AADD Multi-Platform User’s Guide  
Version 0.1



**TU Kaiserslautern**

**Chair of Cyber-Physical Systems**

Hagen Heermann, Christoph Grimm, Carna Zivkovic

Logo TU Kaiserslautern  

The AADD library is developed at the University of Kaiserslautern-Landau 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](#_Toc109644014)

[1 An introduction to AADD 4](#_Toc109644015)

[1.1 Intervals 4](#_Toc109644016)

[1.2 Computing with intervals 4](#_Toc109644017)

[1.3 Affine arithmetic 5](#_Toc109644018)

[1.4 BDDA and AADD 6](#_Toc109644019)

[1.5 Modeling dynamic systems with AADD and AADD streams 9](#_Toc109644020)

[2 Using jAADD from Kotlin and Java 10](#_Toc109644021)

[2.1 “Hello world” examples 10](#_Toc109644022)

[2.2 AADD Builder and AADD operations 11](#_Toc109644023)

[2.3 Storage of internal errors 14](#_Toc109644024)

[2.4 BDDA Builder and Boolean expressions 17](#_Toc109644025)

[2.5 Example: PI control loop 18](#_Toc109644026)

[3 Conditions, Conditional Statements and Iterations 20](#_Toc109644027)

[3.1 From decision variables to conditions 20](#_Toc109644028)

[3.2 Conditional statements 21](#_Toc109644029)

[3.3 Example: water level monitor 22](#_Toc109644030)

[4 Input/Output 23](#_Toc109644031)

[4.1 Conversion to a string 23](#_Toc109644032)

[4.2 Conversion to JSON 23](#_Toc109644033)

[4.3 AADD and BDDA traces 24](#_Toc109644034)

[Annex A: API of AADD, BDD, Conditions 25](#_Toc109644035)

[A.1 jAADD Class Diagram 25](#_Toc109644036)

[A.2 AADD public methods/fields 26](#_Toc109644037)

[A.3 BDD public methods/fields 27](#_Toc109644038)

[Annex B: Further Literature on AADD 28](#_Toc109644039)

# An introduction to AADD

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

## Intervals

Intervals are used for various reasons:

* *Representation of uncertain or unknown values.* Often upper and/or lower bounds of unknown quantities can be given.
* *A precise value cannot be represented.* Quantities such as 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: is mathematically precise, while is wrong.
* *Multiple values are possible.* Often, we leave the precise value for a quantity in a specification open and just give lower and/or upper bounds.

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

We distinguish the following kind of intervals:

* *Empty* – The interval is the empty set {}, and ,
* *Scalar* – The interval has exactly one element: , and ,
* *Finite* – The interval bounds are finite, and ,
* *Open* – One of the bounds is infinite,
* *Reals* – Both bounds are infinite, and the interval is the set (),

AADD supports intervals of the kind *empty, scalar, finite, open,* and all *reals*; furthermore, the AADD library supports intervals of integers (IntegerRange, IDD).

## 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 be a function on the reals, and is its interval-extension. Then, for :

For example, if we add two intervals then the sum of two intervals contains all sums of its real-valued elements.

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

For example, consider and . Then shall be subset of .

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

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

In addition to classical “textbook” interval arithmetic, AADD

* considers the IEEE 754 floating point traps *NaN* (not a Number) and *Infinity* (Overflow). *Infinity* is used to represent an open interval bound. *NaN* is considered as an empty set result.
* ensures safe inclusion of the true results in the presence of roundoff-errors.

## Affine arithmetic

Range arithmetic allow some *over-approximation*. For example, we know that . 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:

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

In longer computational chains, such over-approximations accumulate and eventually lead to an unbounded interval (wrapping effect). To avoid the wrapping effect, AADD uses *affine arithmetic[[1]](#footnote-1)* with some extensions. This avoids the wrapping effect by tracking linear dependencies in affine forms. An affine form is a linear combination of independent noise terms:

where the coefficients are floating point numbers, and the noise variables are unknown reals from the interval . Now, the interval can be represented as ; then . However, can also be represented as ; then we get which describes an interval .

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 .

