Ludwig-Maximilians-Universität
München
Institut für Informatik


Bachelorarbeit

# Extending the JavaSMT Framework with the Apron Library for Numerical Abstract Domain with subsequent Usability Assessment

*Winnie Lilith Sofia Ros*

supervised by
Prof. Dr. Dirk Beyer
Mentor: Daniel Baier

Abgabedatum: 22.12.2023

**Abstract** JavaSMT enables its users to utilize a common API for a great variety of SMT solvers, enabling the solving of formulas in a multitude of different theories. Numerical abstract domains, such as polyhedrons, also enable the solving of first-order logic with equality formulas for numerical properties while offering distinct features to SMT solvers. In this thesis, the Apron Numerical Abstract Domain Library was added to JavaSMT, enabling users to utilize the capabilities of interval-based domains and its features in a newly developed common public API, similar to those used for SMT solvers. But we also want to expose Apron and its multiple domains as a quasi SMT solver for the SMT theories of Boolean, Integer and Rational with some limitations for Boolean. Finally, this thesis evaluates the usability of the integration of the Apron library as a quasi SMT solver. In conclusion, the current realization of the quasi solver cannot fulfill all necessary requirements. But the limitations are described to provide starting points for future work.

# Contents

# 1. Introduction

Before actually using a program inside an application for solving real-world problems, it can be useful or sometimes even mandatory to verify if the program is running without errors. To prevent errors, one has to think about the safety properties and specifications of the program that should always be fulfilled. A typical example of a violated safety property is an out-of-bound array access. There are many different theories for encoding and checking such problems. One way is to use Satisfiability Modulo Theories (SMT), a theory that offers the possibility of using several different theories like Linear Integer Arithmetic. For actually proving if the properties are satisfied, they are represented by some first-order logical formulas that are checked with the help of an SMT solver. As the properties and specifications can be of very different shapes, there are many kinds of SMT solvers specialized for solving various kinds of formulas supporting different theories. A comfortable way to profit from the strengths of the diverse solvers and therefore theories is provided by JavaSMT[3]. This framework unites several solvers under a common API[1]. The common API facilitates using the available solvers, as one can access each solver the same way.

Numerical abstract domains allow similar representations of properties as in SMT while utilizing a distinct approach. With their help, it is possible to make statements about the numerical properties of a program. A library that combines different numerical abstract domains and offers, like JavaSMT, a common API to work with different domains is the APRON Numerical Abstract Domain Library (Apron) [21].

The idea of this thesis is to expand the portfolio of JavaSMT with numerical abstract domains using the Apron library. The difficulty will be to integrate Apron as an SMT solver, as it does not have distinct SMT functionalities. Apron does not offer typical SMT solver features like boolean operations or possible variable allocation to satisfy given constraints. However, this thesis aims to find out if it is possible to use the library for numerical abstract domains directly as a quasi SMT

[1] An API is an application programming interface that gives computer programs the possibility to communicate with each other.

solver. The approach of integrating the Apron library into JavaSMT concentrates on the Integer and Rational SMT theories, as numerical abstract domains are also made to solve numerical properties and consequently offer a fitting setup for these theories. Subsequently it will offer some Boolean theory features. In conclusion, the thesis will present the usability of the Apron solver and which parts would need more work to be done for a successful integration in the JavaSMT framework.

Chapter 2 will present some related work to JavaSMT as well as the combination of abstract interpretation and SMT solvers. The following Chapter 3 will explain all the needed background theories, naming SMT and JavaSMT, numerical abstract domains, and the Apron library. Moreover, it will argue how numerical abstract domains can model an SMT solver. Chapter 4 will explain how the implementation of the Apron SMT solver works and where the limitations of the implementation of the quasi SMT solver are. Subsequently, it is shown how an evaluation of the new solver with the help of the software verification framework CPAchecker[8] could look and why it is not yet an option (Section 5.1). Moreover, it is presented in Section 5.2 to which extent the quasi solver has to be improved for enhancing JavaSMT. Finally, the thesis is concluded with a summary of the findings and possible future work (Chapter 6).

# 2. Related Work

In this chapter, some related work to this thesis is introduced. It will describe on the one hand work related to JavaSMT[3] and on the other hand work related to Apron[21]. Moreover, it will present thesis about the idea of combining abstract interpretation and SMT solvers.

PySMT[19] offers a Python-based common API for SMT solvers that offers the possibility to build formulas independent of the Solver API. It uses the native Python APIs of the solvers and converts them into its own API, comparable with the architecture of JavaSMT. It has an interface for seven different SMT solvers plus an SMT-LIB2[5] binding, which allows using any SMT-LIB2[2] compliant solver. jSMTLIB[12] is a Java library that supports the SMT-LIB2 standards and interacts with several SMT solvers. It can also be used to check files for SMT-LIB2 compatibility. ScalaSMT[9] offers, similar to jSMTLIB, an abstraction of SMT-LIB2. ScalaSMT is able to transform Scala syntax to SMT-LIB2 standard and back.

ELINA is a library for static analysis with numerical abstract domains like Apron[21]. It uses the Apron library for the domains but extends it with their own approach to improving the performance of sub-polyhedron domains. The performance improvement is achieved by decomposition.

Previous works focused on the combination of abstract interpretation and SMT solvers are now presented. Jan Peleska, Elena Vorobev, and Florian Lapschies[25] present an approach to using the widening feature of abstract interpretation inside an SMT solver. They want to use the automatic detection of loop and recursion invariants of abstract interpretation and use this feature when the SMT solver needs to approximate invariants for effective verification. The paper "Block-Wise Abstract Interpretation by Combining Abstract Domains with SMT"[22] introduces an approach and implementation of a combination of abstract domains with SMT. The idea is to analyze a program block, not statement-wise. It abstracts each block into an abstract element of a chosen domain, translates that into an SMT formula,

[2]SMT-LIB2[5] is an initiative founded in 2003 that aims to develop a standard syntax for input and output languages for SMT solvers. Most of the existing solvers offer the possibility to read formulas fulfilling the SMT-LIB2 standard as valid input.

computes it, and translates the result back into an abstract element. The widening operator of abstract domains is used to handle loop invariants.

