jAADD User's Guide
Version 2.0

Chair of Cyber-Physical Systems
Christoph Grimm, Carna Zivkovic, Sevilay Akkus

The jAADD library was and is developed with support from a number of research projects. We in particular want thank the following funding agencies and companies that provide and provided financial support:

- ANCONA; BMBF, Förderkennzeichen 16ES0210-15.
- Robert Bosch AG, Germany.
- Mentor Graphics GmbH, Germany.
- Intel AG, Germany.
- AVL List, Graz, Austria.
- GENIAL!; BMBF F&E 16ES0865-16ES0876.
- Arrowhead Tools; ECSEL Joint Undertaking (JU) under grant agreement No 82645.
- AnastASICa, BMBF F&E.
- AVL List, Graz, Austria.

Contents

Contents **3**
1 Preliminaries: An introduction to AADD **4**
1.1 Intervals 4
1.2 Computing with intervals 4
1.3 Affine arithmetic 5
1.4 BDD^A and AADD 5
1.5 Modeling dynamic systems with AADD and AADD streams 8
2 Arithmetic and Boolean Expressions **9**
2.1 AADD and Arithmetic expressions 9
2.2 BDD^A and Boolean expressions 12
2.3 Example: PI control loop 13
3 Conditions, Conditional Statements and Iterations **14**
3.1 From decision variables to conditions 14
3.2 Conditional statements 15
3.3 Example: Water level monitor 16
4 Input/Output **17**
4.1 Conversion to a string 17
4.2 Conversion to JSON 17
4.3 AADD and BDD^A traces 18
5 Expression Parser and Command Line Interpreter **19**
5.1 The AADD expression language: syntax 19
5.2 Use of AADD expression language in Java 20
5.3 User-defined functions 21
Annex A: API of AADD, BDD, Conditions **22**
A.1 DD public methods/fields 22
A.2 AADD public methods/fields 23
A.3 BDD public methods/fields 24
Annex B: Revision history **25**
Annex C: Further Literature on AADD **26**

1 Preliminaries: An introduction to AADD

This section gives a brief introduction into the theory of AADD. We furthermore point out some related features of jAADD.

1.1 Intervals

Intervals are used for various reasons:

- *A precise value is uncertain.* However, often upper and/or lower bounds of some quantities can be given.
- *Alternative solutions are allowed.* Often, we leave the precise value for a quantity in a specification open and just give lower and/or upper bounds.
- *A value cannot be represented.* Quantities such as π and other Real numbers cannot be represented precisely by e.g. floating-point numbers, no matter how many bits are used. Nevertheless, we can give upper and lower bounds: $3 < \pi < 4$ is mathematically precise, while $\pi = 3.14$ is wrong.

A (real) interval is a set of real numbers that is specified by an upper and lower bound:

$$[min, max] = \{min, max, x \in \mathbb{R} \mid \ min \leq x \leq max\}$$

We distinguish the following kind of intervals:

- *Range* – The interval bounds are finite, and $min < max$
- *Scalar* – The interval has exactly one element: $[x, x] = \{x\}$
- *Empty* – The interval is the empty set {}.
- *Real* – The interval is equal to the set $\mathbb{R}$; there are no (real-valued) upper and lower bounds.

Note, that mathematics also knows open and closed intervals (min, max) where the bounds are not elements of the interval; jAADD does not support open intervals.

> jAADD supports closed intervals of the kind *finite, scalar, empty* and *unbounded*.

1.2 Computing with intervals

The nice thing with intervals is that we can do the arithmetic operations on the intervals as if they were real numbers, and get an interval as result. However, note that not all laws for the reals do hold for intervals. In particular, the following holds: Let $f: \mathbb{R}^m \to \mathbb{R}$ be a function on the reals, and $F: \mathbb{I}^m \to \mathbb{I}$ is its interval-extension. Then, for $V \in \mathbb{I}^m$:

$$v \in V \Rightarrow f(v) \in F(V)$$

For example, if we add two intervals $[1,2]$, $[3,4]$ then the sum of two intervals $[4,6] \ = \ [1 + 3, 2 + 4]$ contains all sums of its real-valued elements.

Furthermore, for two intervals U, V, it holds:

$$U \subseteq V \Rightarrow F(U) \subseteq F(V)$$

For example, consider $[2,3]$ and $[1,4]$. Then $sqr([2,3])$ shall be subset of $sqr([1,4])$.

Note, that this not trivially holds for all functions and intervals; e.g. in the case of local extrema.

> jAADD extends a large selection of arithmetic operations on the reals towards intervals.

1.3 Affine arithmetic

Range arithmetic allow some *over-approximation*. For example, we know that $[4,6] = [1,2] + [3,4]$. In particular for complex nonlinear functions, over-approximation can often not be avoided. A simple example where over-approximation can occur is the subtraction of two intervals:

$$[0,2] - [0,2] = [-2,2]$$

Intuitively, if we subtract a value from itself, we expect the result 0. Unfortunately, we do not know if the intervals $[0,2]$ mean the same value, or two independent values from $[0,2]$; hence, the result – without knowing dependency - must be $[-2,2]$.