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

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

The following rules are used for the interaction between IA and AA:

1. For *finite intervals*, calculations are done by both affine and interval arithmetic, and after each operation, the interval bounds are set to the intersection of both IA and AA interval bounds.
2. *Open intervals* including Reals are represented as an interval and computed only by IA; the Affine Arithmetic parameters are then:
   * Interval bounds are used (minimum, maximum), and at least one of them is infinite.
   * Affine Arithmetic parameters are not used.
3. When a NaN due to an invalid operation may occur, the resulting interval includes only the values that are not NaN; infinite may be used to represent e.g. poles.

## BDDA 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[[2]](#footnote-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 that model the possible Boolean outcomes of a Boolean function and *NaB* (Not a Bool) that models the case where due to the continuous part a leaf True or False is not feasible. 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.

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.



Figure 1‑1: BDD of as an ITE function and as a BDD.

Boolean decision variables are just unknown variables. A BDD that models a Boolean function by the ITE function could be written as:

It would be represented by the following BDD:



jAADD provides a very special and basic implementation of Reduced, Ordered BDD: BDDA

*If you need some BDD implementation without interaction to or support for arithmetic operations don’t use jAADD, have a look at CUDD and many other “pure discrete” BDD packages.*

Conditions: linear constraints as decision variables for BDDA

Linear constraints are conditions of the type , where is an affine form. Nonlinear arithmetic expressions are brought to this form as well: every arithmetic expression, in affine arithmetic, results in an affine form, and all kind of comparisons can be brought to the form . (Note, that we do not consider equalities here; equality and Floating-point representations can lead to many issues).

BDDA

BDDA are a special kind of BDDs where the decision variables can be linear constraints. They are not intended to replace BDDs, but to represent the interaction between predicates that are constraints of non-Boolean values (e.g., x >= 0.0), and the ranges that the non-Boolean values (e.g., x) can take. For example, let there be a Boolean variable , and let be linear constraints on real-valued functions on ; e.g. and . Then we get for the example above the representation with an ITE function:

Or, as figure:



Figure 1‑2: BDDA for the example.

The use of linear constraints as decision variables in BDDA introduces dependencies between the discrete and the continuous domains: for conventional, discrete BDD, all decision variables are just unknown Booleans. For BDDA, they can be mixed with linear constraints of type . Such a condition with an affine form is a linear inequation on the noise symbols as unknowns:

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

* **Condition set** of a path to a leaf : The set of all conditions on internal nodes on a path from the root of a BDDA to the leaf, maybe negated, such that is selected.
* **Path condition** of a path to a leaf : The conjunction of all conditions in .

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

The condition is the “highest” node in the BDDA. The node with the condition is hence a subtree below the condition . Hence, 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** and assigned to the leaf Infeasible. Note, that the above BDDA for all feasible cases has the result True. It can hence be reduced to a single leaf with the value True.

AADD

AADD are BDDA where the leaves are affine forms instead of True, False, NaB (not a Boolean). As an example, let the ITE function be a function that takes AADD for the if/then parameters, and that returns the respective parameter as result. For example, let , then

is represented by the following AADD:



Figure 1‑3: AADD for the above expression.

Note that in AADD, 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 with an affine form with path set is now obtained by a linear program:

Likewise, for the maximum:

Note, that there exists not necessarily a solution; then, the leaf is infeasible. Infeasible leaves can be omitted (reduced) as there exists not assignment such that is reached.

Operations on BDDA and AADD

Operations on BDDA resp. AADDs are defined such that for an operation it holds:

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

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 evaluates trivially to *FALSE*; none of these relations needs further variables.
* A new decision variable that is a constraint of kind 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 BDDA 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 BDDA and AADD.

## 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 , where the subscript is from an at least partially ordered set that models time, and where the values are either real or Boolean values:

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 is a discrete ordered set and introduces a global synchronization between two or more signals, the system is a *discrete-time system*. If 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:

Now, 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, represents not only the enclosing hull of all reachable signal trajectories.

# Using AADD from Kotlin and Java

We now explain the use of AADD and BDD to compute arithmetic and Boolean expressions. AADD 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 AADD jar file must be in the classpath of a Java program, and the library and the used classes, methods, fields must be imported.