# 3. Background and Motivation

In this chapter, the background knowledge that is needed to understand this thesis is explained. Firstly, it starts with the Satisfiability Modulo Theories, abbreviated SMT, and its application, the SMT solvers. This leads to JavaSMT[3], a framework for different SMT solvers. Next, it is shifted to the numerical abstract domain, then the APRON Numerical Abstract Domain Library (Apron)[21] and how it can be interpreted as an SMT solver.

## 3.1 Satisfiability Modulo Theories (SMT)

SMT is an NP-hard decision problem for formulas expressed using first-order logic with equality. The overall goal of Satisfiability Modulo Theories, abbreviated with SMT, is to evaluate whether a logical formula $\varphi$ is satisfiable with respect to some background theory $\tau$. The theory determines the interpretation of the symbols used in $\varphi$[5]. For the concrete computation of a formula, SMT solvers are used.

SMT solvers are programs or tools that aim to solve the SMT problem for a subset of theories. These theories have a wide variation, from integer numbers to functional arrays with extensionality. SMT is an abstraction of the Boolean satisfiability problem called SAT. SAT solvers need propositional logical formulas as input, whereas SMT solvers support various higher-order formulas based on theories like Integer or Arrays. The solvers are used in many different fields, like processor verification, equivalence checking, un-/bounded model checking, static analysis, and more[6]. Broken down, one can say that SMT solvers evaluate if there is a model of $\tau$ that makes $\varphi$ true for the theory it supports[5]. An example could be the following: We have some properties $x < 6$ and $x > -4$ within the Integer-theory with x as an integer variable. An SMT solver checks if there exists a satisfiable variable assignment for the conjunction of the two properties; in this case, for example, $x = 3$. There are two different types of implementation of a solver. One is the eager encoding to SAT, where logical formulas over some theories are transformed into propositional logical formulas and passed on to a SAT solver. The other approach is called lazy SMT solving. Here two solvers are used: first a SAT solver for finding

a solution for the boolean structure of the formula, and then a theory solver for checking whether there is a model according to the underlying theory of the formula. The framework of an SMT solver consists of four parts:

1. An underlying logic, for most of the cases first-order logic with equality.
2. A background theory against which satisfiability is checked.
3. A set of valid input formulas, which the solver can handle.
4. A set of features that the solver provides, for example identifying the unsat-core, a sound sub-set of unsatisfiable formulas.

Here is an example of an SMT solver for linear arithmetic. The underlying logic is Linear Integer Arithmetic (LIA). The set of valid input formulas is the union of in-equations and equations between linear polynomials formed with $+, -, *, /$. A solver accepts a conjunction of valid in-/equations and returns SAT (plus model) for satisfiable input formulas, UNSAT for the opposite, or UNKNOWN as an output[5]. For easy comparability and a standardized input/output language as well as a unified description of logics and theories for all SMT solvers, SMT-LIB[5] was launched in 2003. The goal is improving the research and development of SMT with standard rules for terminology and unified input/output formulas. The current version 2.6 of SMT-LIB, referenced as SMT-LIB2, also provides a description of logics and theories in SMT. This standard makes the usage of SMT solvers easier, as the user can work likewise with different SMT solvers supporting the SMT-LIB2 standard.

## 3.2 JavaSMT

Although SMT-LIB2 makes handling and interaction with SMT solvers easier, it does not support features like Interpolation[3] or Optimization. But accessing the solvers directly via their API for using these features makes it difficult to switch between solvers as their interfaces are not transferable, as discussed in Section 3.1. To avoid solver lock-in and benefit from capabilities offered by multiple available solvers, JavaSMT provides a framework for accessing several solvers with the help of a unified interface. Currently, JavaSMT exposes the following SMT solvers: Boolector[24], CVC4[7], CVC5[4], MathSAT5[11], OptiMathSAT[27], Princess[26], SMTInterpol[10], Yices2[17], OpenSMT2[20] and Z3[16]. The framework makes a

[3]Even though interpolation is not yet available, there is a proposal for including this feature in SMT-LIB2 (`https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf` [12.12.0223]).

straightforward, unified interaction with the solvers possible. It combines the usage of certain features of the specific solvers as well as easy switching between solvers. JavaSMT is written in Java and interacts with the solvers directly via their API. For this purpose, several Java Native Interface (JNI)[4] wrappers were developed to communicate with non-Java solvers. In the case of the integration of the Apron library[21], the core element of this thesis, a Java interface is already available and used. Moreover, JavaSMT is typesafe and creates only a small overhead.

The following describes the architecture of JavaSMT, also visualized in Figure 3.1. The main components are the `SolverContext, FormulaManager` and the `ProverEnvironment`. The `SolverContext` holds the `FormulaManager` as an instance variable. The `SolverContext` has the function to load the SMT solver, keep track of all created objects, and manage the memory of the underlying solver library[3]. With the `FormulaManager` and the `FormulaCreator`, one can create formulas in an existing theory, for example, `x + 0.4` in the theory of real numbers, with x as a rational variable. Formulas can be combined with the operators for the specific theory. For example, two instances of the class `IntegerFormula` combined with a valid operator like + produce a new instance of the same class. This recursive structure of building formulas lets them grow successively. Moreover, it guarantees type safety as one cannot combine non-compatible formulas from different theories. For checking the satisfiablity of the formula, they are passed to a `ProverEnvironment`. Those formulas have to be of the type `BooleanFormula` and are saved on the assertion stack. A prover manages an incremental assertion stack consisting of conjuncted formulas. In this case, incremental means that the stack is growing by gradually adding new formulas to the stack and potentially deleting components in reverse order. A formula on the assertion stack can be seen as a constraint, and the conjuncted set of constraints of a prover can be checked for satisfiability. Besides checking for satisfiabilty, there are features that are only available if the underlying solver supports them. The model feature presents concrete examples that meet the formula. It suggests concrete variable assignments for which the conjuncted constraints on the assertion stack are satisfiable. Or, on the opposite, the unsat core, which breaks down an unsatisfiable formula to a smallest subset of properties that is still unsatisfied. Interpolation can show relationships between different formulas and theories. This process is the same for all solvers available in JavaSMT. The consistent approach makes it easy to use different solvers for checking the same set of properties. If a new solver is added to the framework, the developer has to link the classes and

[4]JNI stands for Java Native Interface. It is a framework that gives the possibility to use native applications and libraries written in another language than Java.

