Abstract—We propose a way to use CafeOBJ algebraic specification language for describing formal specifications of sensor systems and for verifying desired properties to be satisfied by the systems. We give a way to not only verify specifications but also generate test cases from the results of verifications. Although we take a simple mobile device application using an acceleration sensor as an example, our approach may be applied to practical sensor systems.

Index Terms—Algebraic specifications, formal methods, sensor systems, test case generation.

I. INTRODUCTION

Sensor systems are used in variety areas including medical and healthcare systems. For example, we are developing a sensor system for detecting “Pointing and Calling” by nurses in hospitals, with a small wireless acceleration and gyro sensor and a wireless microphone attached to the arm and the head of a nurse respectively [1]. We are also developing other medical and healthcare systems: a visualization and analysis system for chair-rise action of hemiplegia patients, an evaluation system for car driving of encephalopathy patients. Because of a wide use of sensor systems including life-critical or safety critical systems in recent years, it is needed to establish a way to develop reliable sensor systems with formal methods. In this article, we study a way to develop reliable sensor systems with a formal specification language called CafeOBJ.

CafeOBJ is a most advanced formal specification language with many advanced features for describing and verifying equational specifications, for example, flexible mix-fix syntax, powerful and clear typing system with ordered sorts, parametric modules and views for instantiating the parameters, module expressions, and equational logic by rewriting used as a powerful interactive theorem proving system for verification with proof scores [2]. Test case generation from proof scores of CafeOBJ specifications has been studied in [3]. Our benefits of this study are to formalize sensor system especially for our case studies of medical and healthcare systems in algebraic specifications and to apply the test case generation to the sensor systems.

II. AN EXAMPLE OF SENSOR SYSTEM

We take an example of small applications using an acceleration sensor, which has one square box and a ball image, and change the position of the ball according to a tilt, which can be detected by acceleration sensor, of the device, i.e. when the device is tilted, the ball moves to that direction, which looks like a rolling ball on a board (Fig. 2). We assume a coordinate \((x, y)\) means the position of the ball where \((0, 0)\) stands for the leftmost-topmost of the screen, \((200, 0)\), \((0, 300)\) and \((200, 300)\) stand for the rightmost-topmost, the leftmost-bottommost and the rightmost-bottommost of the screen respectively.

A typical implementation is to update the position of the ball for some interval time, e.g. 20ms. Then, it takes the values of the acceleration sensor every 20ms, and updates the position of the ball. The ball is not allowed to go outside of the screen. We use Objective-C and Xcode IDE for developing iOS applications.

III. OTS/CafeOBJ SPECIFICATION FOR SENSOR SYSTEMS

We give a CafeOBJ specification for the above example in the model of the observational transition system (OTS),
which has several case studies of formal description and verification of practical systems, for example, secure protocols [4], e-government system [5] and so on.

CafeOBJ specifications consist of modules. Our OTS/CafeOBJ specification consists of a single module named SS as follows:

mod* SS{
  Import
  Signature
  Axiom
}

where Import, Signature and Axiom are the import part, the signature part and the axiom part of the module respectively.

In the import part, other modules, predefined or built-in, can be declared as imported modules of the module. In SS, we import a built-in module INT with the protect mode, denoted by pr (INT). INT specifies integers with ordinary operator. Since SS, includes pr (INT), we can use integers: ..., -2, -1, 0, 1, 2, ..., and integer expressions, for example, 1 + 2, 3 > 2, and so on, in the signature and the axiom parts in SS.

In the signature part, sorts and operators are declared. A sort is a name of entities of the same type. In OTS models, a system's state is specified via observations and transitions without any explicit structures, like, sets, lists, and so on. In OTS/CafeOBJ specification, a sort of system's states is declared between "*" and "*", and called a hidden sort. In SS, we declare a hidden sort Sys denoting states of the sensor system. An operator is defined on sorts. An operator has its arity and sort, where an arity is a string of sorts. In CafeOBJ, an operator f is declared like op f : arity -> sort. ops is used for declaration of plural operators in one line. bop and bops are used for declaration of behavioral operators in OTS models. A behavioral operator is a special operator whose arity includes the hidden sort. When the sort of a behavioral operator is not hidden, it is called an observation. When it is hidden, it is called a transition. In OTS models, a state is identified by the observed values with the observations and a transition is defined by the change of the observed values from the pre-state to the post-state of the transition. The following is the signature part of SS:

Signature :=
  * [Sys]*
  bops (cx) (cy) : Sys -> Int
  bop (tick__) : Int Int Sys -> Sys
  ops (ux__) (uy__) (lx__) (ly__) : Int Sys -> Bool
  op init : -> Sys

where cx and cy are observations for the coordinate (x, y) of the ball. The term (or expression) "cx S" means the value x of the coordinate (x, y) at the state S. tick is a transition, and the term "tick X Y S" means the state after an interval time with the sensor's acceleration values X and Y of S. The operators ux, uy, lx and ly are auxiliary operators defining the meaning of tick, which give conditions for checking borders of the screen. The operator init is the initial state of the system.

