Synthetic Diversity:

Generating Alternative Representations and Run-time Monitors for Robustness and Cyber-security

Kestrel Institute Technical Report

Stephen Fitzpatrick and Stephen Westfold

March 4, 2019

This report discusses: (i) generating implementations of an application that use alternative concrete representations of abstract data types; and (ii) generating run-time monitors from semantic constraints.

1 Introduction

An abstract data type (*ADT*) such as Set can be defined without regard to how it is represented (i.e., abstractly). For example, two Sets can be defined to be equal when they have the same members, regardless of how those members are associated with each Set, and such operations as inserting an element into a Set or deleting an element from a Set can be defined in terms of how they affect the Set's membership, again without regard to how the membership is represented.

Nonetheless, an ADT must be given *some* representation if it is to be used in *computations*. For any given ADT, there may be several possible representations — for example, a Set may be represented using a List, with some representations allowing repetitions and others not. If the elements can be ordered (e.g., Integers), then the List representations may be sorted — alternatively, an Ordered Tree representation may be used. Or a Hash Map representation may be used, with a given hash value mapped to a bucket of elements having that hash value.

Figure 1 shows alternatives for: storing documents, representing sets, finding the median of a set of data, and algorithm schemas for optimization.

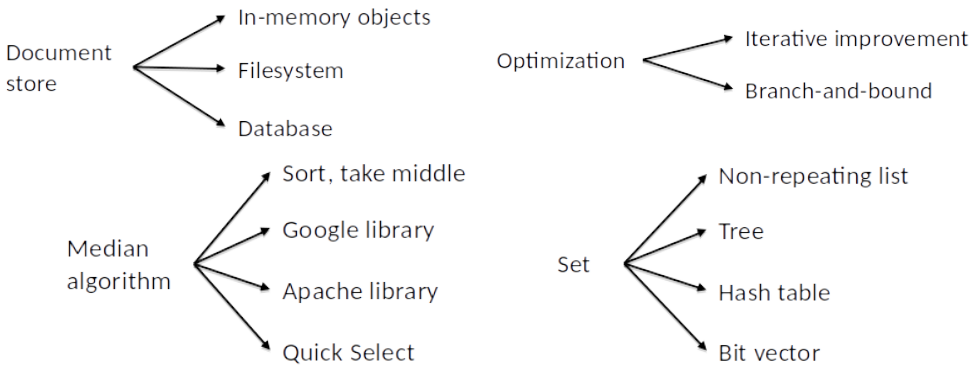

Figure 1: Various alternatives for ADTs and algorithms

Different representations will typically contain different code and exhibit different behaviours — e.g., in the use of memory or CPU time, or in the use of operating system functions or third-party libraries.

In the following, the use of alternative representations is considered for the purpose of achieving *diversity* in an application's code and behaviour. That is, given an application that uses various ADTs, diversified implementations of the application are *generated*, using different combinations of the ADTs' representations. The different implementations will thus contain diversified code and exhibit diversified behaviours. This may help to thwart cyber attacks that depend on detailed knowledge of a *specific* implementation's code or behaviour.

In addition, the generation of run-time monitors is considered. The run-time monitors are based on semantic constraints contained in ADTs (e.g., pre- and post-conditions on their operations), invariants of the concrete representations (e.g., that a particular List-based representation of Set will not contain repetitions), and on the correspondence between an ADT and a representation. A run-time monitor can detect violation of semantic constraints, signaling that something has gone wrong in a computation or that a representation has become corrupt.

The semantic constraints can then be used for *recovery*, by repairing corrupt data or using an alternative representation to repeat an erroneous computation. For example, if a sorted List is found to contain out-of-order elements, then one possible repair is to remove the out-of-order elements; another possible repair is to sort the List. Other repairs are possible.

Monitoring can be enhanced using *multiple*, redundant representations simultaneously, so that they can be cross-checked; likewise, a representation that has been detected to be corrupt can be reconstructed from a representation that is believed to be sound.

The use of monitored and diversified implementations may help to protect an application from:

- attacks that depend on low-level details of a specific representation;
- behaviour-based attacks that depend on an application's use of a specific algorithm (e.g., algorithmic complexity attacks [1]);
- bugs in an algorithm that are peculiar to a specific representation;
- bugs in library routines that are used only by a specific representation.

This report focuses on the high-level principles involved in diversification and the generation of run-time monitors and repair, mainly using examples. See [2] for more technical details.

2 Refining an Abstract Data Type into a Concrete Representation

In this report, an abstract data type, A, and a concrete representation R, are related by a *refinement morphism*: each data type and operation defined in A has a corresponding data type or operation defined in R. The names may be different; for example, in the Set ADT, the type may be called 'Set', whereas in a concrete representation it might be called 'HashSet' or 'SortedList' — the morphism defines the appropriate translation. The concrete representation may also introduce additional (auxiliary) types and operations that do not directly correspond with any types or operations in the abstract data type.

