[]:set Type`
    Returns the types of all expressions.
  - `p_literalTypes[]:set Type`
    Returns the types of all literals.
  - `p_statementsInFunction[f:Function]:set Statement`
    Returns all statements directly contained in the body of a function.
  - `p_statementsAfter[s:Statement]:set Statement`
    Returns all statements contained after `s` in the same function.
  - `p_parameters[f:Function]:set FormalParameter`
    Returns the formal parameters of function `f`.
  - `p_subExprs[e:Expr]:set Expr`
    Returns the direct subexpressions of `e`.
- The model defines the following predicates:
  - `p_containsCall[f:Function]`
    true iff `f` contains a function call in its body.
  - `p_isAssigned[v:Variable]`
    true iff `v` appears on the left side of an assignment anywhere the program.
  - `p_isRead[v:Variable]`
    true iff `v` appears in an expression anywhere the program.
  - `p_isDeclared[v:Variable]`
    true iff `v` is declared through a variable declaration.
  - `p_isParameter[v:Variable]`
    true iff `v` is declared as a parameter.
  - `p_subtypeOf[t1:Type,t2:Type]`
    true iff `t1` is a subtype of `t2`.
  - `p_assignsTo[s:Statement,vd:VarDecl]`
    true iff `s` assigns to the variable declared by `vd`.

Task C. (25 points)

Generate the following instances using Alloy, showing that your model is not overly restricted, or explain why they are infeasible if you cannot generate them:

1. A program with one function, 2 function calls
2. A program with two functions, 2 function calls
3. A program with exactly 1 assignment, 1 variable, 1 literal (no restrictions on other kinds of expressions)
4. A program with exactly 1 function call, which is on the right-hand side of an assignment, where the expression inside the called function's return statement is of a different type than the return type of the function, and both are different from the type of the variable on the left-hand side of the assignment.
5. A program with exactly 1 literal and exactly 2 incomparable types.

Export all feasible models both as images and as XML.

4 Alloy - Dynamic Model

Task D. (30 points)

In this task, we create a very simple model of program executions, to make statements about their runtime behavior. Our model is based on the model and language from Task B. It is recommended to solve Tasks B and C completely first, and then start this task with a copy of the model in Task B.

In the first step, we simplify our model to capture a simplification of the Linear language, called BooleanLinear, in which all values are of Boolean type:

- Extend your model by expressions that model the binary operation And and the unary operation Not.
- Every function is called at most once.
- Every literal is associated with a fixed value - either True or False.
- Remove types declarations, type checking rules and types annotated in expressions and statements.
- We now allow the main function to take parameters.

Then, add to your model the concept of *executions*. An execution reflects the value of each variable at each point in the program and uniquely relates every expression in the program to a value from the set `True`, `False`, `Undefined`. The values of expressions should follow the usual rules, i.e. the expression (V

AND True) is True if and only if the value of V is True. Evaluation is strict. Note that your model should be general enough to model one program with any number of its executions. Make sure that values are correctly passed in and out of functions, and that assignments to variables are reflected correctly. Declared but unassigned variables take the value `Undefined`, undeclared variables have no value.

In addition to satisfying the above requirements precisely, the Alloy model should satisfy the following formal requirements, as well as the ones specified in Task B.

- The signature containing all executions is called `Execution`
- The signature containing all values is called `Value`, its only members are `True,False,Undefined`
- The model defines the following functions:
  - `p_val[e:Execution,p:Expr]:Value`
    Returns the value of `p` in execution `e`.
  - `p_retval[e:Execution,f:Function]:Value`
    Returns the return value of `f` in `e`
  - `p_argval[e:Execution,f:Function,p:FormalParameter]:Value`
    Returns the value of formal parameter `p` in execution `e`
  - `p_numNot[]:Int`
    Returns the number of Not-expressions
  - `p_valbefore[e:Execution,s:Statement,v:Variable]:Value`
    Returns the value of variable `v` before executing statement `s` in execution `e`

Task E. (25 points)

Generate the following instances using Alloy, showing that your model is not overly restricted, or explain why they are infeasible if you cannot generate them:

1. A program that takes 2 arguments and computes AND
2. A program that takes 2 arguments and computes NAND
3. A program that takes 2 arguments, has at least one literal and one assignment and computes OR
4. A program that takes 2 arguments and computes XOR
5. A program that takes 2 arguments, has 3 functions and at least one assignment and computes NAND. Here, one function that is not the main function should contain an And, and the other function that is not the main function should contain a Not.

Export all feasible models both as images and as XML.

To generate these programs, you may only restrict your programs as defined above, e.g. for the first model, you restrict only the number of arguments. However, you may restrict the values of parameters and return values of any number of executions associated with the program. You are also allowed to restrict the number of elements in all sets when running the corresponding predicates, to keep the solving time within reasonable bounds.

5 Deliverables

Submit your solution by email to sae-assistants@lists.inf.ethz.ch, including:

1. A Zargo file of the UML class diagram from Task A in ArgoUML
2. An Alloy file with all the required code from Tasks B–C. The file must include short comments that explain your formalization
3. An Alloy file with all the required code from Tasks D–E. The file must include short comments that explain your formalization'
4. An XML file for each generatable instance from Tasks C and E.
5. A PDF file that includes:
   - The UML class diagram from Deliverable 1
   - A list of details from the project description that cannot be expressed in the UML model
   - The Alloy model from Deliverable 2
   - The Alloy model from Deliverable 3
   - A list of instances from Task C that you were not able to produce, each with a short explanation why the instance is not feasible
   - Image exports of all the generated instances from Task C
   - A list of instances from Task E that you were not able to produce, each with a short explanation why the instance is not feasible
   - Image exports of all the generated instances from Task E

6 Resources

- ArgoUML: http://argouml.tigris.org/
- Alloy: http://alloy.mit.edu/alloy/
- PDF creation: You can use Microsoft Word or LaTeX to create your document and **export it to PDF**