When using Maven or Gradle, use the following dependency (with updated version):

<dependency>

<groupId>com.github.tukcps</groupId>

<artifactId>AADD</artifactId>

<version>3.0.0-SNAPSHOT</version>

</dependency>

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

## “Hello world” examples

Hello BDD example in Java and Kotlin

Keep in mind that the AADD library does not intend to replace more efficient SAT approaches; BDDs are there to represent the interaction between computations on the Reals or Integers and discrete decisions; purely discrete analysis might be better done via a SAT solver.

The following example shows how the AADD library can be used in different languages, i.e., Java and Kotlin. We begin with the simple example from section 1 in Java:

import com.github.tukcps.aadd.\*; // imports

class HelloBDD {

public void main() {  
 DDBuilder builder = new DDBuilder(); // Builder that creates DD

BDD c: BDD = builder.boolean("c");

BDD b: BDD = builder.boolean ("b");

BDD a: BDD = builder.boolean ("a");

BDD f = a.and(b).or(c); // … compute something

system.out.println("f(a,b,c) = "+f);// … and print result

}

}

In the above example, we use a DDBuilder object that creates BDDs. The builder provides several methods to create BDD, AADD, and IDD objects. Internally, a builder maintains the noise symbols and indexes that are shared by all instances created by the same builder.

The DDBuilder object maintains noise symobls and indexes. It is not allowed to apply operations on AADD or BDD from different builders; doing so will throw an exception.

In the example, the method boolean(String: id) of DDBuilder creates an object of the class BDD that is a Boolean variable with a name given as parameter. As shown above, in Java one must call member functions and, or, etc. of BDD. We can also print a BDD via its member function toString() that is automatically called when used where a String is expected. In Kotlin, the same methods can be called in a more readable way by using the overloaded operators and lambdas that implement a DSL (Domain Specific Language):

import com.github.tukcps.aadd.\* // import of jAADD

fun main() {

DDBuilder { // … as in Java, but shorter

val c = boolean("c")

val b = boolean("b")

val a = boolean("a")

val f = (a and b) or c // … and compute something

println("f(a,b,c) = $f") // prints result

}

}

The resulting output is for both the Java and the Kotlin code as expected:

f(a,b,c) = ITE(c, true, ITE(b, ITE(a, true, false), false))

Hello AADD example

The use of AADD is like the use of BDD: A builder must be used. In the following we show just the Kotlin example; Java is straightforward using the same functions.

import com.github.tukcps.aadd.\*

fun main() {

DDBuilder {   
 val x = real(-1.0 .. 1.0, "x")  
 val f = ite(x greaterEquals 0.0, x-100.0, x+100.0)  
 println(" f = $f")

}  
}

In the example,

* The method real(Range, String) creates an AADD object that represents the range using a noise symbol associated with the given string.
* “greaterEquals” is a comparison operator on AADD whose result is a BDD.

The result is an AADD, as in the example in Section 1:

f = ITE(1, [-100,00; -99,00], [99,00; 100,00])

The noise terms are not printed by default; instead, the interval computed by the LP solver is printed; the number 1 in the first ITE parameter is the index in which we can access the condition.

In the following, we give examples in Kotlin and leave the translation to Java to the readers.

## AADD Builder and AADD operations

In AADD, we consider Intervals and affine forms as a special case of an AADD that is just a single leaf. The class Builder provides methods that allow us to create AADD and BDD.

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

the method real(value: Real) of a DDBuilder instance must be called, e.g.:

val scalar = real(a0); // in Scope of a builder

To get an AADD that is represented by an affine form

which spans an interval that is dependent on the noise variable with index, one has to use the method real(range: ClosedRange<Real>, id: String) where “id” can be an arbitrary string that uniquely identifies the noise symbol as a human-readable shortcut:

val range = real(min .. max, id); // min, max: double, id: String

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

val a = Empty; // Empty set; no real number in it.

val b = Reals; // All real numbers are in it, but not NaN or Infinity

Note, that Empty is a single 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.