In addition, the concrete representation must preserve the semantics of the ADT (allowing for any renaming defined by the morphism). For example, the Set ADT (presumably) defines the intersection of two Sets as a Set containing those elements that are members of both argument Sets. The SortedList representation of Set must define a corresponding operation that takes

two SortedLists as input (each representing a Set) and produces a SortedList (representing the intersection) that contains those members that are contained in both of the input SortedLists. The *implementation* of intersection for the SortedList representation may be optimized using the fact that each input SortedList is sorted, but regardless, it must preserve the semantics of intersection given in the definition of Set.

2.1 Worked Example: Complex Numbers

The rest of this report will use for worked examples a simple ADT for complex numbers, Complex. The ADT includes some typical *observers*:

- real : Complex → Real, which gives the real part of the Complex Number;
- imag : Complex → Real, which gives the imaginary part;
- rad : Complex → Real, which gives the radius; i.e., the magnitude, constrained to be non-negative;
- arg : Complex → Real, which gives the argument; i.e., the angle, constrained to be in the range $[0, 2\pi)$.

The rad observer gives a constrained value, as does the arg observer. In addition, these two observers are mutually constrained so that the argument is zero if the radius is zero:

$$\forall C : \mathsf{Complex} \cdot \mathsf{rad}(C) = 0 \implies \mathsf{arg}(C) = 0 \ .$$

When the radius is 0, the argument does not matter, and may even be considered to be undefined. Choosing to make it 0 simplifies development.

The observers real and imag form a set of canonical observers — two complex numbers are equal if and only if their real parts are equal and their imaginary parts are equal.

$$\begin{aligned} \forall C_1, C_2 : \mathsf{Complex} \cdot C_1 = C_2 \iff & \quad \mathsf{real}(C_1) = \mathsf{real}(C_2) \\ \land & \quad \mathsf{imag}(C_1) = \mathsf{imag}(C_2) \end{aligned}$$

Likewise, rad and arg are also canonical:

$$\begin{aligned} \forall C_1, C_2 : \mathsf{Complex} \cdot C_1 = C_2 \iff & \quad \mathsf{rad}(C_1) = \mathsf{rad}(C_2) \\ \land & \quad \mathsf{arg}(C_1) = \mathsf{arg}(C_2) \end{aligned}$$

Note that this is true only because arg is constrained to the range $[0, 2\pi)$ — otherwise the definition of equality would need to allow for two complex numbers that have the same radius but angles that differ by a multiple of 2π. Likewise, if arg were not constrained to be 0 when the radius is 0, then the definition of equality would need to allow for two complex number that both have radius 0, but different angles.

So far, there is no connection between the two sets of observers. Now some mutual constraints are introduced — $\forall C$: Complex:

$$\begin{aligned} \mathsf{real}(C) &= \mathsf{rad}(C) \times \cos(\mathsf{arg}(C)) \\ \mathsf{imag}(C) &= \mathsf{rad}(C) \times \sin(\mathsf{arg}(C)) \\ \mathsf{rad}(C) &= \sqrt{\mathsf{real}(C)^2 + \mathsf{imag}(C)^2} \\ \mathsf{arg}(C) &= \tan^{-1}(\mathsf{imag}(C), \mathsf{real}(C)) \end{aligned}$$

(The final constraint assumes that the inverse tan function returns 0 when both of its arguments are 0.)

The basic arithmetic operations can now be defined:

- Addition, $\mathsf{Complex} \times \mathsf{Complex} \to \mathsf{Complex}$:

$$\begin{array}{rrl} \forall C_1, C_2 : \mathsf{Complex} \cdot & \mathsf{real}(C_1 + C_2) & = \mathsf{real}(C_1) + \mathsf{real}(C_2) \\ \wedge & \mathsf{imag}(C_1 + C_2) & = \mathsf{imag}(C_1) + \mathsf{imag}(C_2) \ . \end{array}$$

- Subtraction, $\mathsf{Complex} \times \mathsf{Complex} \to \mathsf{Complex}$:

$$\begin{array}{rrl} \forall C_1, C_2 : \mathsf{Complex} \cdot & \mathsf{real}(C_1 - C_2) & = \mathsf{real}(C_1) - \mathsf{real}(C_2) \\ \wedge & \mathsf{imag}(C_1 - C_2) & = \mathsf{imag}(C_1) - \mathsf{imag}(C_2) \ . \end{array}$$

- Product, $\mathsf{Complex} \times \mathsf{Complex} \to \mathsf{Complex}$:

$$\begin{array}{rrl} \forall C_1, C_2 : \mathsf{Complex} \cdot & \mathsf{real}(C_1 \times C_2) & = \mathsf{real}(C_1) \times \mathsf{real}(C_2) - \mathsf{imag}(C_1) \times \mathsf{imag}(C_2) \\ \wedge & \mathsf{imag}(C_1 \times C_2) & = \mathsf{real}(C_1) \times \mathsf{imag}(C_2) + \mathsf{imag}(C_1) \times \mathsf{real}(C_2) \ . \end{array}$$

- Division, $\mathsf{Complex} \times \mathsf{Complex} \to \mathsf{Complex}$:

$$\begin{array}{rrl} \forall C_1, C_2 : \mathsf{Complex} \cdot & \mathsf{rad}(C_1/C_2) & = \mathsf{rad}(C_1)/\,\mathsf{rad}(C_2) \\ \wedge & \mathsf{arg}(C_1/C_2) & = \mathsf{arg}(C_1) - \mathsf{arg}(C_2) \end{array}$$

  where the radius of the second parameter cannot be 0.