In longer computational chains, such over-approximations accumulate, and eventually lead to an unbounded interval (wrapping effect). To avoid the wrapping effect, jAADD uses an extended *affine arithmetic*[1]. This avoids the wrapping effect by tracking linear dependencies in affine forms. An affine form $\tilde{x}$ is a linear combination of independent noise terms:

$$\tilde{x} = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$$

where the coefficients x_i are floating point numbers, and the noise variables ε_i are unknown reals from the interval $[-1,1]$. Now, the interval $[0,2]$ can be represented as $1 + \varepsilon_1$; then $(1 + \varepsilon_1) - (1 + \varepsilon_1) = 0$. However, $[0,2]$ can also be represented as $1 + \varepsilon_2$; then we get $(1 + \varepsilon_1) - (1 + \varepsilon_2) = \varepsilon_1 - \varepsilon_2$ which describes an interval $[-2,2]$.

Also note, that if there is some error due to rounding of a coefficient or due to linear inclusion of a nonlinear function that is bounded by x, we can add a new term $x \cdot \varepsilon_k$[2].

> Affine arithmetic represents dependencies among different intervals by the index of noise variables. Noise variables with the same index are correlated; noise variables with different index are independent.

> jAADD uses a combination of interval arithmetic and affine arithmetic. It provides often much tighter boundaries as affine arithmetic or interval arithmetic alone.

1.4 BDD^A and AADD

Real numbers allow us to model many problems. However, some problems are more suitable to be handled by discrete models, e.g. by Boolean functions. Efficient representation of Boolean functions are e.g. Binary Decision Diagrams (BDD[3]). We in particular work with Reduced and Ordered BDD that are often also abbreviated ROBDD; we stay however with the shorter abbreviation BDD.

(RO)BDDs are directed acyclic graph with two leaf nodes: True and False. The internal nodes refer to decision variables. Depending on the (unknown) value of the decision variable, a path through the tree is chosen that ends in True or False, depending on the Boolean function: Boolean decision variables and conditions.

---

[1] Stolfi J.; Figuiredo L.H.: An Introduction to Affine Arithmetic. Trends in Applied and Computational Mathematics, Vol. 4 No. 3 (2003), https://tema.sbmac.org.br/tema/article/view/352.

[2] jAADD uses an interval that sums up all these uncorrelated terms; it can be asymmetric.

[3] Bryant, R.E. (2018) Binary Decision Diagrams. In: Clarke E., Henzinger T., Veith H., Bloem R. (eds) Handbook of Model Checking. Springer, Cham. https://doi.org/10.1007/978-3-319-10575-8_7.

Boolean decision variables and the ITE function

BDDs can be represented visually by a graph or by repeated calls of the ITE (if-then-else) function. The ITE function takes three parameters of type Boolean. If the first parameter – the decision variable - is True, then the result is the 2nd parameter, else the result is the 3rd parameter. However, keep in mind that if the value of a decision variable is known, no BDD is needed: the result can be evaluated to either True or False.

Exercise: *Which is the result of ITE(true, false, true)?*

However, if the decision variable is unknown, we cannot give a Boolean value as result. Then, we have to represent the result as a function of the decision variable.

Exercise: *Which is the result of ITE(a, true, false) where a is unknown? Draw the BDD!*

Boolean decision variables are just unknown variables. A BDD that models a Boolean function $f(a,b,c) = (a\ and\ b)\ or\ c$ by the ITE function could be written as:

$$f(a,b,c) = ITE(c, True, ITE(b, ITE(a, True, False), False))$$

Exercise: *For illustration, draw the BDD.*

jAADD provides a very special and basic implementation of BDD.
If you only need some BDD implementation without interaction to or support for arithmetic operations don't use jAADD.

Note, that in jAADD the ITE function is a method of a BDD (=1st paramteter) that takes two parametes (=2nd, 3rd parameter of ITE function above).

Conditions: linear constraints as decision variables

Linear constraints are conditions of the type $x > 0$, where x is an affine form. Boolean and nonlinear arithmetic expressions are brought to this form: every arithmetic expression, in affine arithmetic, results in an affine form, and all kind of comparisons can be brought to the form $x > 0$. (Note, that we do not consider equalities here – with Floating point number this is futile).

BDD^A

BDD^A are BDDs where the decision variables can be linear constraints. For example, let there be three Boolean variables a, b, c and b, c be the result of some arithmetic computation on $x, y, z \in \mathbb{R}$; e.g. $b = f(x,y,z) > 0, c = g(x,y,z) > 0$. Then we get for our example:

$$f(a,b,c) = ITE(a, True, ITE(g(x,y), ITE(f(x,y), True, False), False))$$

The use of linear constraints as decision variables in BDD^A introduces dependencies between the discrete and the continuous domains: for conventional, discrete BDD, all decision variables are just unknown Booleans. For BDD^A, they can be linear constraints of type $x > 0$. This is a linear inequation on the noise symbols as unknowns:

$$x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n > 0$$

We use the following terms to describe the dependencies between discrete and continuous domains.

- **Condition set** $X_P(v)$ of a path to a leaf v: The set of all conditions on internal nodes on a path from the root of a BDDA to the leaf v, maybe negated, such that v is selected.
- **Path condition** $\chi_P(v)$ of a path to a leaf v: The conjunction of all conditions in $X_P(v)$.