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

import com.github.tukcps.jAADD.\*

fun instantiation() {

DDBuilder {   
 val scalar = *real*(1.0)  
 val range = *real*(2.0 .. 3.0, “range”)

val reals = *Reals*  
 val empty = *Empty*  
 println("scalar = " + scalar)  
 println("range = " + range)  
 println("reals = " + reals)  
 println("empty = " + empty)

}

}

*The example 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 // Java: a.plus(b);

Or, for :

a = a + b \* c/d-e // Java: a.plus(b.times(c).div(d)).minus(e);

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 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 that are independent which means they use affine forms with different noise symbols. Compute and and print and compare the results.*

**val** a = *real*(1.0 .. 2.0, "a")  
 **val** b = *real*(1.0 .. 2.0, "b")  
 println(" a-a = " + (a - a))  
 println("but a-b = " + (a - 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:**

**fun** ellipsoid\_excercise() = DDBuilder {  
 *// Volume of ellipsoid = 4/3 pi a b c where a, b, c=[1,10]* **val** a = *real*(1.0 .. 10.0, "a")  
 **val** b = *real*(1.0 .. 10.0, "b")  
 **val** c = *real*(1.0 .. 10.0, "c")  
 **val** pi = *real*(3.141, 3.142, "pi")  
 **val** vol = *real*(4.0/3.0) \* pi \* a \* b \* c  
 println(**“**Volume **=** $vol”)  
}

The result computed by AADD 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 exist operations that return other kind of AADD, e.g., multiplication with 0 returns an AADD of value 0.

## Representation of internal errors

Computations on affine forms lead to internal errors due to rounding and approximation of the results of nonlinear operations. To not distort the actual result interval, these errors need to be stored in the affine form.

The implementation offers two different ways of storing internal errors: Either in one single error term, the r-term, or in multiple additional noise symbols. While the first option offers shorter computation times, the second option reduces the error growth in long iterative computations.

The default configuration can be found in the configuration file of DDBuilder (jAADDConfig.json):

{

“noiseSymbolsFlag”: false,

“originalFormsFlag”: false,

“maxSymbols”: 200,

“mergeSymbols”: 10,

}

According to the use case, the configuration of an instance of DDBuilder can be adapted:

DDBuilder {

config.noiseSymbolsFlag = true

config.originalFormsFlag = true

config.maxSymbols = 100

config.mergeSymbols = 5

}

In the default case the *noiseSymbolsFlag* is set to false, which means that all errors are stored in a single error term. If it is switched to true, every internal error is stored in an individual noise symbol and approximation errors are mapped to the operation they resulted from. This allows that internal errors from the same source may cancel each other in further computations, which leads to more narrow result intervals.

The following example illustrates the error cancelation and the difference between the two settings:

fun singleErrorTermTest() {

DDBuilder {

val af1 = AF(1.0, 2.0, 1)

val a = af1.log()

val b = af1.log()

val result = a.minus(b)

println(result.radius)

}

}

Since all internal errors are stored in one error term, there is no information on dependencies between the errors. Thus, they are treated as if they were independent and even in a subtraction, the error terms of the affine forms are added to guarantee enclosure of the actual value.

result.radius = 0.05966010114161032

In the example above this means, that the radius of the result is two times the approximation error of the logarithm function plus rounding errors instead of zero.

fun noiseSymbolsFlagTest() {

DDBuilder{

config.noiseSymbolsFlag = true

val af1 = AF(1.0, 2.0, 1)

val a = af1.log()

val b = af1.log()

val result = a.minus(b)

println(result.radius)

}

}

As the approximation errors of both logarithm operations are now stored in the same noise symbol, their dependency is preserved, and their values can be subtracted. So, the radius of the result is much closer to zero and only differs due to rounding errors.

result.radius = 6.661338147750942E-16

The *originalFormsFlag* enables an additional mapping for times and inverse operations for the detection of linear dependencies between approximation errors that resulted from operations on scalar dependent affine forms. Thus, enabling the *originalFormsFlag* without the *noiseSymbolsFlag* has no effect.

Since the computation time in affine arithmetic is directly connected to the number of noise symbols of affine forms, using several noise symbols instead of a single error term leads to higher computation times. Therefore, the *reduceNoiseSymbols* function limits the number of noise symbols in an affine form, in order to obtain feasible computation times for long, iterative calculations.