2.2 A Concrete Representation: Cartesian

The preceding section did not define a concrete representation for Complex — it defined observers, which should be implemented regardless of the representation, and it defined arithmetic operations in terms of the observers.

In this section, the standard Cartesian representation, based on the real and imaginary parts, is defined:

$$\mathsf{Cartesian} = \{\mathsf{rl} : \mathsf{Real}, \mathsf{im} : \mathsf{Real}\} \ .$$

This definition means that the concrete representation Cartesian has two fields, of type Real, called 'rl' and 'im'. Access to the fields is indicated using '.' notation: for an instance C of Cartesian, $C.\mathsf{rl}$ and $C.\mathsf{im}$ give the values of the fields.

For a concrete representation, a constructor is needed to produce instances, *viz.* $\mathsf{mkCartesian} : \mathsf{Real} \times \mathsf{Real} \to \mathsf{Cartesian}$:

$$\mathsf{mkCartesian}(r, i) = \{\mathsf{rl} = r, \mathsf{im} = i\} \ .$$

This definition means that mkCartesian creates an instance of Cartesian that has the value r in its field rl and the value i in its field im.

The real and imag observers are trivially defined on this representation:

$$\begin{array}{rcl} \mathsf{real}(C : \mathsf{Cartesian}) & = & C.\mathsf{rl} \\ \mathsf{imag}(C : \mathsf{Cartesian}) & = & C.\mathsf{im} \ . \end{array}$$

Definitions for the other observers, rad and arg, follow from the constraints on the observers:

$$\begin{array}{rcccl} \mathsf{rad}(C : \mathsf{Cartesian}) & = & \sqrt{\mathsf{real}(C)^2 + \mathsf{imag}(C)^2} & = & \sqrt{C.\mathsf{rl}^2 + C.\mathsf{im}^2} \\ \mathsf{arg}(C : \mathsf{Cartesian}) & = & \tan^{-1}(\mathsf{imag}(C), \mathsf{real}(C)) & = & \tan^{-1}(C.\mathsf{im}, C.\mathsf{rl}) \ . \end{array}$$

Since the arithmetic operations are based solely on the observers, their definitions can be carried over from Complex to Cartesian without modification other than renaming the type (from

Complex to Cartesian). Optionally, the division operation can be defined directly in terms of the real and imaginary parts instead of the radius and argument:

$$\begin{array}{rrcl} \forall C_1, C_2 : \mathsf{Complex} \cdot & \mathsf{real}(C_1/C_2) & = & [\mathsf{real}(C_1) \times \mathsf{real}(C_2) + \mathsf{imag}(C_1) \times \mathsf{imag}(C_2)]/r_2^2 \\ \wedge & \mathsf{imag}(C_1/C_2) & = & [\mathsf{imag}(C_1) \times \mathsf{real}(C_2) - \mathsf{real}(C_1) \times \mathsf{imag}(C_2)]/r_2^2 \\ \text{where} & r_2 & = & \sqrt{\mathsf{real}(C_2)^2 + \mathsf{imag}(C_2)^2}\ . \end{array}$$

Since all of the operations now have definitions for their real and imaginary parts, they can be used to construct instances of Cartesian. For example, the above definition of division is straightforward to convert into a call to the mkCartesian constructor:

$$\begin{array}{rl} C_1/C_2 & = \\ & \mathsf{let}\ r_2 = \sqrt{(\ \mathsf{real}(C_2)^2 + \mathsf{imag}(C_2)^2)}\ \mathsf{in} \\ & \mathsf{let}\ r = (\mathsf{real}(C_1) \times \mathsf{real}(C_2) + \mathsf{imag}(C_1) \times \mathsf{imag}(C_2))/r_2^2, \\ & \mathsf{let}\ i = (\mathsf{imag}(C_1) \times \mathsf{real}(C_2) - \mathsf{real}(C_1) \times \mathsf{imag}(C_2))/r_2^2\ \mathsf{in} \\ & \mathsf{mkCartesian}(r, i) \end{array}$$

All of the operations are now defined in fully executable form on Cartesian, and this concrete representation can be used to carry out computations defined in terms of Complex.

2.3 An Alternative Concrete Representation: Polar

An alternative representation for Complex is Polar, which has a field for the radius and a field for the argument:

$$\mathsf{Polar} = \{\mathsf{rd} : \mathsf{Real}, \mathsf{ar} : \mathsf{Real}\}$$

where the following constraints apply:

- $\mathsf{rd} \geq 0$ — that is, the radius cannot be negative;
- $0 \leq \mathsf{ar} < 2\pi$ — that is, the argument is normalized into the range $[0, 2\pi)$;
- $\mathsf{rd} = 0 \implies \mathsf{ar} = 0$ — that is, if the radius is zero, then so too is the argument.

These constraints directly correspond to the constraints imposed on the rad and arg observers in Complex.