Note, that the path condition is not necessarily feasible. As a simple example, consider:

$$f(a) = ITE(a > 1, ITE(a > 2, True, False), True)$$

The condition $a > 1$ is the "highest" node in the BDD^A. The node with the condition $a > 2$ is hence a subtree below the condition $a > 1$. Hence, $a > 2$ will always be True, the leaf with value False will hence never be reached. More general: if there exists an assignment of values to the noise variables such that the path condition can be true, a node is **feasible**. Otherwise the node is **not feasible**. Note, that the above BDD^A for all feasible cases has the result True. It can hence be reduced to a single leaf with the value True.

AADD

AADD are BDD^A where the leaves are affine forms instead of True or False.

Now, each of the conditions in the condition set of a node introduces a linear constraint on the noise variables. Hence, the interval bounds of the affine forms at the leaves are affected: The minimum value of a leaf v with an affine form $\tilde{a} = a_0 + a_1\varepsilon_1 + \cdots + a_n\varepsilon_n$ with path set X_P is now obtained by a linear program:

$$Min(a_0 + a_1\varepsilon_1 + \cdots + a_n\varepsilon_n)\ s.t. X_P\ \wedge\ -1 \leq \varepsilon_i \leq 1 \forall i \in \{1, \dots, n\}$$

Likewise, for the maximum:

$$Max(a_0 + a_1\varepsilon_1 + \cdots + a_n\varepsilon_n)\ s.t. X_P \wedge\ -1 \leq \varepsilon_i \leq 1 \forall i \in \{1, \dots, n\}$$

Note, that there exists not necessarily a solution; then, the leaf v is infeasible.

Operations on BDD^A and AADD

Operations on BDD^A resp. AADDs x, y are defined such that for an operation $z = x \odot y$ it holds:

For each leaf of x and y there exists one node in z with:

- $Value(z) = Value(x) \odot Value(y)$
- $X_P(z) = X_P(x) \cup X_P(y)$

Or, short and informally, the result is obtained by computing the operation for each leaf, ensuring that a leaf for each path condition of the operands is in the result.

Relational operations

If we compare an affine form resp. an interval with a real value, we can get the following results:

- True or False, if the value does not lie in the interval. For example, 5 > [1, 2] is for sure true, as well as [3,4] < 10.
- A new decision variable that is a constraint of kind $\tilde{a} > 0$ if the value lies in the interval. For example, consider the condition [1,3] > 2. Here, the result can be true and false, depending on the noise variables.

In the latter case, the result of a relational operation is a BDD^A that models the two possible Boolean outcomes and its dependency from the comparison.

Likewise, if we compare two AADD, we will get a BDD as for other operations on BDD^A and AADD.

1.5 Modeling dynamic systems with AADD and AADD streams

AADD can be used for various use cases. A particular use case is the symbolic simulation of dynamical systems in different models of computation.

For this use case, we represent signals by bounded sequences of time-tagged values s_t, where the subscript t is from an at least partially ordered set T that models time, and where the values s_t are either real or Boolean values:

$$s_T = \langle s_1, s_2, \dots \rangle$$

This is a common approach that permits to model systems in different models of computation. If just the (local) order in the sequence is relevant, a system is *untimed*. If T is a discrete ordered set and introduces a global synchronization between two or more signals, the system is a *discrete-time system*. If T is a contiguous subset of the reals, the system is a *continuous-time system*.

To model signals of different kind with AADD, we use *streams* of AADD:

$$\hat{s}_T = \langle \hat{s}_1, \hat{s}_2, \dots \rangle$$

Now, $\hat{s}_T$ does not represent a single value, but a set of signals (signal-set). The signal-set represents all possible trajectories; by assigning the noise symbols some values, all possible signal trajectories can be obtained (maybe additional by over-approximation). However, $\hat{s}_T$ represents not only the enclosing hull of all reachable signal trajectories.

2 Arithmetic and Boolean Expressions

We now explain the use of AADD and BDD to compute arithmetic and Boolean expressions. jAADD 2.0 is provided as a Java archive that can be used from languages that work with the Java Virtual Machine (JVM). These languages include Java, Kotlin, Scala, Groovy and many others.

A shared library version for use with C/C++ is planned in a future release.

For using jAADD on the JVM platforms, the jAADD jar file must be in the classpath of a Java program, and the library and the used classes, methods, fields must be imported.

The following examples use the Java API. For Kotlin, overloaded operators are available that significantly improve readability.

2.1 AADD and Arithmetic expressions

Instantiation of AADD

For computing with arithmetic expressions use the class AADD. Intervals and affine forms are considered as a special case of an AADD that is just a single leaf. To use AADD, one must import it from the package jAADD:

```
import jAADD.AADD;
```

Now you can use methods of the package jAADD's class to instantiate objects of the class AADD. For this purpose, don't use the constructors. They are protected anyway.

There is a *"Factory"*, this are static methods in the AADD class, that provide you with AADD elements.