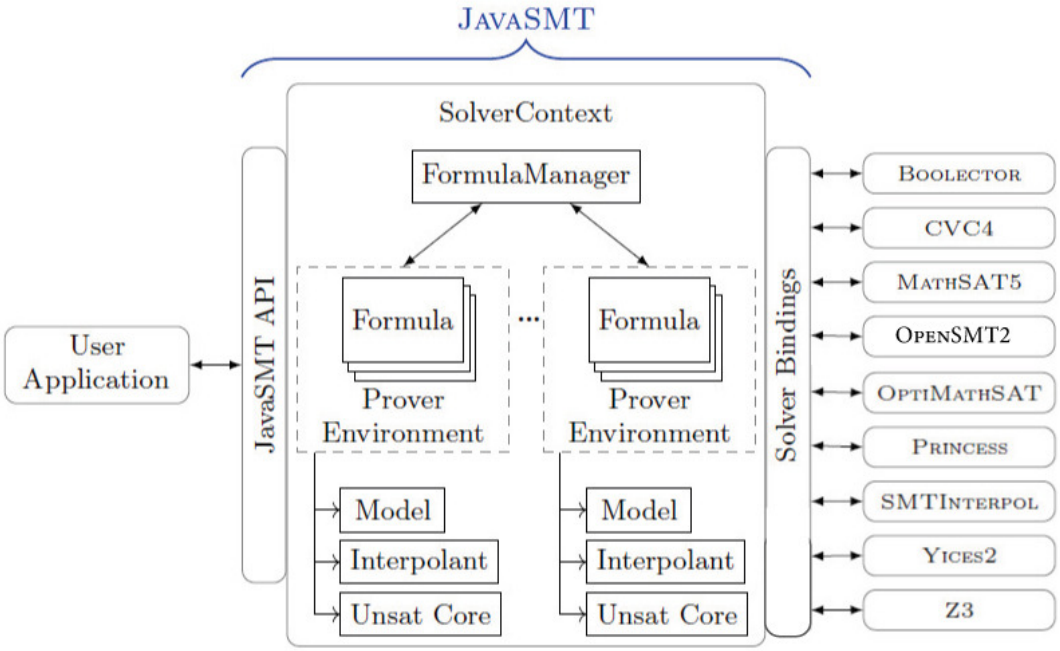



Figure 3.1: Overview of JavaSMT[3].

functionalities of the JavaSMT framework to the equivalent classes and functionalities of the new solver. The interfaces of JavaSMT determine how this has to be done.

## 3.3 Numerical Abstract Domain

Numerical abstract domains are domains of abstract interpretation. Abstract interpretation, in turn, is a theory that can be used, for example, in performing static analysis. Static analysis is a way of finding unwanted behavior in a program without actually executing the code. It can be used, for example, to find safety violations in software. Abstract interpretation was first formalized by Radhia and Patrick Cousot in the 1970s[14]. The general goal of abstract interpretation is to prove if a program is doing (the semantics of the program) what it is supposed to do (the specifications of the program). The semantics of a program describe all possible options of execution for all possible or valid inputs. Because testing every possible option takes too long or is sometimes even impossible, abstract interpretation tries to formalize the semantics by overapproximating them. That is a way to shift them to a higher level of abstraction, or, to say, a sound superset of the concrete program semantics. Through this method, it is possible to get information about the behaviors of a program without knowing or executing every possible path of the as-

sociated reachability graph[5]. A way of approximating the semantics of a program is to form abstract domains for the properties of the program. As the properties have a great variety in what they represent, each one needs a fitting abstract domain. In cases of numerical properties like $x + 4 \leq 6$ with $x$ as a numerical variable, we work with numerical abstract domains. With the help of numerical abstract domains, we can gain information about the numerical invariants and numerical bounds of a program, such as detecting invalid array access or checking loop termination conditions. There are different numerical domains, mainly intervals, also called box, octagon, and polyhedron, which differ in how precisely they represent a numerical property. Intervals have the form $X_i \in [a_i, b_i]$[14][6], octagons encode properties of the form $\pm X_i \pm X_j \leq \beta$[23][7] and polyhedron $\sum_i \alpha_i X_i \geq \beta$[15][8]. Moreover, interval is a non-relational domain, whereas octagon and polyhedron are relational between different variables. For example, $x \in [3, 6] \wedge y \in [-2, 0]$ is non-relational, but $3x + 4y < 9$ is for $x, y$ as numerical variables[23]. Given that a polyhedron can present linear dependencies among variables, it can reduce the value range significantly compared to intervals. If abstract domains are too costly or otherwise imprecise, widening and narrowing operators should be used to fine-tune the trade-off between cost and precision[13]. An example of approximation due to the cost/precision trade-off is visualized by Figure 3.2. The same set of points is included in an interval, an octagon, and a polyhedron with increasing complexity. The interval is a widened form of the other two; the polyhedron is a narrowed form of the interval and the octagon. Depending on preference or resources, one can choose between the domains with either a loss of precision (choosing the interval) or increasing costs (going with the polyhedron). The following code snippet shows a numerical property which can be formalised with the simplest domain, the interval.

```
int x = Math.random();
if(x <= 5){
    if(x >= 2){
       //some array access at x;
       array[x];
    }
}
```

[5]A reachability graph is a directed graph that shows for every possible state in a program all subsequent states that can be reached by the given transitions.

[6]With $X_i$ for all $x_i$ in the definition set and elements of the interval with $a_i$ as a lower bound and $b_i$ as an upper bound.

[7]Defined by a set of constraints with $X_i$ and $X_j$ for all $x_i$ and $x_j$ in the definition set and $\beta$ as some constant.

[8]Defined by a set of constraints with $X_i$ for all $x_i$ in the definition set and $\alpha$ and $\beta$ as some constant.

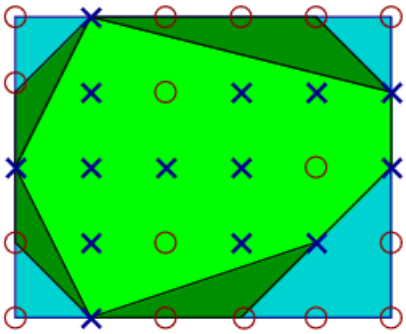


Figure 3.2: Three abstractions of the same set of points.