As with Cartesian, two observers correspond directly with the fields, and so are trivially realized:

$$\begin{array}{rcl} \mathsf{rad}(C : \mathsf{Polar}) & = & C.\mathsf{rd} \\ \mathsf{arg}(C : \mathsf{Polar}) & = & C.\mathsf{ar}\ . \end{array}$$

A constructor is defined, *viz.* $\mathsf{mkPolar} : \mathsf{Real} \times \mathsf{Real} \to \mathsf{Polar}$:

$$\mathsf{mkPolar}(r, a) = \{\mathsf{rd} = r, \mathsf{ar} = a\}\ .$$

The other observers follow from the constraints defined in Complex:

$$\begin{array}{rcccl} \mathsf{real}(C : \mathsf{Polar}) & = & \mathsf{rad}(C) \times \cos(\mathsf{arg}(C)) & = & C.\mathsf{rd} \times \cos(C.\mathsf{ar}) \\ \mathsf{imag}(C : \mathsf{Polar}) & = & \mathsf{rad}(C) \times \sin(\mathsf{arg}(C)) & = & C.\mathsf{rd} \times \sin(C.\mathsf{ar})\ . \end{array}$$

As was the case for Cartesian, the arithmetic operations can be carried over as is to operate on Polar. However, it may be preferable to give an alternative definition for the product operation that should be more efficient than the definition given for the ADT:

$$\begin{array}{rrcl} \forall C_1, C_2 : \mathsf{Polar} \cdot & \mathsf{rad}(C_1 \times C_2) & = & \mathsf{rad}(C_1) \times \mathsf{rad}(C_2) \\ \wedge & \mathsf{arg}(C_1 \times C_2) & = & \mathsf{arg}(C_1) + \mathsf{arg}(C_2) \mod 2\pi\ . \end{array}$$

Of course, this alternative definition is constrained to produce the same results as the original definition.

The concrete representation Polar can now be used to carry out computations defined using the Complex abstract data type.

3 A Simple Application: Solving Quadratic Equations

As a simple example of an 'application', consider a solver of quadratic equations. Given an equation in the form $ax^2 + bx + c = 0$, where the coefficients a, b and c are complex and a is not zero, there are two (possibly equal) solutions for x, given by $x = -b \pm \sqrt{b^2 - 4ac}/2a$.

The complex square root can be defined using:

$$\forall C : \mathsf{Complex} \cdot (\sqrt{C})^2 = C \ .$$

Note that this does not uniquely define the square root, since if $x^2 = C$ then $(-x)^2 = C$.

For the Polar representation, the complex square root can be given a more computation-oriented definition:

$$\begin{array}{rrcl} \forall C : \mathsf{Polar} \cdot & \mathsf{rad}(\sqrt{C}) & = & \sqrt{\mathsf{rad}(C)} \\ \wedge & \mathsf{arg}(\sqrt{C}) & = & \mathsf{arg}(C)/2 \ . \end{array}$$

This uniquely defines the complex square root, assuming the usual convention that on Real numbers $\sqrt{}$ denotes the positive square root.

Likewise for the Cartesian representation [3]:

$$\begin{array}{rrcl} \forall C : \mathsf{Cartesian} \cdot & \mathsf{real}(\sqrt{C}) & = & \frac{\sqrt{2}}{2}\sqrt{r + \mathsf{real}(C)} \\ \wedge & \mathsf{imag}(\sqrt{C}) & = & \mathsf{sign}(\mathsf{imag}(C))\frac{\sqrt{2}}{2}\sqrt{r - \mathsf{real}(C)} \\ \text{where} & r & = & \sqrt{\mathsf{real}(C)^2 + \mathsf{imag}(C)^2} \ . \end{array}$$

Given these definitions, the solver application can be implemented using either the Cartesian or the Polar representation.

4 Generating Run-time Monitors

The definition of Complex contains various semantic constraints — e.g., each of the arithmetic operations constrains its output with respect to its inputs, and for the divide operation, the second argument (the divisor) cannot be zero. In addition, the Polar representation constrains its fields that represent the radius and argument.

Moreover, each concrete representation is constrained by the abstract data type that it represents. For example, the definition of the product operation in Polar may differ from the definition in Complex, but the former must conform to the latter. Likewise for the definition of the division operator in Cartesian compared with the definition in Complex.

Furthermore, the constructive definitions of square root in Cartesian and Polar must conform to the non-constructive definition in Complex (i.e., the result squared must equal the argument).

Many such semantic constraints can be used to generate monitors that check the constraints at run-time. For example:

- A constraint on an operation's arguments can be checked when computation of the operation begins. For example, the complex division operation can check that the divisor is non-zero.
- If the Polar representation is used, then the fields can be checked against their constraints: e.g., that the radius is non-negative and that the argument is in the range $[0, 2\pi)$.
- A constraint on an operation's result can be checked when computation of the operation completes. For example, the operation to solve a quadratic equation produces two supposed solutions; the equation can be evaluated and confirmed for each of the solutions.
- A computation using a concrete representation can be checked against the definition in the abstract data type. For example, the computation of a square root using Cartesian can be checked against the definition in Complex (i.e., the result squared should equal the original value).