To get an AADD of kind scalar, that means an affine form

$$\tilde{a} = a_0$$

the method `AADD.scalar(a0)` must be called. The use of one of the constructors shall be avoided. `Scalar` is a static member of the class AADD. It can hence be called without a concrete instance, just by using the class name AADD as a prefix, e.g.:

```
AADD scalar=AADD.scalar(a0);
```

To get an AADD of the kind Range that is represented by an affine form

$$\tilde{a} = a_0 + a_i\varepsilon_i$$

which spans an interval $[a_0 - a_i, a_0 + a_i]$ that is dependent on the noise variable with index i, one has to use the method `AADD.Range(min, max, "ai")`: where "ai" can be an arbitrary string that uniquely identifies the noise symbol:

```
AADD range=AADD.range(min, max, name); // min, max: double, name: String
```

To describe dependency, two AADD must share a noise variable. This can be achieved by giving them the sam noise variable's name. AADD that model an interval of kind unbounded in or empty are represented by the following constants:

```
AADD a = AADD.Empty; // Empty set; no real number in it.
AADD b = AADD.Reals; // All real numbers in it.
```

Note, that the Empty interval is a single, static final instance. All references of it share this single instance that cannot be changed. Cloning Empty will return a reference to the same object again.

Exercise: *Instantiate AADD of kind scalar with value 1.0, of kind range using a noise variable with index 1, the whole set of the Reals, and an empty interval.*

```
import jAADD.AADD;
void instantiation() {
    AADD scalar = AADD.scalar(1.0);
    AADD range  = AADD.range(2.0, 3.0, "range's noise symbol");
    AADD reals  = AADD.Reals;
    AADD empty  = AADD.Empty;
    System.out.println("scalar = " + scalar);
    System.out.println("range  = " + range);
    System.out.println("reals  = " + reals);
    System.out.println("empty  = " + empty);
}
```

The exercise prints the following output:

```
  scalar = 1.0
  range  = [2,00; 3,00]
  reals  = (−∞;+∞)
  empty  = ∅
```

Arithmetic computations with AADD

Objects of the class AADD have methods that permit to do arithmetic operations. For example, to compute the sum of two AADD a, b, i.e. $a = a + b$:

```
a = a.plus(b);
```

Or, for $a = a + b * c/d - e$:

```
a = a.plus(b.times(c).div(d)).minus(e);
```

As can be seen by the example above, the arithmetic expression has to be mapped to subsequent calls of the respective methods.

Note that Objects of the class AADD are *immutable*. This means, objects of the class AADD can only be created, but once created, the objects cannot be changed anymore (at least not by arithmetic operations): each operation on AADD will create as a result a new object of the class AADD, but its parameters are not changed.

However, also note that in Java the variables are *references* that can and will change. For example, if we declare an AADD c, we can assign it an object and later assign c another object. Then, the objects are not changed; they are immutable. But the variable c refers to different objects.

Exercise 1: *Given two intervals $a = [1,2], b = [1,2]$ that are independent which means they use affine forms with different noise symbols. Compute $a - a$ and $a - b$ and print and compare the results.*

```
AADD a = AADD.range(1.0, 2.0, "a's noise variable");
AADD b = AADD.range(1.0, 2.0, "b's noise variable");
System.out.println("    a-a = " + a.minus(a));
System.out.println("but a-b = " + a.minus(b));
```

The exercise prints the following output:

```
    a-a = [-0,00; 0,00]
but a-b = [-1,00; 1,00]
```

Exercise 2: *What is the max and min volume of an ellipsoid with width/height/depth independently from the range [1, 10]?*

Solution:

$$vol = \frac{4}{3}\,\pi\, a\, b\, c$$

```
void ellipsoid_excercise() {
    // Volume of ellipsoid = 4/3 pi a b c where a,b,c=[1,10]
    AADD a = AADD.range(1.0, 10.0, "a");
    AADD b = AADD.range(1.0, 10.0, "b");
    AADD c = AADD.range(1.0, 10.0, "c");
    AADD pi = AADD.range(3.141, 3.142, "pi");
    AADD vol = pi.mult(a.mult(b.mult(c)));
    vol = vol.mult(AADD.scalar(4.0/3.0));
    System.out.println("Volume = " + vol);
}
```

The result computed by jAADD is:

```
Volume = [4,19; 4189,33]
```

Operations on unbounded and empty intervals

Operations on empty intervals always return an empty interval.

Operations on unbounded intervals return in many cases an unbounded interval. However, there can exist operations that return other kind of AADD, e.g. multiplication with 0 which returns an AADD of kind scalar and value 0.

2.2 BDD^A and Boolean expressions

For symbolic computation on Boolean variables, we have to use its symbolic representation by BDD^A. BDDA are like (RO)BDD, but interact with AADD as we will see in the next section. To use it we import it from the jAADD library:

```
import jAADD.BDD;
```

Instantiation of BDD^A

The class BDD, provides the following constants and methods to create BDD:

```
BDD BDD.constant(Boolean val); // leaf with true resp. false
BDD BDD.variable(String uid);  // internal node, decision variable uid
```

Furthermore, BDD.True and BDD.False are static final objects (constants) for which only one single instance exists. In fact, BDD^A are Reduced Ordered BDD that, due to the reduction, have at most one True and one False leaf.

To create a decision variable with an internal node, the method BDD.variable(String uid) shall be used. It creates a BDD with an unknown decision variable that is True resp. False, depending on the variable named `varname`.

Furthermore, the method `BDD.constant(bool)` can be used to create new objects that model just a single Boolean value with one of the values *true* or *false*; it will be one of the leaves True or False.