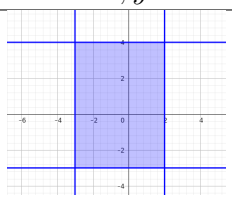
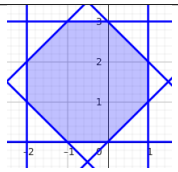
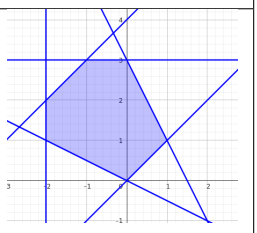
| **Name** | **Form** | **Ex.:** $x, y \in \mathbb{R}$ |
|---|---|---|
| Box | $X_i \in [a_i, b_i]$ |  |
| Octagon | $\pm X_i \pm X_j \leq \beta$ |  |
| Polyhedron | $\sum_i \alpha_i X_i \geq \beta$ |  |

Table 3.1: Overview of numerical abstract domains.

The numerical property over the variable x is $x \leq 5 \wedge x \geq 2$, which is equal to the interval $x \in [2, 5]$.This is a way to guarantee if there is an array access out of bounds or not. In summary, Table 3.3 shows an overview of the three main numerical abstract domains. The third column shows a visualization of the domains for numerical properties, including two variables. In summary, numerical abstract domains come in great variety and differ in how precisely they can represent the numerical properties of a program. Despite solving the question of satisfiability with SMT solvers, the numerical abstract domain focuses on definitely including all possible variable assignments. The point set represented by the domain can include values that might be unsatisfiable. This is also shown in Figure 3.2. The interval includes points that are not actual valid solutions, but the domain is nevertheless sound but imprecise in the sense of abstract interpretation. SMT solvers, on the other hand, can ensure

that if the solver decides for satisfiable for some constraints, this is definitely true. SMT solvers deliver sound and precise results. The implications of this difference will be addressed later in Chapter 5.2. A way to use numerical abstract domains for static analysis in practice will be presented in the next section.

## 3.4 APRON Numerical Abstract Domain Library

The APRON Numerical Abstract Domain Library (Apron)[21] is a library written in C that creates a framework for different numerical abstract domains. It can be used in static analyzers and is based on the theory of abstract interpretation. It is released under the GNU LESSER GENERAL PUBLIC LICENSE Version 2.1[9] The decision for the Apron library over others like ELINA[10] is that ELINA is based on Apron and due to previous work one has already been acquainted with Apron.

Like SMT solvers, the domains diverge in how they are implemented, their API is structured, or the user has to evolve their own domain. Similar to JavaSMT for SMT solvers, the goal of Apron is to offer a unified interface for using different numerical abstract domains in an easy and correct way. Moreover, Apron offers the possibility to extend the library with new domains. At the moment, there are intervals, octagons,convex polyhedrons[11] and zonotopes. Some of the domains are implemented by using the Parma Polyhedra Library, short PPL[2]. This library was originally developed for the handling of convex polyhedrons and was extended over the years. Its actual goal is to provide numerical abstractions, including convex polyhedrons, grids, and finite powersets. A grid is any subset of $\mathbb{R}^n$ whose elements satisfy all the congruences of a congruence system. A congruence system is a finite subset of congruence relations. Grids are part of Apron using the PPL. The framework is based on two levels: level 0 and level 1. Level 0 forms the basis, whereas level 1 is an abstraction of level 0. Level 0 is meant to be used for implementing new domains in the framework. Level 1 is for actually working with and using the domains that already exist. It is more straight-forward to use, and in the implementation, most functions of level 1 are passed to the level 0 implementation. This thesis is not focusing on extending the framework; it focuses on level 1 implementation. Apron is written in C but offers public APIs in Java.

The next part refers to the Java API for Apron, as the whole approach of this

[9]For more details, see `https://github.com/antoinemine/apron/blob/master/COPYING` [13.12.2023].

[10]`https://elina.ethz.ch/#domain` [13.12.2023].

[11]A convex polyhedron is convex if it can be constructed as the convex hull of its vertices. If a polyhedron is convex, the line that connects any two points on the surface of the polyhedron always lies inside the polyhedron.

thesis is written in Java. The first component is the `Manager`, which specifies the type of the domain. It is passed to an object of the `Abstract1` class, which is the class for level 1 numerical abstract values. The seven available domains are sub-classes of `Manager`:

- `Box`: Sub-class for creating the interval abstract domain.
- `Octagon`: Sub-class for creating the octagon abstract domain.
- `Polka`: Sub-class for creating the convex polyhedron domain. The domain can either be strict or not. If the polyhedron is strict, it can be non-closed. For a polyhedron with two axes, this can mean, for example, that one side is not limited by an edge.
- `PolkaEq`: Sub-class for creating linear equalities.
- `PplPoly` Sub-class for creating convex polyhedrons (strict or non-strict) using the Parma Polyhedron Library.
- `PplGrid`: A sub-class for creating linear congruence equalities (grids), defined by the Parma Polyhedra Library[2].
- `PolkaGrid`: Sub-class for the reduced product of polyhedrons and grids. This is the intersection of a polyhedron and a grid. We have some congruence relations that are limited by the range in which they must lie.

Then the properties of the domain can be built by declaring the variables, instances of `Var`, which are held by an environment object of the class `Environment`. Variables can be used to build terms of the form $ax$[12], linear expressions of the form $a_0x_0 + ... + a_nx_n + b$ or linear constraints of the form $a_0x_0 + ... + a_nx_n + b\omega 0$ with $\omega \in \geq, >, =, \neq$ [13]. All constraints have to be of this shape and this in can happen that constraints of other shapes have to be transformed for being mapped to the correct Apron format. The corresponding class names are `Linterm1`, `Linexpr1` and `Lincons1`. For non-linear expressions and constraints, Apron offers expression-tree data types, which are an abstraction of linear expressions. The building of a tree-expression works recursively. Every expression is an instance of the abstract class `Texpr1Node`. A node can be a constant, a variable, or a unary or binary node. Unary nodes are constructed with an unary operator, like negation, and an argument, respectively, from another node. In line with the unary node, binary nodes need an operator and

[12] With $a$ as a numerical constant and $x$ as numerical variable.
[13] With $a_i, b$ as numerical constants and $x_i$ for numerical variables.

two other nodes as arguments. Constant nodes are of the class `Texpr1CstNode` and take the value, namely the coefficient, wrapped by a `Coeff` object. Variables are implemented in the class `Texpr1VarNode` and represent the name of the variable. Objects of `Terxpr1UnNode` symbolize unary nodes and take an operator and an object of `Texpr1Node` as arguments. `Texpr1BinNode` objects also take an operator and two objects of `Texpr1Node` as input. Valid operators are addition, division, modulo, multiplication, power, and subtraction. For constructing constraints, one has to build an object of `Tcons1` with a given node of the type `Texpr1Node` and constraint type, like, for example, equality or strict inequality, plus the environment object.