There are several caveats: a constraint may quantify over a domain that cannot be effectively checked (e.g., all functions or all integers); a constraint may involve real numbers which are only approximately represented in a computation (e.g., as finite-precision floating point numbers), so different algorithms that are supposed to be equivalent might produce slightly different values.

In addition, constraints involving large data structures may be checkable, but incur significant overhead. For example, a Set of Integers may be represented by a sorted List. In principle, every operation on the sorted List could check that the List is indeed sorted, to try to detect corruption of the List. However, if the List is long, then the time required to repeatedly check may be too high.

To reduce overhead, monitoring can be performed using sampling techniques. For example: an operation's arguments can be checked at random, with some suitable probability; or some small subset of the elements in a supposedly sorted List can be selected for comparison with neighbouring elements.

4.1 Augmenting a Representation to Enhance Monitoring

Unlike the Polar representation, the Cartesian representation is internally unconstrained: no matter what real numbers the real and imaginary fields contain, they together represent some complex number. Consequently, useful monitors cannot be generated from the representation *per se*.

However, a representation can be augmented with additional fields that are derived from the original fields, and that thus introduce semantic constraints. For example, the Cartesian representation can be augmented with an additional field that stores the radius of the complex number:

$$\begin{aligned} \mathsf{CartesianWithRadius} &= \{\mathsf{rl} : \mathsf{Real}, \mathsf{im} : \mathsf{Real}, \mathsf{rd} : \mathsf{Real}\} \\ \text{where } \mathsf{rd} &= \sqrt{\mathsf{rl}^2 + \mathsf{im}^2}\ . \end{aligned}$$

The added semantic constraint can be used to check the integrity of instances of the augmented type, CartesianWithRadius. The added field may also be used to simplify computations: for example, the rad operation can now be optimized to return the value of the rd field instead of calculating the radius from the real and imaginary parts.

Of course, any added fields must be properly initialized when an instance of the representation is constructed. Code to do this can be automatically generated.

Adding the rd field is an example of a domain-specific augmentation. Other examples include adding a field to record the sum of the elements in a numeric List (the sum being incrementally maintained as elements are added or removed), or adding a field to record the minimum and maximum elements in a List of words.

In contrast, a generic augmentation is adding a field to record some hash code of the other fields. For example:

$$\begin{aligned} \mathsf{CartesianWithHash} &= \{\mathsf{rl} : \mathsf{Real}, \mathsf{im} : \mathsf{Real}, \mathsf{h} : \mathsf{Integer}\} \\ \text{where } \mathsf{h} &= \mathsf{hash}(\mathsf{rl}, \mathsf{im}) \ . \end{aligned}$$

With this augmentation, run-time monitors can check the validity of instances by evaluating hash(rl, im) and comparing the result with that stored in the field h.

Of course, a hash field can be added to representations that already have semantic constraints, such as Polar.

4.2 Using Multiple Representations for Cross-Checking

In addition to augmenting a single representation, run-time monitoring can be enhanced by using multiple representations simultaneously and cross-checking the representations.

For example, given the Cartesian and Polar representations for Complex, a cross-checking representation, ComplexCX can be generated that incorporates both representations:

$$\begin{aligned} \mathsf{ComplexCX} &= \{\mathsf{ct} : \mathsf{Cartesian}, \mathsf{pl} : \mathsf{Polar}\} \\ \text{where} &\quad \mathsf{ct} = \mathsf{pl} \ . \end{aligned}$$

The equality of the ct and pl fields can be defined using any canonical set of observers (e.g., real and imag).

For each operation that is defined on Complex, an operation can be generated on ComplexCX that computes the operation using both the Cartesian and Polar representations, and then confirms that they agree. For example, consider the product operation:

$$\begin{aligned} \forall C_1, C_2 : \mathsf{ComplexCX} \cdot \quad & C_1 \times C_2 = \\ & \quad \textit{compute the product using the Cartesian representations} \\ & \mathsf{let}\ car = C_1.ct \times C_2.ct, \\ & \quad \textit{compute the product using the Polar representations} \\ & \mathsf{let}\ pol = C_1.pl \times C_2.pl, \\ & \quad \textit{check the two products against each other} \\ & \mathsf{in\ if}\ car = pol\ \mathsf{then\ mkComplexCX}(car, pol)\ \mathsf{else\ error} \end{aligned}$$

where error represents some value that denotes an error condition.

5 Diversifying an Application

Given:

- an application that uses some abstract data type D: $A(D)$;
- and multiple representations of D: R_i, for $i = 1, \ldots, n$;

multiple implementations $A(R_i)$ of the application can be generated, one for each representation.

For example, if $Q(\mathsf{Complex})$ is the quadratic equation solver discussed above, then possible implementations include:

- $Q(\mathsf{Cartesian})$,
- $Q(\mathsf{Polar})$,
- $Q(\mathsf{CartesianWithRadius})$,
- $Q(\mathsf{CartesianWithHash})$,
- $Q(\mathsf{PolarWithHash})$, and
- $Q(\mathsf{ComplexCX})$.

Clearly there are other representations of Complex that could be used to extend this list of implementations.

