An aspect language for robust programming

Pascal Fradet
IRISA/INRIA-Rennes
www.irisa.fr/lande/fradet

Mario Südholt
École des mines de Nantes
www.emn.fr/sudholt

1 Introduction

Robust programs should satisfy two basic conditions. First, compute well-defined output values from well-defined sets of input values, henceforth called the program's standard domain. Second, input values from the complement of the standard domain, henceforth called the program's exceptional domain, should result in well-defined error situations. Most real programs, however, are not robust. While the first condition is the subject of almost all programming methodologies and most programming efforts, the second is violated in most non-trivial programs which often yield incorrect results once an erroneous situation has appeared. Worse, the incorrect results are frequently not recognized as such, for instance, because they approximate correct values quite well or they are immediately used in subsequent computations. In this position paper, we advocate the use of an aspect language for robust programming. AOP is particularly appealing for this task because robustness crosscuts traditional structuring means. Consider, for instance, the problem of ensuring that a global index remains in a given range. The code needed to check such an invariant is typically scattered all over the program.

The paper is structured as follows: Section 2 presents an example-driven introduction of the proposed aspect language for program robustness; Section 3 discusses its semantics and implementation; Section 4 suggests extensions and concludes.

2 An aspect language for robustness

Conceptually, an aspect language for robustness must provide means for two kinds of information: the standard/exceptional domains of variables and the handling of exceptional situations. We focus here on numerical domains and fix component programs to be imperative programs [4, 1].

Consider the program shown in Figure 1 which calculates the arithmetic means of an initial segment of length `i` of an array `x`. The example program is not robust for at least three reasons: 1. since `i` is used to index `x` it must lie within the array bounds; 2. `j` must not be zero in the dividing statement; 3. the calculation of the sum in the while-loop may overflow.

```
int mean(int x[99], int i) {
  int n = 0;
  int j = 0;
  while (i != 0) {
    i := i - 1;
    j := j + 1;
    n := n + x[i];
  }
  n = n / j;
  return n;
}
```

Figure 1: C-like program calculating arithmetic means

Our robustness aspect allows these three robustness conditions to be formulated as follows:

1. `INVARIANT (0 <= i <= 99) HANDLE abort "index i out of range"`

   An invariant directive enforces its condition on the whole component. It refers to syntactic entities of the component entities by means of patterns, such as pattern `i` for "the program variable `i`". Definition 1 states that the standard domain of `i` must be $[0, 99]$. In general, a condition can be any linear constraint involving numerical variables. Definition 1 stipulates that program execution has to be aborted with the given error message if this invariant is violated. It would cause no semantic difficulties to generalize the handling to arbitrary *closed* programs (i.e. where the component execution is not resumed) instead of a simple abort.

2. `DOMAIN (V:var != 0) IN (_ / V) HANDLE abort "division by V == 0"`

   A domain directive enforces its condition of a set of program points specified using patterns. Here, `V` is an aspect variable of type `var` denoting component program variables. The pattern `(_ / V)` matches all dividing statements whose divisor is a single program variable. The aspect language offers several high-level patterns to select, for example, name spaces (scopes), regions between two specific statements, the enclosing block of a specific statement, etc. Definition 2 states that, for each program point where a division by a variable occurs, this variable must be different from zero. In other words, it requires the standard domain to exclude divisions by null variables. It applies to the division using `j` in the example program.

3. `OVERFLOW IN (while(_, _);_) HANDLE extend(n)`

   An overflow directive specifies a set of regions where overflow exceptions must be caught and treated by a specific handling. For overflows (and underflows) the handling is more sophisticated and returns to the component code for a retry. The semantics remains manageable because we restrict ourselves to a standard and controlled handling. `extend(vars)` doubles the number of bits used to represent the variables `vars` by two before a retry (and so on until no overflow occurs). Definition 3 specifies that overflow has to be treated in the statement sequence containing the while-loop and the statement following the loop. This overflow will be handled by extending the representation of the component variable `n` in the region comprising the while-loop and the following statement.

3 Semantics and implementation

The standard domain is defined as part of the robustness specifications: invariant directives impose restrictions on the whole program, domain directives allow restrictions to be imposed on specific parts of the program (such as all division statements) and overflow directives impose restrictions on numerical calculations. The semantics of the woven program is the same as the original program on the standard domain. The semantics on the exceptional domain is more complicated since there is no insurance that all possible overflows or underflows are captured by the aspect. On the exceptional domain, the woven program yields either an error message (e.g. an invariant directive has been violated), the same value as the standard semantics of the original program over arbitrary-precision values (overflows/underflows have been handled) or, as the standard semantics of the original program, an ill-defined result (an overflow/underflow has not been captured).

Let us point out that the semantics of the aspect-oriented program has been specified without consideration of the implementation, the aspect weaver or the actual woven program. We believe this to be a crucial property of an aspect language.