An abstract value object of the class `Abstract1` represents a set of points in the environment of a certain manager. The abstract value object can be approximated in various ways and checked for satisfiablilty and other features. The `Abstract1`-object also handles the constraints, which are passed to the object as arrays of `Lincons1` or `Tcons1` objects. With all these features, it is possible to construct different numerical domains consisting of constraints over numerical properties. The following code snippet shows an example of how to use tree expressions in Apron.

```
Manager man = new Polka(); //domain-type is polyhedron
Environment env = new Environment();
//Expression-Tree example, 4x+5 > 0
//variable "x"
Texpr1VarNode varNode =
    new Texpr1VarNode("x");
//constant 4; MpqScalar is a subclass of Coeff;
Texpr1CstNode four =
    new Texpr1CstNode(new MpqScalar(4));
//constant 5
Texpr1CstNode five =
    new Texpr1CstNode(new MpqScalar(5));
//4*x
Texpr1BinNode term =
    new Texpr1BinNode(Texpr1BinNode.OP_MUL, four, varNode);
//4x+5
Texpr1BinNode expr =
    new Texpr1BinNode(Texpr1BinNode.OP_ADD,term, five);
//4x+5 > 0
Tcons1 constraint = new Tcons1(env,Tcons1.SUP,expr);
Tcons1[] tcons = new Tcons1[]{constraint};
Abstract1 abstract1 = new Abstract1(man,tcons);
assert abstract1.isBottom(man);
```

### 3.4.1 Apron Binding in JavaSMT

As presented so far, Apron[21] is not an SMT solver but a library that unites different numerical abstract domains based on the theory of abstract interpretation. JavaSMT[3], on the other hand, represents a framework for SMT solvers, grounded in Satisfiability Modulo Theories. In this section, it is argued why it is possible and potentially sensible to add the Apron library to JavaSMT as an SMT solver. As explained in Section 3.1, SMT is a decision problem evaluating if some logical formula $\varphi$ is satisfiable with respect to some background theory $\tau$. In abstract interpretation, the question is whether a set of properties meets a set of specifications. We check that by approximating these properties and evaluating if they are satisfiable or not. So in both cases, the formulas of a background theory are examined with respect to the logic according to the chosen theory. To translate the numerical abstract domain into an SMT solver, it needs to provide the four components each SMT solver needs to have.

1. An underlying logic: Apron will be added to JavaSMT with quantifier-free linear Integer and Rational number arithmetic, short form QF_LIA, QF_LRA[5] as well as some limited version of Boolean.

2. A background theory: according to the logics, the Apron SMT solver will provide Integer and Rational number theory.

3. A set of valid input formulas: the valid input formulas for the theories will be formed by the JavaSMT interface for Boolean, Integer and Real and point to the equivalent classes in Apron.

4. A set of features the solver provides: the solver will provide a `ProverEnvironment` with the obligatory ability for checking satisfiability as well as presenting a model by using features of the Apron `Abstract1` element.

The interesting question will be how the quasi SMT solver via the Apron library will succeed compared to the other solvers. Although it is argued here, that the integration of the Apron library as a quasi SMT solver can be sensible, the developers of Apron confirmed that the implementation can cause problems[14]. How the implementation looks in detail will be explained in the next chapter.

[14] `https://github.com/antoinemine/apron/issues/91` [12.12.2023].

# 4. Implementation

This chapter describes how the Apron library[21] can form an SMT solver in JavaSMT[3]. It will show how the core elements of abstract interpretation are linked to the components of JavaSMT and where the difficulties were encountered[15].

## 4.1 Usage of the Apron Library

Apron offers a Java API. This makes the usage of the Apron code relatively easy. For using the library in JavaSMT, one has to follow the steps[16]for including the Apron library in the IDE in use. Apron is not yet published in an Ivy or Maven repository but will probably be done by JavaSMT in the future[17], respecting the license requirements. The API also includes a Java binding for the GMP[1][18] and MPFR libraries[18][19] as they are used by Apron. The Java binding for Apron is a Beta version where all functions are implemented but not completely tested. During the implementation of this project, a bug on the level 1 implementation was found and reported[20]. The fixing of the bug started due to the report and is ongoing at the time of completion of this thesis. `Texpr1Node` has the method to check whether a variable is part of the formula or not (ex.: Does the formula $a + 4 + b$ contain the numerical variable $a$?). The method does not give correct results unless the `Texpr1Node` is transformed to the lower-level `Texpr0Node` with the method *toTexpr0Node()*. On level 0, the same method gives the correct results [21]. This handling of the bug is also found in the implementation of the classes *ApronModel.getComplexValue()* and *ApronIntegerFormulaManager.modulo()*.

[15]The implementaion files can be found here: `https://github.com/sosy-lab/java-smt/pull/346` [20.12.2023].

[16]`https://github.com/antoinemine/apron` [30.10.2023].

[17]`https://github.com/antoinemine/apron/issues/91` [30.10.2023].

[18]The GMP library is commonly used for computing with arbitrary precision arithmetic and is widely used in many different application areas[1].

[19]MPFR stands for multiple-precision binary floating-point library and is specialized on calculating with floating-points, including correct rounding and arbitrary precision.

[20]`https://github.com/antoinemine/apron/issues/94` [30.10.2023].

[21]More details on the difference between level 1 and level 0 of the Apron implementation can be found in Section 3.4.

## 4.2 Integration of a new Solver

For integrating a new solver into JavaSMT, it is obligatory to implement the following functionalities and classes. `SolverContextFactory` is the main entry point for JavaSMT. The class generates and loads all solvers available by generating the `SolverContext` for each solver. All available solvers can be found in the enum `Solver`, where Apron was also added as *APRON*. The `SolverContext` creates the `FormulaCreator` and the `FormulaManager`. Moreover, the different explicit formula managers for the different theories that the new solver provides are generated. Additionally, it is crucial to implement a `ProverEnvironment` for checking constraints (instances of `BooleanFormula`) and a `Model` to give concrete value suggestions for variables in satisfiable constraints. Functionalities and theories that are obligatory to implement but cannot be supported by the solver throw `UnsupportedOperationExceptions`.

