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 information contained in abstract data types and their implementations.

1 Introduction

A data type such as Set can be defined abstractly, i.e., without regard to how it is represented. For example, two Sets are defined to be equal when they have the same members (regardless of how those members are associated with each Set), and inserting an element into a Set is defined in terms of how it affects membership.

An abstract data type (ADT) must be given some representation if it is to be used in computations. There may be several possible representations for any given ADT. For example, a Set may be represented using a List, with some representations allowing repetitions and others not. If the elements are ordered (e.g., Integers), then the List representations may be sorted — alternatively, a Tree representation may be used. Or a Hash Map may be used (with elements stored in buckets according to their hash values).

In the following, the use of alternative representations for diversity is considered — that is, given an application that uses an ADT, an implementation of the application is generated for each of the possible representations of that ADT. (Of course, when multiple ADTs are used, their various representations can be combined in an implementation.)

In addition, the generation of run-time monitors is considered. The run-time monitors are based on the semantic constraints contained in ADTs (e.g., pre- and post-conditions on their operations), invariants of the concrete representations (e.g., that a List representing a Set should contain no 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 to try to recover, by repairing corrupt data or using an alternative representation to repeat the 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;
- bugs in an algorithm that is 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, 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 can 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, when the intersection of two sets is computed using the representation SortedList, the result must conform to the definition of intersection given in the Set ADT.

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:

- $\text{real} : \text{Complex} \rightarrow \text{Real}$, which gives the real part of the Complex Number;
- $\text{imag} : \text{Complex} \rightarrow \text{Real}$, which gives the imaginary part;
- $\text{rad} : \text{Complex} \rightarrow \text{Real}$, which gives the radius; i.e., the magnitude, constrained to be non-negative;
- $\text{arg} : \text{Complex} \rightarrow \text{Real}$, 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:

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

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.

\[ \forall C_1, C_2 : \text{Complex} \cdot C_1 = C_2 \iff \text{real}(C_1) = \text{real}(C_2) \land \text{imag}(C_1) = \text{imag}(C_2) \]

Likewise, rad and arg are also canonical:

\[ \forall C_1, C_2 : \text{Complex} \cdot C_1 = C_2 \iff \text{rad}(C_1) = \text{rad}(C_2) \land \text{arg}(C_1) = \text{arg}(C_2) \]

(Note that this would not be true if where not constrained to be 0 when rad is 0.)

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

\[
\begin{align*}
\text{real}(C) &= \text{rad}(C) \times \cos(\text{arg}(C)) \\
\text{imag}(C) &= \text{rad}(C) \times \sin(\text{arg}(C)) \\
\text{rad}(C)^2 &= \text{real}(C)^2 + \text{imag}(C)^2 \\
\text{arg}(C) &= \tan^{-1}(\text{imag}(C), \text{real}(C))
\end{align*}
\]

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

The basic arithmetic operations can now be defined:

- **Addition**, Complex \( \times \) Complex \( \rightarrow \) Complex:
  \[
  \forall C_1, C_2 : \text{Complex} \cdot \text{real}(C_1 + C_2) = \text{real}(C_1) + \text{real}(C_2) \\
  \land \text{imag}(C_1 + C_2) = \text{imag}(C_1) + \text{imag}(C_2).
  \]

- **Subtraction**, Complex \( \times \) Complex \( \rightarrow \) Complex:
  \[
  \forall C_1, C_2 : \text{Complex} \cdot \text{real}(C_1 - C_2) = \text{real}(C_1) - \text{real}(C_2) \\
  \land \text{imag}(C_1 - C_2) = \text{imag}(C_1) - \text{imag}(C_2).
  \]

- **Product**, Complex \( \times \) Complex \( \rightarrow \) Complex:
  \[
  \forall C_1, C_2 : \text{Complex} \cdot \text{real}(C_1 \times C_2) = \text{real}(C_1) \times \text{real}(C_2) - \text{imag}(C_1) \times \text{imag}(C_1) \\
  \land \text{imag}(C_1 \times C_2) = \text{real}(C_1) \times \text{imag}(C_2) + \text{imag}(C_1) \times \text{real}(C_2).
  \]

- **Division**, Complex \( \times \) Complex \( \rightarrow \) Complex:
  \[
  \forall C_1, C_2 : \text{Complex} \cdot \text{real}(C_1/C_2) = \text{rad}(C_1)/\text{rad}(C_2) \\
  \land \text{arg}(C_1/C_2) = \text{arg}(C_1) - \text{arg}(C_2).
  \]

where the second parameter cannot be zero (i.e., the Complex with real and imaginary parts both 0).