If an application uses multiple abstract data types, each with multiple representations, then each *combination* of one representation per abstract data type may generate an implementation. So if there are n ADTs, with ADT j having r_j implementations ($j = 1, \ldots, n$), then $\prod_{j=1,\ldots,n} r_j$ implementations may be generated.[1]

5.1 Example: Clustering

This section considers a somewhat larger application for diversification: *clustering*. Clustering is a machine-learning technique for grouping points in some space into groups (clusters), such that points in the same group are close together, while points in different groups are well-separated.

There are several widely used algorithms for clustering, two of which are considered here:

- *k-Means* [4] (also known as LLoyd's algorithm) is an iterative technique in which each point is assigned to the nearest of a set of k cluster centres, and then the points assigned to each centre are averaged to update the centre. This is repeated until the centres stop changing. The initial centres may be generated randomly.

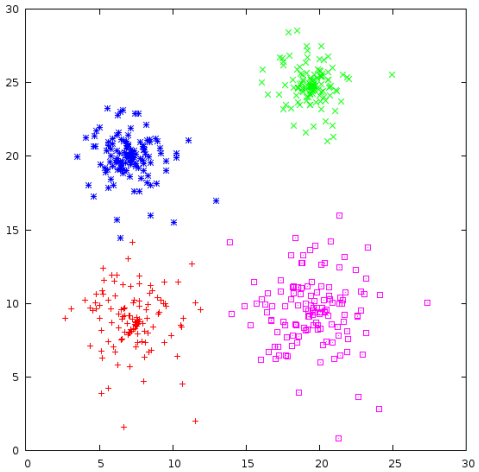

Figure 2: k-Means clusters

[1] This assumes that representations do not themselves use abstract data types. Counting the number of implementations is more complex if this is not the case. Nonetheless, the essential notion still holds that multiple ADTs, each with multiple representations, generate a combinatorial number of implementations.

- *DBSCAN* (*density-based spatial clustering of applications with noise*) [5] determines if a point is part of a cluster based on the number of points within a certain distance, merging overlapping clusters together.

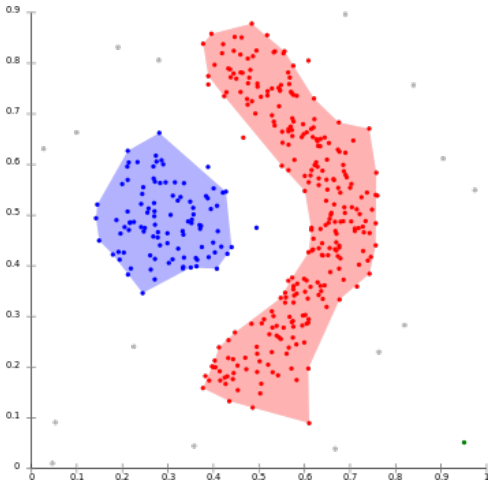

Figure 3: DBSCAN clusters (from WikiPedia)

k-Means is well suited to data sets that are generated by Gaussian processes, whereas DBSCAN is perhaps better for clusters that may not be symmetric.

Figure 4 shows possible diversified implementations of a clustering application.

- The first level of diversification uses either k-Means or DBSCAN.
- The algorithms use various ADTs. One is Sequence, which may be represented using either Tree or Array.
- Another ADT is Set, which may be represented using Hash Table or Unordered List.
- The Ordered Set ADT may be represented using Ordered Tree or Ordered List.
- Finally, the algorithms use contiguous blocks of coordinate spaces, which may be represented as Bounding Boxes (specified as the 'lower-left' corner and the 'upper-right' corner) or as Products of Intervals (with one interval per spatial dimension).

These possibilities, which are certainly not exhaustive, combined give 48 implementations.

5.2 Spatial and Temporal Diversity

Given multiple implementations of an application, the implementations can be deployed one at a time or simultaneously.

- Deploying the implementations sequentially, on a single computer, may be referred to as *temporal* diversity. The transfer from one implementation to another may be triggered when a run-time monitor detects a problem, or periodically, or when the cumulative activity reaches some threshold, etc.
- Deploying multiple implementations simultaneously to multiple computers may be referred to as *spatial* diversity.

Of course, a deployment scheme may use a mixture of both temporal and spatial diversity.