Exercise: *Try instantiation of the constants using the method and the constants described. Print the objects.*

Solution:

```
void BDDinstantiation() {
    BDD a = BDD.constant(true); // gets BDD w/ Boolean true or false
    BDD x = BDD.variable("x");  // Unknown decision variable "x"
    System.out.println("a="+a);
    System.out.println("x="+x);
}
```

Boolean expressions and functions

On BDD the usual Boolean operations are available as class methods. Hence, one can evaluate Boolean expressions. Like for AADD, the operations and, or, xor, not, etc. are defined as methods on BDD. Furthermore, on BDD the ITE function is defined. The ITE function takes two parameters: the first parameter is a decision variable. If it is true, then the 2nd parameter is returned, otherwise the 3rd parameter is returned. On BDD it is implemented as a method. The decision variable is then the BDD object itself, the 1st parameter of the method is the result for the case that the decision variable is true, and the 2nd parameter of the method is the result for the case that the decision variable is true.

To compute the expression on Boolean constants and the Boolean variable x from above: `d = false && x || true` and `e = true && x`:

```
BDD d = False.and(x).or(True);
BDD e = True.and(BDD.variable("x"));
System.out.println("d=" + d + "\ne=" + e );
```

The resulting output is:

```
d=true
e=ITE(“x”, true, false)
```

Exercise: *Explain the results.*

2.3 Example: PI control loop

In this section we give a more comprehensive example.

Exercise: *We can model a simple control loop in Java and with doubles as follows:*

```
void PIcontrolDouble() {
    var set   = 0.5;
    var is    = 1.0;
    var piout = 0.4;
    double inval;
    for(int i = 1; i<50; i++ ) {
        inval = setval – inval;
        piout += inval * 0.05;
        isval = isval*0.5 + piout*0.5;
    }
}
```

Re-write the PI controller such that we can see all possible is-values for a range from 0 to 2, and also set the PI controller output to a matching initial state.

```
var setval = AADD.range(0.4, 0.6, "setval");
var isval  = AADD.range(0.9, 1.0, "isval");
var piout  = AADD.range(0.5, 0.51, "piout");
AADD inval;
for (int i = 1; i<50; i++) {
    inval = setval.minus(isval);
    piout = piout.plus(inval.times(AADD.scalar(0.05)));
    isval =
      isval.times(AADD.scalar(.5)).plus(piout.times(AADD.scalar(.5)));
}
```

For visualization, see Section 4.

3 Conditions, Conditional Statements and Iterations

In Section 2 we have computed arithmetic and Boolean expressions. However, the results had no impact on the control flow. We will show in this section how to compute conditions, how to deal with conditional statements and with iterations (loop).

3.1 From decision variables to conditions

Decision variable with unknown value

In a BDD the decision variables are Boolean variables of unknown value. To create a BDD that is not a leaf, one must use the Factory method BDD.variable. It adds a decision variable and returns a BDD with an internal node and the leaves true and false. Note, that if the decision variable is known to be true or false, jAADD will return true or false directly, and not a BDD. An example has already been given in the section on BDD. However, why do we add this superscript "^{A}" to BDD? Let's try two examples.

Decision variable that is a condition

If we compare an affine form resp. an interval with a real value, we can get the following results:

- True or False, if the value does not lie in the interval.
- A new decision variable if the value lies in the interval.

The *comparison* of two AADD returns a BDD whose leaves are results of the respective comparisons of the comparison of the two AADD. For example, consider the following Java code:

```
void comparison() {
    AADD a = AADD.range(1.0, 3.0, 1);
    AADD b = AADD.range(2.0, 4.0, 2);
    BDD  c = a.gt(b);
    System.out.println("c="+c);
}
```

We compare the two AADD a, b that represent intervals [1,3] resp. [2,4]. We the result of the comparison is unknown; the program prints:

```
c=ITE(2, true, false)
```

The "2" in the ITE function refers to the condition. It is just the index of the comparison "a>b".

Now let's slightly modify this example as follows:

```
void comparison2() {
    AADD a = AADD.range(1.0, 3.0, "a");
    AADD b = AADD.range(2.0, 4.0, dep("a", 1.0));
    BDD  c = a.gt(b);
    System.out.println("c="+c);
}
```

Now, we get the following result:

```
c=false
```

This is because now a and b are affine forms that share a dependency; there is an overlap of the interval, but both affine forms grow similarly: b will always be $a + 1$, and $a > b$ is hence false.

Exercise: *For a, b as declared above, compute the condition $a * b > a + b$ and print the result.*

```
AADD a = AADD.range(1.0, 3.0, "a");
AADD b = AADD.range(2.0, 4.0, "b");
BDD  c = a.times(b).gt(a.plus(b));
System.out.println("c="+c);
```

The result is:

```
c=ITE(2, true, false)
```

3.2 Conditional statements

Imagine a computation as follows, where the result depends on a conditional statement:

```
double a = [-1.0, 1.0]; // let a be a double from this interval.
if (a <= 0) a = a+2;
else a = a-2;
```

How can we compute the value of a after the conditional statement? The problem is that the condition (a <= 0) cannot be evaluated to True or False. For this purpose, jAADD provides the functions

- `ifS`,
- `endS`,
- `elseS` and
- `assignS`.