### 4.2.1 Main Apron Components in JavaSMT

This section gives an overview of how the core components of JavaSMT and Apron are linked. A numerical abstract domain with Apron consists of components that are not trivially represented by the obligatory classes for a solver in JavaSMT. This section explains how the Apron components are embedded in the JavaSMT Apron solver. The `Manager` which specifies the type of numerical abstract domain (octagon, polyhedron, etc.) is part of the `ApronSolverContext`. The `Manager` defines the underlying domain and will later be passed to the `Abstract1` object. Therefore, it is part of the `ApronSolverContext` as it is the main and firstly generated component of the solver. The `Environment` for the variables is part of the `ApronFormulaCreator` because this class has the functionality to create new variables with the method *makeVariable()*. New variables have to be added to the `Environment` as well, hence why the `ApronFormulaCreator` holds the `Environment`. The formulas that are built in JavaSMT are linked to the Apron `Texpr1Node` and `Tcons1`. They are wrapped in the `ApronNode` interface (more details in Section 4.3.3. The `Abstract1` object is generated and part of the `ApronTheoremProver`. With the `Abstract1` object, one can check all asserted formulas for satisfiability, and because of the similarity to the functionalities of the `ApronTheoremProver`, it is located there.

## 4.3 Main Components of the Apron Solver

This section gives a detailed description of the implementation of the several classes and interfaces of the Apron solver. It explains how Apron and JavaSMT can be linked in practice and where the difficulties and limitations are.

### 4.3.1 ApronSolverContext

The `ApronSolverContext` is the access point to the Apron solver and generates most of the main class objects, like `ApronFormulaCreator` and `ApronFormulaManager`. The `ApronSolverContext` defines which numerical abstract domain is used. The domains are all sub-classes of `Manager` from the Apron library. The options for domains are `Box` for intervals, `Octagon`, `Polka` for convex polyhedron, `PolkaEq` for linear equalities, `PplPoly` for convex polyhedrons using the Parma Polyhedra Library[2], `PplGrid` for grids using the Parma Polyhedra Library, or `PolkaGrid` for a combination of polyhedron and grid. The default domain is the strict polyhedron, which means that the polyhedron can be non-closed. There is no method to change the domain by the user, this could be extended by future work.

### 4.3.2 ApronFormulaCreator

The `ApronFormulaCreator` determines the type of the formulas, in this case, the `ApronNode` and the type of the formula types. The different types of formula types are defined by the `ApronFormulaType` interface. The interface contains the enum `FormulaType` with three entries, namely BOOLEAN, INTEGER and RATIONAL which are the available formula types of the Apron solver. Each of the types has its own class implementing the interface with the method to return the `FormulaType`. Moreover, the `ApronFormulaCreator` determines the solver-specific environment, which is defined by the `Environment` from the Apron library. This class is an obligatory component of a domain in Apron and consists of two sets of variable names. The first set is for the integer-valued variables, and the second one is for the rational-valued ones. This is limited, so that no variable can have the name of another variable in the set. Additional to the `Environment`-object, a map of the string name and the associated formula (`ApronNode`) is stored in the `ApronFormulaCreator` to have the opportunity to get the formula by name of a used variable or vice versa. The `Environment` does not offer such a method. For adding new variables, the method *makeVariable()* is overwritten in this class. Although Apron provides boolean-formulas, it is only possible to create variables of integer and rational types,

as the Environment keeps track of the available variables and only supports integer and rational variables. The method also updates this and the map.

### 4.3.3 ApronNode

In Apron formulas can be build with the Texpr1Node class. In JavaSMT formulas are all of the Type Formula and both architectures cannot be directly linked. Therefor the Apron solver has ApronNode as an interface for providing formulas of the type Formula that wraps the logic of Apron Texpr1Node. The interface ApronNode inherits from the more general interface Formula and is a wrapper class for linking formula constructions in Apron with the syntax and semantics from JavaSMT[3]. This structure allows type safe constructions of formulas that mirror the Texpr1Node architecture of the Apron library and simultaneously provide all functionalities that JavaSMT demands. ApronNode has two sub-categories: the interface ApronNumeralNode for all numerical (integer and rational) formulas and the class ApronConstraint for all boolean formulas. The ApronNumeralNode interface is implemented by several classes that can be separated into two sections: integer formulas and rational formulas. Every ApronNode has a type, an instance of FormulaType, and an equivalent Apron formula (Texpr1Node) as an attribute. The possible numeral formulas are both existing for integer and rational, mostly differentiated by the FormulaType attribute and have a corresponding instance of Texpr1Node. These are the different numeral formulas:

The simplest one is the constant (ApronRatCstNode and ApronIntCstNode) defined by a BigInteger-valued numerator, respectively, a numerator and a denominator. The associated Apron class, represented in the attribute "cstNode" is the Texpr1CstNode. The next formula is the variable, represented by ApronRatVarNode and ApronIntVarNode. It is defined by the string name of the variable and connected to the Texpr1VarNode. The important thing is that a new variable is added to the Environment object. Therefor the name has to be added to the correct set of integer- or rational-valued variables. The updated Environment-object is then passed to the ApronFormulaCreator. Subsequently, the unary formulas are implemented. An instance of ApronRatUnaryNode or ApronIntUnaryNode is defined by an ApronNode and an unary arithmetic operator. An unary operator is, for example, the negation. The corresponding Apron class is Texpr1UnNode. Similar works the architecture of the ApronRatBinaryNode and the ApronIntBinaryNode as they are defined by two ApronNode objects and an arithmetic operator like OP_ADD for addition. All integer-valued formulas also have a constructor for transforming a rational-valued formula into an integer one.

All boolean formulas are wrapped in the `ApronConstraint` class:

The corresponding class from the Apron library for all boolean formulas in JavaSMT is `Tcons1`. The formulas are defined by a map of an `ApronNode` object and a boolean operator (EQ, DISEQ, SUP, SUPEQ). The reason for the map is that Apron library does not allow conjunctions of constraints in a single object, so stacking constraints is the solution for implementing a boolean *and()* method for the Apron solver. All boolean formulas in the Apron library have the syntactical structure of $<$ `Texpr1Node` $><$ *booleanoperator* $> 0$, so all constraints of a different shape have to be adjusted to fit this structure.