In the axiom part, equations to be satisfied are described. An equation is declared with eq. An equation with a condition is declared with ceq. The following equations are definition of the initial state:

eq cx init = 100 .

eq cy init = 150 .

where the terms "cx init" and "cy init" are the values observed by observations cx and cy at the initial state init. The above equations mean that x- and y-coordinates of the initial state are 100 and 150 respectively. The transition tick is defined via observations. The following is one of the equations for tick:

ceq cx (tick X Y S) = (cx S) + X
  if ux X S and lx X S

where X, Y and S are variables, and the equation means that the x-coordinate increases by X when the sensor's acceleration value for x-coordinate is X if the range conditions for the x-coordinate hold. Note that the term "tick X Y S" stands for the result state of applying tick with the acceleration values X and Y. The conditions ux and lx are defined by the following equations:

eq ux X S = (cx S) + X <= 200 .

eq lx X S = (cx S) + X >= 0 .

where the first equation means that the condition "ux X S" is the statement that the value of the ball's x-coordinate is less than or equal to 200 after adding X or not. The second one is it is greater than or equal to zero. Thus, the above conditional equation holds if the value of the x-coordinate is between zero and 200 after adding X or not.

In order for the ball not to go outside of the screen, it is needed to give the meaning of tick for the case that ux (or lx) does not hold as follows:

ceq cx (tick X Y S) = 200 if not ux X S .

ceq cx (tick X Y S) = 0 if not lx X S .

where the first (resp. the second) equation means that if the x-coordinate of the ball goes beyond 200 (resp. below zero), then it is forced to be set just 200 (resp. zero).

The observed values for the y-coordinate can be given similarly as follows:

ceq cy (tick X Y S) = (cy S) + Y
  if uy Y S and ly Y S .

ceq cy (tick X Y S) = 300 if not uy Y S .

ceq cy (tick X Y S) = 0 if not ly Y S .

eq uy Y S = (cy S) + Y <= 300 .

eq ly Y S = (cy S) + Y >= 0 .

IV. SPECIFICATION EXECUTION

CafeOBJ supports specification execution based on the term rewriting theory. A given term is reduced by applying equations in the axiom part. In reduction, each equation is regarded as a left-to-right rewrite rule. For a given term, an instance of the left-hand side of an equation is replaced with the corresponding right-hand side when the corresponding condition is reduced into true; the replacement is called a rewriting. Reduction is done by rewriting repeatedly until it cannot. The following is an example of reduction by
CafeOBJ languages (some parts are omitted):

CafeOBJ> red in SS : cx (tick -10 20 init) .
(90):NZNat

where the input term "cx (tick -10 20 init)" means
the x-coordinate of the ball after applying tick with -10 and
20 for x- and y-coordinates respectively to the initial state. It
should be (90, 170) from the initial coordinate (100, 150),
and the above output is 90 as expected. The input and output
of reduction is equivalent in all model of the specification.
Thus, the above reduction can be regarded as a proof of the
equation cx (tick -10 20 init) = 90.

V. VERIFICATION

By equational reasoning as above, more complicated
proofs can be described and verified semi-automatically.
In this example, we try to verify the following invariant
property: for an arbitrary state reachable from the initial state
by applying tick, the coordinate of the ball is inside of the
screen. The property can be given formally as follows:

\[
\begin{align*}
\text{op } & \text{inv } : \text{Sys } \to \text{Bool} \\
\text{var } & \text{S } : \text{Sys} \\
\text{eq } & \text{inv(S) } = \text{cx S } \leq 200 \text{ and} \\
& (\text{cx S }) \geq 0 \text{ and} \\
& (\text{cy S }) \leq 300 \text{ and} \\
& (\text{cy S }) \geq 0.
\end{align*}
\]

where our goal is to prove \text{inv(s)} for all reachable state \text{s}.

The notion of proof scores is useful to prove invariant
properties [2]. A proof score consists of several proof
passages. A proof passage consists of reduction with (if
needed) premises given by equations with constants. A proof
of invariant property is given by the induction on the
structure of reachable states. The base step is a proof for the
initial state:

red \text{inv(init)} .