For using AADD, this program then can be written as:

```
void iteExample1() {
    AADD a = AADD.range(-1.0, 1.0, "a");
    ifS(a.le(AADD.scalar(0.0), "a<=0")
        a = a.assignS(a.add(AADD.scalar(2.0)));
    elseS()
        a = a.assignS(a.sub(AADD.scalar(2.0)));
    endS();
    System.out.println("a="+a);
}
```

The result we get is:

```
a=ITE("a<=0", [-2,00; -1,00], [1,00; 2,00])
```

In general, to compute a conditional statement with AADD and BDD^A, apply the following steps:

- Replace the keywords if and else with ifS and elseS; add a terminating endS() at the end of the if and else block.
- Replace assignments `x = f(x, y, …)` with `x = x.assign(f(x, y, …))`

3.3 Example: Water level monitor

In the following we demonstrate the use of jAADD for (very simple) bounded-time model checking:

- First, do a symbolic simulation
- Second, check assertions on the symbolic representation of all possible signals.

A water level monitor can be modelled in discrete time with jAADD. Assume a water tank that can be either filled with inflow, or drained with outflow. An automaton sets the rates after a lower threshold 2.0 was been crossed, or drained after an upper threshold 10.0 was crossed.

We assume uncertain parameters and initial state:

- The initial level is in [1, 11],
- The outrate is in [-1, -0.6],
- The inrate is in [0.6, 1], and
- The initial rate (inrate or outrate) is unknown.

We can model this as follows in Kotlin and jAADD; in Java, in particular overloaded operators are not available:

```
var outrate = range(-1.0 .. -.6, "outrate")
var inrate  = range(0.6 .. 1.0, "inrate")
var level   = range(4.0 .. 5.0, "initial level")
var rate = inrate
```

Now, we can make a simple dynamic model that, in discrete time steps of 1 sec, checks if an upper threshold (10.0) or a lower threshold (2.0) is crossed. And, if the threshold is crossed, assign a new value to the rate:

```
var time = 0.0
for (time in 0 .. 30) {
  assert(wl in 0.99 .. 11.01)
  ifS(level Gt scalar(10.0));
      rate = rate.assignS(outrate)
  endS()
  ifS (level Lt scalar(2.0));
      rate = rate.assignS(inrate)
  endS()
  level += rate
}
```

The resulting level continuously stays in the range [1, 11] which already can be seen after the first step with a negligible runtime. One could already stop here, but above we compute 30 steps to analyse scalability. Note, that this example exhibits exponential runtime as it runs into the path explosion problem. Nevertheless, for 30 steps, runtime should be a few seconds, but towards a minute for 100 steps etc.

4 Input/Output

4.1 Conversion to a string

AADD provide infrastructure for conversion to a string. This is the Java method toString.

For example, to convert an AADD a to a String, just call this method:

```
AADD a;
String s = a.toString();
```

As the method “toString” is an overloaded standard-method of Java for conversion to strings, you can directly print strings as follows:

```
System.out.println(“a is: ” + a);
```

4.2 Conversion to JSON

JSON is a format for exchanging objects across platforms, e.g. between a Java Backend and a JavaScript frontend. JSON represents the data in an object as a String.

Conversion to JSON can be done by the method `toJson`.

```
void JsonExample() {
    AADD a = AADD.range(1.0, 2.0, “a”);
    String s = a.toJson(a);
    System.out.println(“s = ” + s);
}
```

The result is:

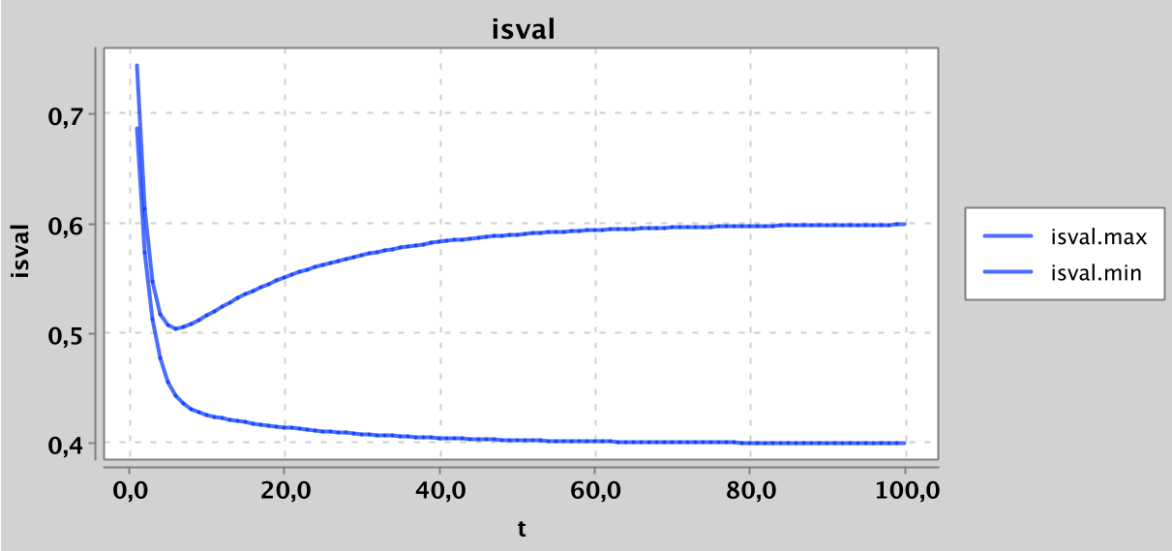
```
s = {
  "index": 2147483647,
  "leafValue": {
    "x0": 1.5,
    "xi": {
      "4": 0.5
    },
    "r": 0.0,
    "min": 1.0,
    "max": 2.0
  },
  "feasible": true
}
```