### 2.2 A Concrete Representation: Cartesian

In the preceding section, no concrete representation was defined. In this section, the standard Cartesian representation, based on the real and imaginary parts, is defined:

\[
\text{Cartesian} = \{ \text{rl} : \text{Real}, \text{im} : \text{Real} \}.
\]
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: \( C.\text{rl} \) and \( C.\text{im} \). Thus, the real and imag observers are trivially defined on this representation:

\[
\begin{align*}
\text{real}(C : \text{Cartesian}) & = C.\text{rl} \\
\text{imag}(C : \text{Cartesian}) & = C.\text{im} .
\end{align*}
\]

For a concrete representation, a constructor is needed to produce instances, viz. \( \text{mkCartesian} : \text{Real} \times \text{Real} \to \text{Cartesian} : \)

\[
\text{mkCartesian}(r, i) = \{ \text{rl} = r, \text{im} = i \} .
\]

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

\[
\begin{align*}
\text{rad}(C : \text{Cartesian}) & = \sqrt{\text{real}(C)^2 + \text{imag}(C)^2} \\
\text{arg}(C : \text{Cartesian}) & = \tan^{-1}(\text{imag}(C), \text{real}(C)) .
\end{align*}
\]

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:

\[
\forall C_1, C_2 : \text{Complex} \cdot \text{real}(C_1/C_2) = \frac{[\text{real}(C_1) \times \text{real}(C_2) + \text{imag}(C_1) \times \text{imag}(C_2)]/r_2^2}{\text{real}(C_2)^2 + \text{imag}(C_2)^2} \\
\land \text{imag}(C_1/C_2) = \frac{[\text{imag}(C_1) \times \text{real}(C_2) - \text{real}(C_1) \times \text{imag}(C_2)]/r_2^2}{\text{real}(C_2)^2 + \text{imag}(C_2)^2} .
\]

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 \( \text{mkCartesian} \) constructor:

\[
\begin{align*}
C_1/C_2 & = \text{let } r_2^2 = \text{rad}(C_2)^2 \text{ in} \\
& \text{let } \text{pr} = (\text{real}(C_1) \times \text{real}(C_2) + \text{imag}(C_1) \times \text{imag}(C_2))/r_2^2, \\
& \text{let } \text{pi} = (\text{imag}(C_1) \times \text{real}(C_2) - \text{real}(C_1) \times \text{imag}(C_2))/r_2^2 \text{ in} \\
& \text{mkCartesian}(\text{pr}, \text{pi})
\end{align*}
\]

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 using 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:

\[
\text{Polar} = \{ \text{rd} : \text{Real}, \text{ar} : \text{Real} \}
\]

where the following constraints apply:

- \( \text{rd} \geq 0 \) — that is, the radius cannot be negative;
- \( 0 \leq \text{ar} < 2\pi \) — that is, the argument is normalized into the range \([0, 2\pi)\);
- \( \text{rd} = 0 \implies \text{ar} = 0 \) — 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:

\[
\begin{align*}
\text{rad}(C : \text{Cartesian}) &= C.\text{rd} \\
\text{arg}(C : \text{Cartesian}) &= C.\text{ar}.
\end{align*}
\]

A constructor is defined, viz. \text{mkPolar} : \text{Real} \times \text{Real} \rightarrow \text{Polar}:

\[
\text{mkPolar}(r, a) = \{ \text{rd} = r, \text{ar} = a \}.
\]

The other observers follow from the constraints defined in Complex:

\[
\begin{align*}
\text{real}(C : \text{Cartesian}) &= \text{rad}(C) \times \cos(\text{arg}(C)) \\
\text{imag}(C : \text{Cartesian}) &= \text{rad}(C) \times \sin(\text{arg}(C)).
\end{align*}
\]

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:

\[
\begin{align*}
\forall C_1, C_2 : \text{Polar} \cdot \text{rad}(C_1 \times C_2) &= \text{rad}(C_1) \times \text{rad}(C_2) \\
\land \quad \text{arg}(C_1 \times C_2) &= \text{arg}(C_1) + \text{arg}(C_2).
\end{align*}
\]

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

The concrete representation \text{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 \(a \neq 0\), 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:

\[
\forall C : \text{Complex} \cdot (\sqrt{C})^2 = C.
\]