The implementation is based on a generic analysis and transformation based framework for AOP [3]. In this framework, an *aspect* is just a collection of source-to-source transformations whereas *join points* are defined using patterns over abstract syntax trees. Join points and transformations are specified using TrafoLa-H [5], a language designed to express program transformations. The generic aspect *weaver* is defined simply as a fixpoint applying the transformations to the component program. Transformation application is done exhaustively (i.e. as long as one of the transformations can still be applied) and can be done in any order.

In this setting, the implementation of our robustness aspect amounts to translating its directives into TrafoLa-H expressions. Invariant and domain directives are translated into transformations which introduce tests in the component program ensuring that program execution is aborted if the corresponding conditions are violated. The overflow directive translates to a transformation which encapsulates the code it matches into a while-loop executing this code on values of higher and higher precision. This loop will be executed until no overflow occurs anymore. On exit of this code, extended variables are cast back to their original precision (an overflow at this point would entail an abort).

Since weaving is a repeated application of the transformations in no specific order, the translation must ensure termination and confluence. In our case, this amounts to making sure that transformations do not overlap and cannot insert repeatedly the same test at the same program point.

Up to now, we said that conditional statements are inserted before any statement matching the code pattern of invariant and domain directives. Obviously this approach is not very efficient because many superfluous tests may be inserted. Static analysis techniques should be used to eliminate many of these useless tests without changing the semantics of the woven code. For numerical constraints, we rely on standard analysis techniques to infer linear constraints among program variables [2]. The analysis annotates the abstract syntax tree with properties. Invariant and domain directives are translated into transformations which insert tests only if the property inferred by the analyzer at the current program point does not imply the condition. For example, the domain directive

```
DOMAIN (V:var != 0) IN (_ / V) HANDLE abort "division by V == 0"
```

is translated into

`stmt && contains(_:expr / V:var)` $\Longrightarrow$

if $\not\vdash$ `V` $\neq 0$ and `!precedes(ifStmt, match)` then `ifStmt; match`

where `ifStmt = if (V == 0) abort "division by V == 0"`

For each statement where a division by a variable `V` occurs, if the analysis is not able to prove that `V` is different from 0 (i.e. $\not\vdash$ `V` $\neq 0$) then a test is inserted. The pattern `!precedes(ifStmt, match)` ensures that the same test is not inserted twice at the same program point and guarantees termination of weaving.

Actually, our language offers a fourth kind of directive to add knowledge and eliminate tests. ASSUME directives allow the analysis to make the hypothesis that a property holds at a specific set of program points. If we know that the procedure `mean` of Figure 1 is always called with a strictly positive `i`, we could add the directive `ASSUME i>0 IN int n = 0;`. Fewer tests will be generated to check the invariant directive. Furthermore, the analysis will

infer that `j` is always greater than 0 (the loop is executed at least once) and no test needs to be inserted to check division by zero. For the example, a single test (`if i>99 then abort "index i out of range"`) has to be inserted at the entry of `mean` to enforce the invariant and domain directives.

4 Conclusion

Our approach to AOP separates issues in three levels. An implementation (or generic) level which provides tools (a generic transformation language and a generic weaver) to implements all kinds of aspect languages. A linguistic level defining a domain specific aspect language; semantics issues are addressed at this level. An application (or user) level where aspects for specific components are written. Here, we focused on the linguistic level and proposed an aspect language for robustness. The aspect language was restricted/designed to provide a clear semantics for aspect oriented programs. Its implementation consists in a translation in terms of basic program transformations of the generic level and a static analysis to optimize weaving. Up to now, we have a completely formal definition of the robustness aspect, its translation to source code transformations and the generic aspect weaver. We are working on the integration of an analyzer to complete a prototype implementation.

We are considering two directions for further research. First, we focused on numerical calculations in this paper because robustness is particularly important in this field. The robustness aspect should be extended to be able to express directives on the other data types. If booleans are a trivial extension, a proper treatment of pointer-based data structures implies the integration of an alias (or points-to) analysis. Second, we are investigating the application of the generic framework to other aspects, in particular a debugging and a security aspect. The former should allow the definition of debugging properties such as "trace the value of variable `x` in procedure `p` as soon as `y` becomes null". A first approach to the latter is the integration of resource-based security schemes such as "deny access to I/O port 123 from processes belonging to process group `pg`".

References

[1] E. Best, F. Cristian: *"Systematic Detection of Exception Occurrences"*, Science of Computer Programming, 1(1-2), 1981

[2] P. Cousot, N. Halbwachs: *"Automatic discovery of linear restraints among variables of a program"*; Proc. of 5th ACM Symp. on Principles of Programming Languages, 1978.

[3] P. Fradet, M. Südholt: *"AOP: towards a generic framework using program transformation and analysis"*; Proc. Workshop on AOP, ECOOP'98

[4] J. R. Hauser: *"Handling floating-point exceptions in numeric programs"*; ACM Transactions on Programming Languages and Systems, 18(2), 1996

[5] R. Heckmann, G. Sander: *"TrafoLa-H Reference Manual"*, LNCS 680, ch. 8, 1993.