The initial coordinate is (100, 150) and it satisfies the
predicate \text{inv}. CafeOBJ returns true for this reduction. The
induction hypothesis is given as follows:

\[
\begin{align*}
\text{op } & \text{s } : \to \text{Sys} \\
\text{eq } & \text{(cx s)} \leq 200 \text{ = true} . \\
\text{eq } & \text{(cx s)} \geq 0 \text{ = true} . \\
\text{eq } & \text{(cy s)} \leq 300 \text{ = true} . \\
\text{eq } & \text{(cy s)} \geq 0 \text{ = true} .
\end{align*}
\]

where the operator (constant) \text{s} stands for an arbitrary state
satisfying the invariant predicate. Note that the conjunction
of the above four equations is equivalent to \text{inv(s)}.

\[
\begin{align*}
\text{op } & \text{s' } : \to \text{Sys} \\
\text{eq } & \text{s' = tick x y s} . \\
\text{red } & \text{inv(s')} .
\end{align*}
\]

If this reduction of \text{inv(s')} returns true, which means
that \text{inv(s)} implies \text{inv(s')} is proved, then the proof by
the induction was completed since every reachable state from
\text{init} in \text{SS} can be represented as "\text{tick} x_0 y_0 (\text{tick} x_{n-1},
y_{n-1} ... (\text{tick} x_0 y_0 init) ... ) " for some \text{x_0, y_0, x_n, y_n} ... in \text{Int}.

Unfortunately, the above reduction does not return true
since CafeOBJ supports only simple equational reasoning.
Thus, we need to give appropriate case splitting for this
induction proof.

For example, when the following equations are assumed,
the reduction returns true:

\[
\begin{align*}
\text{eq } & \text{lx x s } = \text{false} . \\
\text{eq } & \text{ly y s } = \text{false} .
\end{align*}
\]

Let \text{LX} and \text{~LX} be the formulas "\text{lx x s } = \text{true}" and
"\text{lx x s } = \text{false}". Let \text{LY}, \text{~LY}, \text{UX}, \text{~UX}, \text{UY} and \text{~UY}
be the formulas defined in the same way. The above
equations correspond to \text{~LX} and \text{~LY}. The meaning of the
reduction returning true under the assumption is a proof of
the formula: \text{~LX}, \text{~LY}, \text{inv(s)} implies \text{inv(s')}.
To complete the induction proof, we need to prove all the other
cases, e.g. \text{~UX}, \text{~UY}, \text{inv(s)} implies \text{inv(s')}.

We have the completed proof score that consists of ten proof passages
as follow:

1) \text{inv(init)}
2) \text{LK}, \text{UX}, \text{LY}, \text{UY}, \text{inv(s)} implies \text{inv(s')}
3) \text{~LK}, \text{LY}, \text{UY}, \text{inv(s)} implies \text{inv(s')}
4) \text{~UX}, \text{LY}, \text{UY}, \text{inv(s)} implies \text{inv(s')}
5) \text{LK}, \text{UX}, \text{~LY}, \text{inv(s)} implies \text{inv(s')}
6) \text{~LK}, \text{~LY}, \text{inv(s)} implies \text{inv(s')}
7) \text{~UX}, \text{~LY}, \text{inv(s)} implies \text{inv(s')}
8) \text{LK}, \text{UX}, \text{~UY}, \text{inv(s)} implies \text{inv(s')}
9) \text{~LK}, \text{~UY}, \text{inv(s)} implies \text{inv(s')}
10) \text{~UX}, \text{~UY}, \text{inv(s)} implies \text{inv(s')}

where all proof passages return true. Note that they cover all
cases, i.e. the conjunction of the proof passages from 2 to 10
is equivalent to the formula "\text{inv(s)} implies \text{inv(s')}".

VI. IMPLEMENTATION

In [3], a way to generate a skeleton code from an
OTS/CafeOBJ specification has been given. We extend it for
sensor systems. From the observations, the interface named
\text{state} is defined as follows:

```objective-c
@interface state : NSObject
-(float)cx;
-(float)cy;
@end
```

where \text{state} has a method corresponding to each observation.
Then, the main interface is defined as follows:

```objective-c
@interface ball : UIViewController
<UIAccelerometerDelegate>{
    state *s;}
    -(void)tick:(float)x :(float)y;
@end
```

141
where the class, named ball, has the instance variable s of state, and the methods cx and cy for the observations and tick for the transition of the input specification. The methods for the observations just return the corresponding value of state, e.g., (float)cx{ return [s cx];}. The method tick is called in a method called every 20ms as follows:

```c
- (void)accelerometer : ... {
    [self tick:acceleration.x
        :acceleration.y];
    // the remaining part omitted
}
```

where accelerometer is a method called every 20ms in the sensor system (we omit the details).