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:

\[
\begin{align*}
\forall C : \text{Complex} \cdot \text{rad}(\sqrt{C}) &= \sqrt{\text{rad}(C)} \\
\land \quad \text{arg}(\sqrt{C}) &= \text{arg}(C)/2.
\end{align*}
\]

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

Likewise for the Cartesian representation[1]:

\[
\begin{align*}
\forall C : \text{Complex} \cdot \text{real}(\sqrt{C}) &= \frac{\sqrt{2}}{2} \sqrt{r + \text{real}(C)} \\
\land \quad \text{imag}(\sqrt{C}) &= \text{sign}(\text{imag}(C)) \frac{\sqrt{2}}{2} \sqrt{r - \text{real}(C)} \\
\text{where} \quad r &= \sqrt{\text{real}(C)^2 + \text{imag}(C)^2}.
\end{align*}
\]
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 computed results must satisfy the non-constructive definition).

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 for the repeated 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 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 (based on the representation itself).

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:

\[
\text{CartesianWithRadius} = \{rl : \text{Real}, \text{im} : \text{Real}, \text{rd} : \text{Real}\}
\]

where \( \text{rd} = \sqrt{\text{rl}^2 + \text{im}^2} \).

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 \( \text{rad} \) operation can now be optimized to return the value of the \( \text{rd} \) field instead of calculating the radius from the real and imaginary parts.

Of course, any added fields must be properly computed when an instance of the representation is constructed.

Adding the \( \text{rd} \) 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 the elements are added or removed), or adding a field to record the minimum and maximum elements in a List of words.

A rather generic augmentation is adding a field to record some hash code of the other fields. For example:

\[
\text{CartesianWithHash} = \{rl : \text{Real}, \text{im} : \text{Real}, h : \text{Integer}\}
\]

where \( h = \text{hash}(\text{rl}, \text{im}) \).

With this augmentation, run-time monitors can check the validity of instances by evaluating \( \text{hash}(\text{rl}, \text{im}) \) 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:

\[
\text{ComplexCX} = \{\text{ct} : \text{Cartesian}, \text{pl} : \text{Polar}\}
\]

where \( \text{ct} = \text{pl} \).

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

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

\[
\forall C_1, C_2 : \text{ComplexCX} \cdot C_1 \times C_2 =
\]

\[
\begin{align*}
\text{let } p_1 &= C_1.\text{ct} \times C_2.\text{ct}, \\
\text{let } p_2 &= C_1.\text{pl} \times C_2.\text{pl}, \\
\text{if } p_1 &= p_2 \text{ then mkComplexCX}(p_1, p_2) \text{ else error}
\end{align*}
\]
where error represents some value that denotes an error condition. Note that the computation of the product for \( p_1 \) uses the Cartesian definition while for \( p_2 \) it uses the Polar definition.

5 Diversifying an Application

Given:

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

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

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

- \( Q(\text{Cartesian}) \),
- \( Q(\text{Polar}) \),
- \( Q(\text{CartesianWithRadius}) \),
- \( Q(\text{CartesianWithHash}) \), and
- \( Q(\text{ComplexCX}) \).

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 \( i \) having \( r_i \) implementations (\( i = 1, \ldots, n \)), then \( \prod_{i=1}^{n} r_i \) implementations may be generated.

5.1 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.

---

1This 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.
5.2 Translation between Representations

If the application requires communication between its 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 \( \text{mkComplex} : \text{Real} \times \text{Real} \rightarrow \text{Complex} \):

\[
\forall r, i : \text{Real} \cdot \quad \text{real(mkComplex}(r, i)) = r \\
\land \quad \text{imag(mkComplex}(r, i)) = i
\]

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

For the Cartesian representation, the \( \text{mkComplex} \) constructor is trivial to express using the native \( \text{mkCartesian} \) constructor.

For the Polar representation, the implementation of \( \text{mkComplex} \) needs to compute the radius and argument from the real and imaginary parts before calling the native \( \text{mkPolar} \) 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 implementations of the chosen set of canonical observers, \( R_1 \), real and \( R_1 \), imag; \( R_2 \) then reads these values from memory and passes them to the implementation of the canonical constructor, \( R_2 \), \( \text{mkComplex} \).

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 $2\pi$ if it is too large) or it could be mapped into the valid range modulo $2\pi$. 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 Summary

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.

References