4.3 AADD and BDD^A traces

Let's come back to the PI controller example. It would be nice to not only compute the AADDs over time, but to also trace and show the signals over time. In this case, we have t={1, 2, …, 100}. This can be done via an AADDTrace object.

```
void piControl() {
  var setval = AADD.range(0.4, 0.6, "setval");
  var isval  = AADD.range(0.9, 1.0, "isval");
  var piout  = AADD.range(0.5, 0.51, "piout");
  var t = new AADDTrace("isval");
  AADD inval;
  for (int i = 1; i<50; i++) {
    inval = setval.minus(isval);
    piout = piout.plus(inval.times(AADD.scalar(0.05)));
    isval =
isval.times(AADD.scalar(0.5)).plus(piout.times(AADD.scalar(0.5)));
    t.add(isval, i);
  }
  t.display()
}
```

In a window, the following graph is displayed. It shows the minimum and the maximum values of isval:



The results might be complex and we might want to save it in a database for later analysis. To write the results in a file, just call this method of AADDstreams:

```
static public void toJson(String filename)
```

5 Expression Parser and Command Line Interpreter

The use of methods to describe arithmetic expressions is inconvenient. jAADD provides an expression parser that is easy to use in the package exprParser. In this section we show how to write code like:

```
System.out.println("a+b*c=" + p.eval("a+b*c"));
```

5.1 The AADD expression language: syntax

Statements

The AADD expression language supports four statements:

- The assignment of values to variable names.
- The definition of functions.
- The definition of equations.
- Evaluation of expressions including functions.

All lexical elements are case-insensitive; for example, “PI” and “pi” mean the same identifier. As identifier (“NAME”) we allow sequences of letters, separated by dots (“.”). As numbers (“FLOAT”) we allow floating point numbers in the Java-Syntax.

No declaration of names is necessary. Type checks are done during run-time, and not declared names will be resolved as one of all possible Real or Bool values.

The syntax is, in EBNF, as follows, where CExpr is a function call or Boolean or arithmetic expression:

```
Stmt    :-    "var" NAME ":=" CExpr
            |  "fun" NAME "(" CExpr ("," CExpr)* ")" ":=" CExpr
            |  "eqn" NAME "==" CExpr
            |  CExpr
```

An expression CExpr is described by the following rule:

```
CExpr.  :-    Expr  [ (">" | "<" | "=" | “<=” | “>=”) Expr ]
```

The above operators compare two AADD and return a BDD.

```
Sum     :-    Product (("+" | "-" | "|") Product)*

Product :-    Value (("*"|"/"|"&") Value)*
```

The arithmetic operators are +, -, *, /; Boolean operators are & (and) and | (or).

```
Value   :-    ["-"] (
                    FLOAT
                  | "(" CExpr ")"
                  | VARIABLE_NAME
                  | FUNCTION_NAME "(" CExpr ("," CExpr)* ")"
                )
```

Note, that

- a unary sign can precede any value,
- brackets can be used to enforce a particular order of computation.

Expressions in the above language can be integrated in Java code, or be applied interactively in a command line interpreter.

5.2 Use of AADD expression language in Java

The use of the AADD expression language in Java requires the following steps:

1. Instantiate an object of the class `ExprParser`.
2. Evaluate of some expressions, in particular:
    a. Define some variables with known values.
    b. Compute expressions or functions with the variables.
3. Get the result.

The step 2 can throw the exception `ParseError`. This is a hint for a syntax error. A diagnostic message is reported.

The step 2 can also throw the exception `ExprError`. This is a hint for a type incompatibility or other semantic errors. A diagnostic message is reported.

The instantiation is simple:

```
ExprParser p = new ExprParser();
```

You need not to declare variables. Any variable that is not declared or assigned a value has an unknown value: Real or Bool.

However, computing with unknown values in many cases results in unknown results. One can

- A string of the variable's name.
- Its value, either an AADD or a BDD.

Note, that we model a variable that has no value assigned to have a value AADD.Real or BDD.X.

```
p.eval("var a := aaf(1.0, 2.0, 1)");
p.eval("var b := range(2.0, 3.0)");
p.eval("var c := 3.0");
```

Now you can evaluate an expression on a, b, c:

```
var result = p.eval(“a+b∗c”);     // result is of type AADD or BDD.
AADD  aadd = p.evalAADD(“a+b∗c”); // aadd is of type AADD.
BDD    bdd = p.evalBDD(“a+b∗c”);  // bdd is of type BDD.
```

Note, that the first call will return an object of type AADD or BDD, depending on the expression that was parsed. It is up to the user to cast it to the right class.

The second and third calls check the results to be of type AADD or BDD, respectively, and return objects of the respective class. In the 2nd call this matches, and an AADD is assigned to aadd. However, the 3rd call evaluates an expression a+b*c that is results in an AADD; it throws an exception as a BDD type is expected.