### 4.3.4 ApronFormulaManager

The `ApronFormulaManager` defines the available formula managers, referring to the supported theories. In the case of the Apron solver, these are the `ApronBooleanFormulaManager`, the `ApronIntegerFormulaManager`, and the `ApronRationalFormulaManager`. The `ApronFormulaManager` does not support the methods *parse()*, *dumpFormula()*, and *substitution()* because of lacking equivalences in the Apron library.

### 4.3.5 ApronNumeralFormulaManager

This class is the superclass for all numeral formula-managers and has the only method to check whether an `ApronNode` is numeral or not. For the Apron solver, this is the case for all formulas that are not of the type boolean. The `ApronNumeralFormulaManager` is extended by the `ApronIntegerFormulaManger` and the `ApronRationalFormulaManger`, both described in the next two sections.

### 4.3.6 ApronRationalFormulaManager

The `ApronRationalFormulaManager` implements all needed methods inherited from the `AbstractNumeralFormulaManager` and implements the `RationalFormulaManager` interface. All rational formulas are built by returning a matching instance of `ApronNode`. The *floor()* method uses the constructors of the integer-type nodes to transform a rational-type node into an integer one.

### 4.3.7 ApronIntegerFormulaManager

The `ApronIntegerFormulaManager` implements all needed methods inherited from the `AbstractNumeralFormulaManager` and implements the

`IntegerFormulaManager` interface. Most implementations of the arithmetic methods are straight-forward, as Apron supports them with corresponding operations and classes. A method with no corresponding support from the Apron library is *modularCongruence(IntegerFormula number1, IntegerFormula number2, long n)*. This is implemented with the following equation: $((= x(*n(divxn))))$ with $x = number1 - number2$. Moreover, *modulo()* has the limitation that it is only supported for constant denominators.

### 4.3.8 ApronBooleanFormulaManager

The `ApronBooleanFormulaManager` extends the `AbstractBooleanFormulaManager`. As the Apron library is specialized on numeral properties, it does not really support boolean methods. But the Apron solver can support some of the inherited methods from the parent class by using the functionalities of Apron. The *makeBooleanImpl()* method that returns an `ApronNode` object that is wrapping a true or false by returning an `ApronConstraint` symbolizing the equation $1! = 0$ for true and $1 = 0$ for false. The *and()* method is implemented by stacking the constraints. This is the reason why `ApronConstraint` consists of a map of constraints. The *isTrue()* method involves creating an `Abstract1` element and adding all input constraints. Then the object is tested to see if it is empty (for false) or not. Simultaneously works the architecture of the *isFalse()* method. It is not possible to create boolean variables, as the `Environment` only stores integer- or rational-valued variables. An important limitation here is, that Apron can only guarantee for a true *isFalse()*. This means, that the domain is empty. If it is not empty, this can mean it is satisfiable or unknown! The user is warned about this behaviour in the implementation.

### 4.3.9 ApronUFManager

The `ApronUFMangager` exists because it is obligatory for every solver in JavaSMT, but as Apron does not support uninterpreted functions[22], so it has no functionalities.

### 4.3.10 ApronEvaluator

The `ApronEvaluator` is a class that provides the possibility to evaluate formulas. Here it uses the `Model` of the `ApronTheoremProver` with the *getValue()* method. This method is explained in more detail in Section 4.3.12.

[22] `https://github.com/antoinemine/apron/issues/77` [30.10.2023].

### 4.3.11 ApronTheoremProver

The core of the `ApronTheoremProver` is the `Abstract1` object. It is a numerical abstract value that includes the `Manager`, the `Environment`, and a set of constraints (`Tcons1`). With the help of the `Abstract1`-object, it is possible to test whether the domain is satisfied or not. Similar functionality is provided in JavaSMT via the method *isUnsat()*, which checks whether the `Abstract1` object *isBottom()* or not. *isBottom()* from the Apron library tests if the element is empty, which means that the constraints are unsatisfiable. The method *isUnsatWithAssumptions()* uses the *joinCopy()* method of the Apron library and can support the assumption-solving feature. This method returns a new `Abstract1` object with the set-union of the old and new constraints and can then be checked for satisfiability. Not supported by the Apron solver are unsat-core functionalities and evaluation without checks.

### 4.3.12 ApronModel

One of the main tasks of the `ApronModel` is to evaluate all asserted formulas and to find possible variable assignments if the formulas are satisfiable. The asserted formulas are all constraints that were added to the `ApronTheoremProver`. The core of the model is a list of instances of `ValueAssignment`. This list contains possible assignments for all variables that are part of any added constraint. The list is generated by looping through all existing constraints, and they are checked for variables and the type of the variables. The procedure for getting the assignment for the variable is mostly similar for both types (integer and rational). The next step uses functionality from the Apron library. For each variable in the environment of an `Abstract1` object, it is possible to get an interval, which shows the range of values the variable can take to satisfy the set of constraints. The upper and lower bounds of the interval are determined and checked for infinity. If both bounds go towards infinity, the `ApronModel` decides on zero as a possible assignment. If the lower bound is not infinite, its value is the assignment. Otherwise, the upper bound defines the stored assignment. After the model is generated, one can ask for an evaluation of the formulas. This works with the help of the more complex *getValue()* method. The Apron library does not offer such a method. If the formula in question is a variable, the list of assignments is searched for an entry for the variable, and it returns this value. If the formula is a constraint, the constraint is evaluated with the methods *isTrue()* and *makeBooleanImpl()* from the `ApronBooleanFormulaManager`. If the formula to evaluate is more complex, the model loops through all assignments. If a variable assignment is for a variable that is also part of the formula, the *substitute()*

method of the Apron library is used. This method substitutes all occurrences of the variable with the assignment. For example, the formula is $x + 4$ and the assignment is $x = 3$. The new substituted formula is $3 + 4$. After all possible substitutions are made, the formula is checked for more variables. If it has no more variables, a constraint of the form $result =< substitutedFormula >$ is built and passed to an `Abstract1` object. For the upper example, this would be $result = 3 + 4$ Then the `Abstract1` object is asked for an interval of possible values for *result* and a value inside the interval is returned. If the formula still has variables, the formula is returned.