The function is based on a method by Jorge Stolfi and Luiz Henrique De Figueiredo[[3]](#footnote-4). The authors suggest to “condense” affine forms, which means reducing the number of noise symbols by combining two or more independent symbols into a new one, by adding up their absolute values. This can be seen as a compromise between tracking all errors on their own and merging all errors into one noise variable. To minimize the loss of correlation information, only error terms that are small in relation to the overall range of the affine form should be merged and condensing should be limited on noise variables that represent approximation or roundoff errors. This guarantees that noise variables that were part of the input stay separated, as they might be important to analyse correlations between the result and the input.

This approach is slightly adapted for the *reduceNoiseSymbols* function.

As every operation introduces at least one new noise symbol to track roundoff errors, the *reduceNoiseSymbols* function is called at the end of every operation, before its result is returned. The function checks whether the determined limit of noise symbols per affine form is exceeded and if this is the case, a fixed number of noise symbols is merged until the desired size is reached or no symbols can be merged anymore.

For the merging process, the function differentiates between roundoff errors, approximation errors and external noise symbols that were part of the input. Roundoff errors carry the least amount of information, as they are not mapped to the operation they resulted from and therefore cannot cancel each other later. Thus, they are prioritized for merging.

Approximation errors are only merged, when there is no sufficient amount of roundoff errors left to reduce the size of the affine form. In contrast to roundoff errors, there exists a mapping in the *nonLinearNoise* HashMap for every approximation error, which allows to distinguish them. External noise symbols are completely excluded from merging, as they are generally the most important for the analysis of correlations. Thus, no noise variables with an index less than 10000000 are merged.

Apart from this hierarchy the noise symbols are chosen by the magnitude of their deviation, prioritizing smaller values in order to minimize the loss on information. The chosen noise symbols are removed from the affine form, while their absolute values are added up and stored in a new unique noise symbol in the form.

The following pseudo code illustrates this process:

fun reduceNoiseSymbols():AffineForm {

if (this.xi.size>this.builder.config.maxSymbols) {

**//stores the absolute sum of the merged noise symbols**

var nval = 0.0

**//stores the value of of the minimum deviation**

var mini: Double? = null

**//stores the index of the noise symbol with the minimum deviation**

var mkey = 0

while (xi.size>this.builder.config.maxSymbols){

for (i in 1..this.builder.config.mergeSymbols){

for (entries in xi) {

**//search for the minimum roundoff error**

}

if (**/\*no roundoff errors left to merge\*/**) {

for (entries in xi) {

**//search for the minimum approximation error**

}

if (**/\*minimum found\*/**) {

nval += abs(mini)

xi.remove(mkey)

mini = null

mkey = 0

}

else {**//no internal noise symbols are left**

if (**/\*some noise symbols have been merged\*/**) {

nval += nval.ulp

xi[builder.noiseVars.newGarbageVar()] = nval}

return this}

}

**// the desired number of noise symbols has been merged**

nval += nval.ulp

xi[builder.noiseVars.newGarbageVar()] = nval

}

return this

}

return this

}

The maximum number of noise symbols per affine form, as well as the number of symbols that should be merged can be determined in the configuration by adapting *maxSymbols* and *mergeSymbols*.

A lower maximum size leads to lower computation times, as the computation time for a single operation in affine arithmetic depends on the number of noise symbols of the operands. On the other hand, a lower number of merged symbols leads to an increased number of newly introduced noises symbols and higher computation time, as the reduction function needs to be executed more often. Therefore, a low maximum size with a high number of merged symbols should lead to the best computation time. However, both lowering the maximum size and increasing the number of merged symbols increases the loss of correlation information. Thus, it is not recommended to set the default maximum size of affine forms and HashMaps too low, as this could lead to problems in use cases with more non-affine operations. Instead, the size should be carefully reduced, after gaining more information on the number and kind of noise symbols that are introduced during the computation.

Previous tests have shown that the difference between the maximum size and the number of merged symbols needs to be at least the same value as the number of relevant noise symbols that should never be merged. To determine this number, we must consider the number of external noise symbols as well as the number of different non-linear operations in a computation loop.