The following constants are pre-defined:

- For AADD: `PI, E, Real`
- For BDD: `True, False, Bool`

The following functions are pre-defined for BDD:

- `BDD ite(BDD, BDD, BDD)`
- `BDD not(BDD)`
- `Conditions sat()`

The following functions are pre-defined for AADD:

- `AADD AAF(double, double, int)`
- `AADD Range(double, double)`
- `AADD ITE(BDD, AADD, AADD)`
- `AADD Exp(AADD), Sqrt(AADD)`
- `AADD Intdt(AADD, AADD)`
- `AADD Constrain(min, max)`
- `Conditions SAT()`

5.3 User-defined functions

A new function can be defined as follows:

```
void userFunctionCallsExample() throws ExprError, ParseError {
    p.eval("fun f(x) := x*x*x");
    System.out.println("f(2) = "+p.eval("f(2)");
}
```

The output is:

```
f(2) = 8.0
```

Annex A: **API of AADD, BDD, Conditions**

A.1 **DD public methods/fields**

The class DD provides the following fields and methods that are common to AADD and BDD:

```
/** Returns true if the node is a leaf. */
public boolean isLeaf() { … }

/** Returns true if the node is an internal node */
public boolean isInternal() { … }

/** Returns the number of leaves. */
public int numLeaves() { … }

/** Returns the height of the tree. */
public int getHeight() { … }

/** Returns the Value of a leaf. */
public ValT getValue() { … }

/** Returns a string representation */
public String toString() { … }

/** Returns a Json string */
public String toJson() { … }
```

A.2 AADD public methods/fields

The class AADD extends DD provides the following public methods and fields:

```
    public static final AADD Real
    public static final AADD Empty
    public static final AADD Infeasible

    /** Threshold for call of LP solver. */
    public static double LPCallTh = 0.001;

    /** Threshold for calling reduction of AADD. */
    public static double joinTh   = 0.001;

    /** Creates a new leaf with a scalar as value. */
    static public AADD scalar(double value)

    /** Creates a new leaf with an affine form as a value. */
    static public AADD range(double min, double max, String name)

    /** Creates a deep copy. */
    public AADD clone()

    /** Arithmetic functions on AADD. */
    public AADD negate()
    public AADD exp()
    public AADD sqrt()
    public AADD log()
    public AADD inv()

    public AADD intersect(double lb, double ub)

    /** Arithmetic operations on AADD. */
    public AADD plus(AADD other)
    public AADD minus(AADD other)
    public AADD times(AADD other)
    public AADD times(double other)
    public AADD div(AADD other)

    /** Comparison operations on AADD yield BDD. */
    public BDD Lt(AADD other)
    public BDD Le(AADD other)
    public BDD Gt(AADD other)
    public BDD Ge(AADD other)

   /** Calls LP solver to compute precise bounds. */
    public Range getRange()
```

A.3 **BDD public methods/fields**

```
public static BDD True
public static BDD False

public static BDD constant(boolean value)
public static BDD variable(String varname)

/**
 * Clone method. Copies the tree structure, but not conditions.
 * Also does reduction “on the fly”.
 * The leaves are not copied for BDD.
 */
public BDD clone()

public BDD not()
public BDD and(BDD other)
public BDD or(BDD other)
public BDD xor(BDD other)
public BDD nand(BDD other)
public BDD nor(BDD other)
public BDD xnor(BDD other)

/**
 * The ITE function merges BDD by an if-then-else-function.
 * Note, that the condition itself that is this BDD, is also a BDD.
 * The parameters are not changed.
 */
public BDD ite(BDD t, BDD e)

/**
 *  The ITE function merges two AADD by an if-then-else-function.
 *  Note, that the condition itself that is this BDD, is also a BDD.
 *  The parameters are not changed.
 */
public AADD ite(AADD t, AADD e)

// Returns the number of paths that go to the value true resp. false
public int numTrue()  // same as numSAT
public int numFalse() // same as numUnSAT
```

Annex B: **Revision history**

Version 1.0

- Initial version published on MAVEN.

Version 2.0

- Operator overloading and re-worked API.
- Multi-threaded computation of LP problems.
- Experimental: display of graphs and AADD trees.

Experimental features in development for future versions

- Piecewise linear modeling of nonlinear functions with higher accuracy
- Gaussian noise symbols
- Garbage collection on noise symbols which reduces over-approximation
- C/C++ support
- Constraint nets

jAADD will provide means to model and solve constraint nets. The equation statement of the form (not yet in master branch; highly instable testing)

```
eqn VariableName == Expression
```

The equation statement will search for assignments in the variables at the left and right side of the equation such that the equality holds for every element of the variables at the left and right side.

Var a = Range(1,5) // constraint for eqn. ; can only be restricted to smaller values

Var b = Range(4,5) // constraint for eqn.

Var c = Real

Var d = Real

EQN a*b == b+c*d

Then, we can solve the equation and print the feasible range of each value.

Annex C: **Further Literature on AADD**

[1] C. Zivkovic, C. Grimm: Nubolic Simulation of AMS Systems with Data Flow and Discrete Event Models. In: Design, Automation and Test in Europe (DATE). Mar. 2019, pp. 1457–1462. DOI: 10.23919/DATE.2019.8715278.

[2] C. Zivkovic, C. Grimm, M. Olbrich, O. Scharf, E. Barke: Hierarchical Verification of AMS Systems with Affine Arithmetic Decision Diagrams. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 38.10 (Oct. 2019), pp. 1785–1798. DOI: 10.1109/TCAD.2018.2864238.

[3] C. Grimm, M. Rathmair: Dealing with Uncertainties in Analog/Mixed-Signal Systems (Invited). In: Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. DOI: 10.1145/3061639.3072949.6