An implementer is expected to fill in the definition part of the methods for transitions (tick in this example) together with other implementation if needed, e.g., instance variables, their getters and setters for state and so on. For example, tick may be implemented as follows:

```c
- (void)tick:(float)x :(float)y{ // implementation for x
    if([s cx] + x < 0){
        [s setBx:0];
    } else if([s cx] + x > 200){
        [s setBx:200];
    } else {
        [s setBx:[s cx] + x];
    }
    // implementation for y (omitted)
}
```

Note that the above implementation of tick is not generated from the specification, and is expected to be tested such that it satisfies the invariant property.

VII. TESTING

Formal verification for an invariant property guarantees that it has been mathematically proved that the specification satisfies the invariant property. However, there is a gap between a specification and its implementation, for example, the former assumes INT to be the (infinite) set of all integers, though the latter may use 32-bit integers. Thus, even if the specification is verified formally, we still need to check whether its implementation satisfies the properties. For this purpose, we give a way to generate test cases for the property. The literature [3] gives not only a way to generate a skeleton code but also test cases generated from the specification and their proof scores.

To obtain a test case, we give a translation from operators except behavioral ones used in the input specification to methods in the implementation. For SS, the addition operator +, the comparison operators >= and <= of INT, the auxiliary operators ux, uy, lx and ly are translated into the following methods respectively:

```c
+ (float)plus:(float)x :(float)y{
    return x + y;
}
+ (boolean_t)ge:(float)x :(float)y{
    return x >= y;
}
```

Then, we have a way to translate the equations in the specification, and the invariant property and an assumption in each proof passage into the test codes.

A. Axiom Tests

Testing for axioms is obtained from the specification’s equations defining the meaning of transitions. In SS, the transition tick is defined by six equations. From each of these equations, we have a skeleton code of test cases that test the equation under the condition. Consider the following equation:

```c
cq cx (tick X Y S) = (cx S) + X
```

The following skeleton code of test cases (we use SenTestingKit package built in Xcode IDE) is generated from the above equation:

```c
// Axiom_test_1
x = [test setCx:pre_cx = ];
[y = [test tick:x :y];
pre STAssertTrue([test cx] == [test plus:pre_cx :x],
[StringWithFormat:@"pre:err"]);
post STAssertTrue([test cx] == [test plus:pre_cx :x],
[StringWithFormat:@"post:err"]);
```

The test code consists of four parts. The first part is the preparation part, where a tester fills an appropriate value for each variable in the boxes. The second one is an assertion for the pre-condition, where the variables are tested such that they satisfy the condition generated from the condition of the equation. The third one is the test execution of the transition. The last one is an assertion for the post-condition, where the current value of the x-coordinate is tested such that it is equivalent with the sum of the x-coordinate of the pre state and the value of the variable x, which comes from the right-hand side of the equation.

B. Invariant Tests

As we showed in the previous section, a proof score for an invariant property consists of several proof passages. Each proof passage for the induction step corresponds to an assumption (case splitting). The assumptions are considered to be important for verifying the invariant property, and they are expected to be useful for tests of the invariant property in an implementation. The following is the output from the proof passage 2 for the case of LX, UX, LY, UY in the previous section:

```c
// Inv_test_1
x = [test setCx:pre_cx = ];
[y = [test setCx:pre_cx = ];
```
In the pre-condition part, two assertions exist. The one assertion checks the assumption of the proof passage; LX, UX, LY, UY, and the other one checks the invariant property for the variables filled by a tester. After executing the transition tick, the invariant property is checked again.

VIII. CONCLUSION

We applied formal methods with CafeOBJ language to development of a simple acceleration sensor system. We gave a way to model the sensor system in OTS models, and to describe a CafeOBJ specification. Some invariant property was verified by using the notion of proof scores. Lastly, we gave a way to generate test cases that may be appropriate for testing the invariant property in the implementation.

In the previous section, we only showed a way to generate test cases for logic unit tests with a tester explicitly setting variables. To give a way to obtain test cases for application tests by using real sensor values is one of our future work.

The example we provided in this article is so simple, though we can extend our approaches for more practical system. To apply our proposed methods to practical sensor system is another one of our future work. One of the candidates is the medical and healthcare systems we have developed, e.g. in [1].

REFERENCES