Version	Clustering Algorithm	Represenation of Sequence	Represenation of Set	Represenation of Ordered Set	Represenation of *n*-Dimensional Space
version 1	CentroidClustering	Tree	HashTable	OrderedTree	BoundingBox
version 2	CentroidClustering	Tree	HashTable	OrderedTree	ProductOfIntervals
version 3	CentroidClustering	Tree	HashTable	OrderedList	BoundingBox
version 4	CentroidClustering	Tree	HashTable	OrderedList	ProductOfIntervals
version 5	CentroidClustering	Tree	HashTable	OrderedArray	BoundingBox
version 6	CentroidClustering	Tree	HashTable	OrderedArray	ProductOfIntervals
version 7	CentroidClustering	Tree	UnorderedList	OrderedTree	BoundingBox
version 8	CentroidClustering	Tree	UnorderedList	OrderedTree	ProductOfIntervals
version 9	CentroidClustering	Tree	UnorderedList	OrderedList	BoundingBox
version 10	CentroidClustering	Tree	UnorderedList	OrderedList	ProductOfIntervals
version 11	CentroidClustering	Tree	UnorderedList	OrderedArray	BoundingBox
version 12	CentroidClustering	Tree	UnorderedList	OrderedArray	ProductOfIntervals
version 13	CentroidClustering	Array	HashTable	OrderedTree	BoundingBox
version 14	CentroidClustering	Array	HashTable	OrderedTree	ProductOfIntervals
version 15	CentroidClustering	Array	HashTable	OrderedList	BoundingBox
version 16	CentroidClustering	Array	HashTable	OrderedList	ProductOfIntervals
version 17	CentroidClustering	Array	HashTable	OrderedArray	BoundingBox
version 18	CentroidClustering	Array	HashTable	OrderedArray	ProductOfIntervals
version 19	CentroidClustering	Array	UnorderedList	OrderedTree	BoundingBox
version 20	CentroidClustering	Array	UnorderedList	OrderedTree	ProductOfIntervals
version 21	CentroidClustering	Array	UnorderedList	OrderedList	BoundingBox
version 22	CentroidClustering	Array	UnorderedList	OrderedList	ProductOfIntervals
version 23	CentroidClustering	Array	UnorderedList	OrderedArray	BoundingBox
version 24	CentroidClustering	Array	UnorderedList	OrderedArray	ProductOfIntervals
version 25	DensityClustering	Tree	HashTable	OrderedTree	BoundingBox
version 26	DensityClustering	Tree	HashTable	OrderedTree	ProductOfIntervals
version 27	DensityClustering	Tree	HashTable	OrderedList	BoundingBox
version 28	DensityClustering	Tree	HashTable	OrderedList	ProductOfIntervals
version 29	DensityClustering	Tree	HashTable	OrderedArray	BoundingBox
version 30	DensityClustering	Tree	HashTable	OrderedArray	ProductOfIntervals
version 31	DensityClustering	Tree	UnorderedList	OrderedTree	BoundingBox
version 32	DensityClustering	Tree	UnorderedList	OrderedTree	ProductOfIntervals
version 33	DensityClustering	Tree	UnorderedList	OrderedList	BoundingBox
version 34	DensityClustering	Tree	UnorderedList	OrderedList	ProductOfIntervals
version 35	DensityClustering	Tree	UnorderedList	OrderedArray	BoundingBox
version 36	DensityClustering	Tree	UnorderedList	OrderedArray	ProductOfIntervals
version 37	DensityClustering	Array	HashTable	OrderedTree	BoundingBox
version 38	DensityClustering	Array	HashTable	OrderedTree	ProductOfIntervals
version 39	DensityClustering	Array	HashTable	OrderedList	BoundingBox
version 40	DensityClustering	Array	HashTable	OrderedList	ProductOfIntervals
version 41	DensityClustering	Array	HashTable	OrderedArray	BoundingBox
version 42	DensityClustering	Array	HashTable	OrderedArray	ProductOfIntervals
version 43	DensityClustering	Array	UnorderedList	OrderedTree	BoundingBox
version 44	DensityClustering	Array	UnorderedList	OrderedTree	ProductOfIntervals
version 45	DensityClustering	Array	UnorderedList	OrderedList	BoundingBox
version 46	DensityClustering	Array	UnorderedList	OrderedList	ProductOfIntervals
version 47	DensityClustering	Array	UnorderedList	OrderedArray	BoundingBox
version 48	DensityClustering	Array	UnorderedList	OrderedArray	ProductOfIntervals

Figure 4: Diversified implementations of clustering

5.3 Translation between Representations

If the application requires communication between concurrent instances, then it is likely that two different implementations will need to exchange data. Since the implementations are using different representations, some form of translation is required. For example, an implementation using the Cartesian representation may need to send complex numbers to an implementation using the Polar representation.

Depending on how the implementations are communicating (e.g., shared memory vs. network), then either a direct translation (Cartesian to Polar) or a translation via an intermediate form (e.g., Cartesian serialized to bit sequence and then deserialized to Polar) may be required.

Direct translation is straightforward if all of the representations have a common constructor and a common set of observers. These may be defined in the abstract data type, though implemented differently in each representation.

For Complex, two pairs of canonical observers were defined above: real and imag form one pair, and rad and arg the other pair. Suppose that the former pair is chosen for translation purposes.

No canonical constructor was defined for Complex, but one is simple to add, matching the chosen canonical observers — say $\mathsf{mkComplex} : \mathsf{Real} \times \mathsf{Real} \to \mathsf{Complex}$:

$$\begin{array}{rrcl} \forall r, i : \mathsf{Real} \cdot & \mathsf{real}(\mathsf{mkComplex}(r, i)) & = & r \\ \wedge & \mathsf{imag}(\mathsf{mkComplex}(r, i)) & = & i \end{array}$$

All of the representations for Complex already implement the chosen observers.

For the Cartesian representation, the mkComplex constructor is trivial to express using the native mkCartesian constructor.