## BDDA Builder and Boolean expressions

For symbolic computation on Boolean variables, we must use its symbolic representation by BDDA. 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, and we need to use the builder like for AADD:

BDD Instantiation of BDDA

The AADD Context provides the following methods to create BDD:

BDD constant(Boolean val); // leaf with true resp. false

BDD variable(String uid); // internal node, decision variable uid

Furthermore, fab.True and fab.False are objects (constants) in the current context for which only one single instance exists. In fact, BDDA are a kind of Reduced Ordered BDD that, due to the reduction, have at most one True and one False leaf (and leaves Infeasible, NaB depending on the outcomes of the path condition’s LP problem)

To create a decision variable with an internal node, the method fab.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 [builder.]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.

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

**Solution (Java, in Class that inherits DDBuilder):**

void BDDinstantiation() {  
 BDD a = constant(true); // gets BDD w/ Boolean true or false  
 BDD x = 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(variable(“x”));  
System.*out*.println(“d=” + d + “\ne=” + e );

Note, that the Boolean values are TRUE, FALSE from Java, and True, False from Kotlin.

The resulting output is:

d=true

e=ITE(“x”, true, false)

***Exercise:*** *Explain the results.*

## 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:*

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

isval.times(*scalar*(.5)).plus(piout.times(*scalar*(.5)));  
}

For visualization, see Section 4.

# 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).

## 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 Builder method 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 = *range*(1.0, 3.0, 1);  
 AADD b = *range*(2.0, 4.0, 2);  
 BDD c = a.gt(b); // Kotlin: 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 = *range*(1.0, 3.0, "a");  
 AADD b = *range*(2.0, 4.0, "a");  
 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: will always be , and is hence false.

***Exercise:*** *For as declared above, compute the condition and print the result.*

AADD a = *range*(1.0, 3.0, "a");  
AADD b = *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)

## 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

* IF,
* END,
* ELSE and
* assignS.

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

void iteExample1() {  
 AADD a = *range*(-1.0, 1.0, "a");  
 IF(a.le(*scalar*(0.0))  
 a = a.assign(a.add( 2.0 ));

ELSE()

a = a.assign(a.sub( 2.0));

END();   
 System.*out*.println("a="+a);  
}

The result we get is:

a=ITE(1, [-2,00; -1,00], [1,00; 2,00])

Note, that the number 1 is the index of the 1st condition in the ITE example, the comparison a <= 0.0.

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

* Replace the keywords if and else with IF and ELSE; add a terminating END() at the end of the if and else block.
* Replace assignments x = f(x, y, …) with x = x.assignS(f(x, y, …))

## 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)  
 IF(level gt scalar(10.0));  
 rate = rate.*assignS*(outrate)  
 END()  
 IF (level lt scalar(2.0));  
 rate = rate.*assignS*(inrate)  
 END()  
 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.

# Input/Output

## 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);

## 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 =*range*(1.0, 2.0, “a”);  
 String s = a.*toJson*(a);  
 System.*out*.println(“s = ” + s);  
}

The result is:

s = {

"index": 2147483647,

"value": {

"x0": 1.5,

"xi": {

"4": 0.5

},

"r": 0.0,

"min": 1.0,

"max": 2.0

},

"feasible": true

}

## AADD and BDDA traces

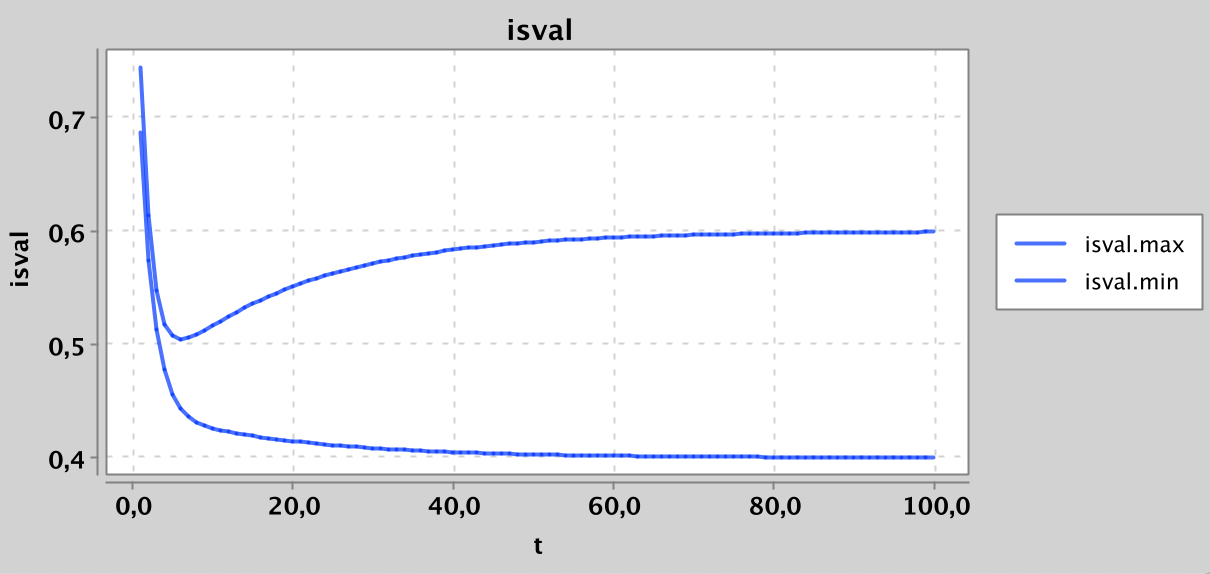
Let’s come back to the PI controller example. It would be nice to not only compute the AADDs over time, but to also trace and show the signals over time. In this case, we have t={1, 2, …, 100}. This can be done via an AADDTrace object.

**void** piControl() {  
 **var** setval = *range*(0.4, 0.6, **"setval"**);  
 **var** isval = *range*(0.9, 1.0, **"isval"**);  
 **var** piout = *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(*scalar*(0.05)));  
 isval = isval.times(*scalar*(0.5)).plus(piout.times(*scalar*(0.5)));  
 t.add(isval, i);

**}**

t.display()  
}

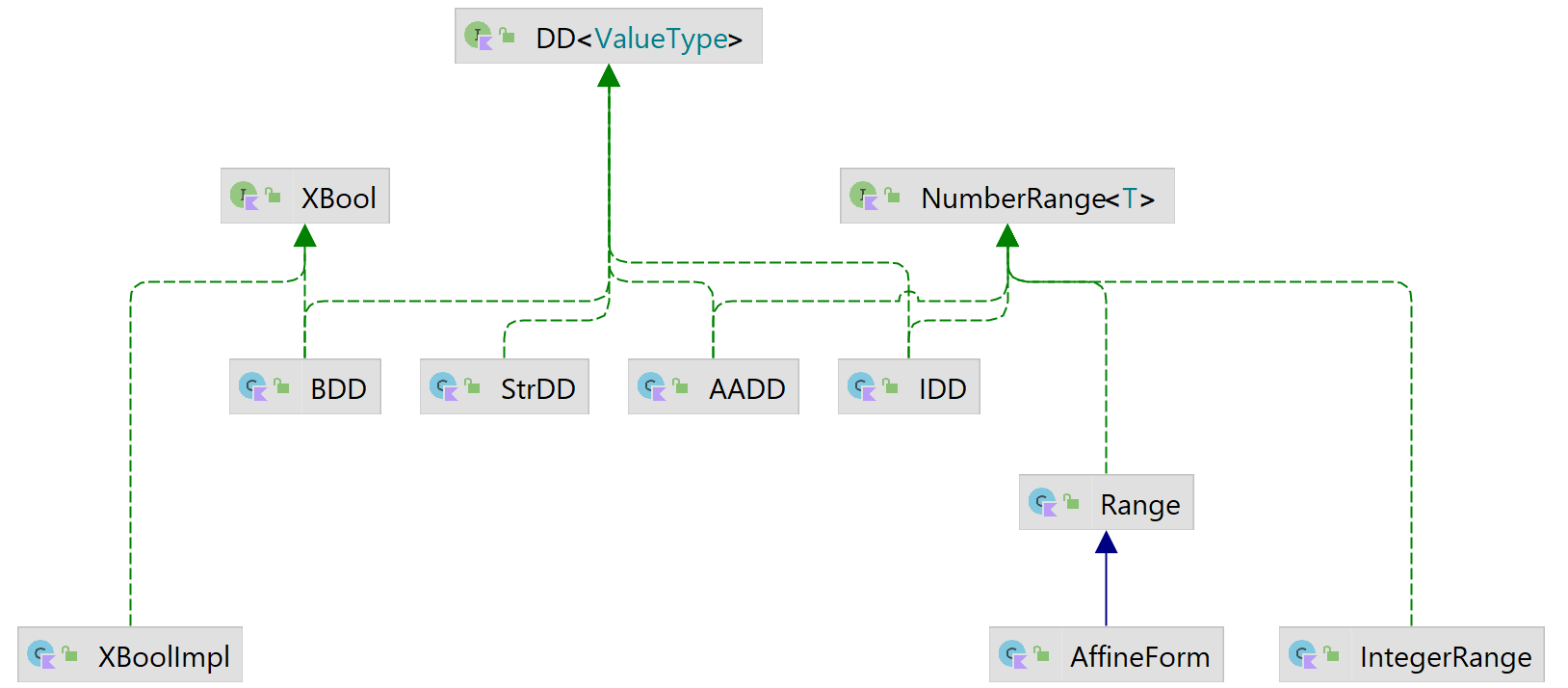
In a window, the following graph is displayed. It shows the minimum and the maximum values of isval:



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)

1. API of AADD, BDD, Conditions
   1. jAADD Class Diagram



interface DDBuilderIF {  
  
 */\*\* Constructors for AADD \*/* fun scalar(value: Double) : AADD  
 fun range(min: Double, max: Double, id: String) : AADD  
 fun range(r: ClosedFloatingPointRange<Double>, id: String) : AADD  
 fun range(r: ClosedFloatingPointRange<Double>, id: Int) : AADD  
 fun range(r: ClosedFloatingPointRange<Double>) : AADD  
  
 fun internal(index: Int, T: AADD, F: AADD): AADD  
 fun leaf(value: AffineForm, status: DD.Status): AADDLeaf  
 fun leaf(value: AffineForm): AADDLeaf  
  
 */\*\* Constructors for BDD \*/* fun variable(varname: String) : BDD  
 fun constant(value: Boolean) : BDD  
 fun internal(index: Int, T: BDD, F: BDD): BDD  
  
 */\*\* Constructors for Affine Forms \*/*  
 fun AF(min: Double, max: Double, symbol: Int): AffineForm  
 fun AF(min: Double, max: Double, symbol: String): AffineForm  
 fun AF(central: Double): AffineForm  
  
 */\*\* Functions for enabling nice control flow annotations \*/* fun IF(cond: BDD): BDD  
 fun END(): BDD  
 fun ELSE(): BDD  
  
 */\*\* Constants \*/* val Reals: AADD // Java: getReals() must be used.   
 val Empty: AADD // … as above, getter for all val.   
 val RealsNaN: AADD  
 val Infeasible: AADD  
 val InfeasibleB: BDD  
 val True: BDD  
 val False: BDD  
 val AFReals: AffineForm  
  
 fun aaddFromJson(json: String): AADD  
 fun bddFromJson(json: String): BDD  
  
 */\*\* Settings \*/* var toStringVerbose : Boolean *// Sets verbosity of toString method* var lpCallTh : Double *// Set range from which on the LP solver will be called* var joinTh : Double *// Set size of noise symbol below leaves are joined* var lpCalls: Int  
}

* 1. AADD public methods/fields

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

*/\*\* Creates a deep copy. \*/*  
 public Any clone() // Needs cast to AADD  
  
 */\*\* Arithmetic functions on AADD. \*/* public AADD negate()public AADD exp()public AADD sqrt() public AADD log() public AADD inv()

public AADD intersect(AADD other)

public AADD constraintTo(Range other)   
  
 */\*\* 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()

* 1. BDD public methods/fields

/\*\*  
 \* Clone method. Copies the tree structure, but not conditions.

\* Also does reduction “on the fly”.   
 \* The leaves are not copied for BDD.  
 \*/  
**public** Any clone() // Needs cast to BDD

**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

1. 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

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>. [↑](#footnote-ref-1)
2. 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>. [↑](#footnote-ref-3)
3. Jorge Stolfi and Luiz Henrique De Figueiredo. “Self-validated numerical methods and applications”. In: Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro. Citeseer. Vol. 5. 1. Citeseer. 1997 [↑](#footnote-ref-4)