For the Polar representation, the implementation of mkComplex needs to compute the radius and argument from the real and imaginary parts before calling the native mkPolar constructor.

In general, when implementation R_1 needs to send a complex number to implementation R_2, R_1 writes into memory the values returned by the chosen set of canonical observers, e.g., real and imag; R_2 then reads these values from memory and passes them to its implementation of the canonical constructor, mkComplex.

Translation through serialization can be treated similarly, with canonical serialization and deserialization operators.

6 Recovering from Corrupted Data or Errors

When a runtime monitor discovers a violation of a semantic constraint (e.g., a data type invariant or a pre- or post-condition), it raises an error. There are several ways in which the error can be handled:

- The error can be allowed to terminate the application. This might be more acceptable in some deployment contexts than allowing the application to proceed with corrupt or erroneous data.
- If the violation occurs during the computation of an operation (for example, one of the computation's intermediate values has a violated invariant, or the operation's post-condition is violated), then:

  - The computation can be retried, in case the violation was the result of a transient problem (e.g., a race condition, a time out or a hardware glitch).
  - The representation in which the error arose (e.g., CartesianWithHash) can be replaced with a different representation (e.g., Polar) and the computation restarted.
  - If the representation is redundant (e.g., ComplexCX) then the erroneous representation can be reconstructed from the sound representations and the computation allowed to proceed.
- If the error is a data type violation, then the data can be repaired and the computation allowed to proceed — see below.

In each of the above cases, code can be automatically generated to catch and handle the error.

6.1 Data Repairs

When a violation of a data type invariant is detected, it may be possible to *repair* the data such that the repaired data has no violations. For example, a value of type Polar has three constraints:

1. The radius must be non-negative.
2. The argument must be in the range $[0, 2\pi)$.
3. If the radius is 0, then the argument must be 0.

Suppose the radius is detected to be negative. There are several possible repairs:

- The radius's sign can be flipped.
- The radius can be given the closest value (according to some metric) that eliminates the violation. In this case, the closest value may be 0. However, using that value may introduce a violation of constraint #3, so perhaps some small, positive value might be used.
- The complex number could be given some value that is known, in context, to be safe.

Likewise, if the argument is outside its valid range, it could be snapped to the closest value (0 if it is negative, or 2π if it is too large) or it could be mapped into the valid range modulo 2π. And if the radius is 0, then a non-zero argument could be made 0.

Without further information about the cause of the constraint violation, it is generally not possible to determine which repair is best. However, choosing some repair may be safer than allowing the application to continue with data that is corrupt.

Some repairs are generic (e.g., ensuring that a number is within some range) and can be automatically introduced. Other repairs, such as using a value that is safe in context, are application-specific; nonetheless, they can be stated declaratively and code generated to make use of them.

7 Conclusion

This report gives a high-level overview of how multiple representations of a data type can be used to generate diversified implementations of an application. It illustrates how run-time monitors can be generated from semantic constraints in the abstract data type and the representations.

It illustrates how representations can be augmented to enhance monitoring. Finally, it shows how data can be repaired when monitoring detects corruption.

The synthetic diversity discussed in this report lies on a spectrum of diversification techniques ranging from instruction set randomization and variable layout randomization to the use of multiple protocols and architectures — see Figure 5.

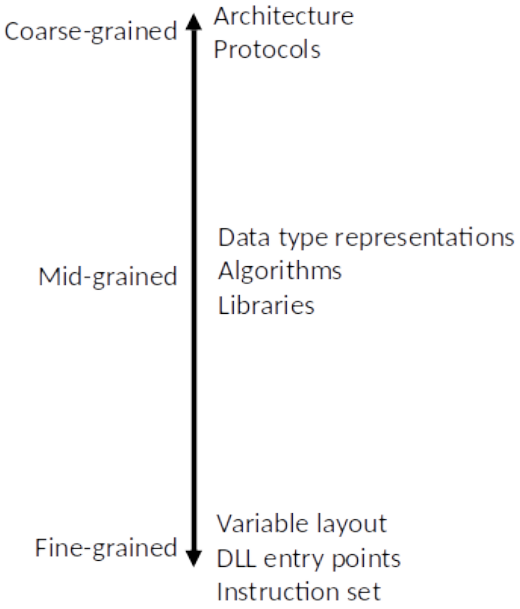

Figure 5: A spectrum of diversification techniques

References

[1] *Algorithmic Complexity Attack*.
`https://en.wikipedia.org/wiki/Algorithmic_complexity_attack`

[2] Stephen Fitzpatrick, Cordell Green, Stephen Westfold and James McDonald. *Using Software Generation and Repair for Cyber-defense*. Report #AFRL-RI-RS-TR-2014-111, Final Technical Report, May 2014, Kestrel Institute.
`https://apps.dtic.mil/dtic/tr/fulltext/u2/a599889.pdf`

[3] Eric W. Weisstein. *Square Root*. From MathWorld — A Wolfram Web Resource.
`http://mathworld.wolfram.com/SquareRoot.html`

[4] *k-Means Clustering*
`https://en.wikipedia.org/wiki/K-means_clustering`

[5] *DBSCAN*
`https://en.wikipedia.org/wiki/DBSCAN`