### 4.3.13 ApronApiTest

The `ApronApiTest` class contains five tests that should help to understand the Apron library and to show unexpected behavior. The first test (*example()*) shows how to build constraints and add them to a domain. The two tests *distinctTest()* and *integerRoundingTest()* show different examples where constraints are not added because of inherent overapproximation of the chosen domains. The problem for an SMT solver is that, as a consequence, sometimes Apron declares sets of constraints as satisfiable, although they are not. This issue is debated in more detail in Chapter 5.2. The *addConstraintsTest()* method shows an example of how Apron simplifies added constraints. If $x = 0$ and $x + x = 0$ are added, Apron stores only the first constraint. The *hasVarTest()* shows a bug that was found and reported to the developers of Apron during this project[23].

## 4.4 Tests

JavaSMT[3] provides several test classes for testing the functionalities and theories of the included solvers. As the Apron library was never intended to work as an SMT solver, several classes and tests are not working for the Apron solver. Tests that should skip a solver call a *require()* method at the beginning of a test method. Apron cannot support tests that require precision, optimization, interpolation, parsing, visitor, unsat-core and uninterpreted functions. Moreover, only Integer and Rational theory are supported. The `BooleanFormulaManager` of Apron cannot support *not()*, *or()*, *XOr()*, or non-numeral-valued variables. The `IntegerFormulaManager` of Apron can only evaluate *modulo()*-formulas if they are linear. Moreover, some

[23] `https://github.com/antoinemine/apron/issues/94#issuecomment-1742038224` [30.10.2023].

tests in `ModelTest` are switched off for the Apron solver because the evaluation gives wrong results due to overapproximation of the existing properties.

# 5. Evaluation

This chapter discusses the difficulties of using the APRON Numeric Abstract Domain Library[21] as a quasi SMT solver. It first shows why an automatic evaluation with empirical data via CPAchecker is not yet possible. Thereafter, it is talked about which weak points need to be addressed for meaningful integration into JavaSMT[3].

## 5.1 Evaluation with CPAchecker

This section will present how an evaluation of the Apron solver with the help of CPAchecker[8] could look in theory and where the difficulties are.

The goal of this evaluation is to show how competitive the Apron solver is compared to other solvers. CPAchecker is a tool for automatic program verification of C programs based on Configurable Software Verification (CPA). The JavaSMT framework[3] is used in CPAchecker as a back-end tool for different analyses. Therefore, all solvers in JavaSMT can be used in CPAchecker. A user can verify a program against certain algorithms, for example, Bounded Model Checking or Symbolic Execution. If the algorithm chosen is SMT based, CPAchecker will encode the properties of the program as SMT formulas. These formulas are subsequently checked for feasibility with the chosen SMT solvers available in JavaSMT. Therefore, CPAchecker can be used for evaluating the Apron solver against other existing solvers in JavaSMT. For comparing solvers, one has to create a set of programs that can be used as test cases and use the solvers to verify them. The chosen algorithm, like, for example, Symbolic Execution, is used for the same programs, but the selected solver is changing. For gaining sensible, comparable results, one can only use solvers that provide the same theories. In this case, Integer and Rational. CPAchecker then shows the results of comparable values, like run-time or error rate, for the solvers that were used.

The first step to using the Apron solver in CPAchecker is to locally publish it with Ant on JavaSMT and then build CPAchecker with the renewed JavaSMT. For locally publishing the new solver, it was necessary to reconstruct the steps of building a usable Apron library and add them to the `.xml` files that are used for

building JavaSMT with Apache Ivy[24]. It could be shown that the implementation of the new quasi solver is running in CPAchecker. Contrary to initial assumptions, it is not possible to use a solver in CPAchecker that does not support uninterpreted functions. As Apron does not support them or offers trivial solutions for adding an `UFManager`[25], it is not possible to evaluate the quasi solver and see how it performs in comparison to others. This gives reason to consider the Apron solver as not enriching to the JavaSMT framework. More reasons for this assessment are given in Section 5.2.

## 5.2 Limitations of the Apron Solver

As the Apron library is specialized in numerical abstract domains, there are several issues that prevent the Apron solver from working as a full-fledged SMT solver. Our initial hypothesis from Section 3.4.1 stated that Apron can be translated to a sensible SMT solver. However, non-negligible problems were discovered during the implementation. Additionally, the main developer of the library also assured that the integration of Apron in an SMT solver framework can cause problems[26].

One problem is that, because of overapproximation, there is a loss of precision. The overapproximation causes that some constraints that are added to the theorem prover are not added to the `Abstract1` object. As a result, unsatisfiable results may be returned as satisfiable results, with an assignment that is incorrect in regards to SMT. This is often the case for inequalities with ! =, especially for convex polyhedrons[27], but many different examples can be found. The following code snipped shows such a case:

```
1    Solvers solver = Solvers.APRON
2    SolverContext context = [...]
3    RationalFormulaManager rfm = [...]
4    BooleanFormulaManager bfm = [...]
5    RationalFormula x = rfm.makeVariable("x");
6    RationalFormula y = rfm.makeVariable("y");
7    RationalFormula zero = rfm.makeNumber("0");
8    BooleanFormula xIsZero = rfm.equal(x, zero);
9    BooleanFormula yIsZero = rfm.equal(y, zero);
10   BooleanFormula xIsNoty = rfm.distinct(x, y);
11   ProverEnvironment prover = [...]
```

[24] Apache Ivy is a build-management tool for loading libraries in a Java project. For more details: `https://ant.apache.org/ivy/` [06.12.2023].

[25] `https://github.com/antoinemine/apron/issues/77#event-9932486346` [06.12.2023].

[26] `https://github.com/antoinemine/apron/issues/91` [30.10.2023].

[27] `https://github.com/antoinemine/apron/issues/92#event-10527322585` [30.10.2023].

```
12      //x, y = 0 and x != y are pushed to the prover
13      prover.push(xIsZero);
14      prover.push(yIsZero);
15      prover.push(xIsNoty);
16      //This assertion will fail
17      assertTrue.(prover.isUnsat());
```

The code snipped shows a case where three constraints are added to the theorem prover that should be unsatisfiable but are reported to be satisfiable by Apron. The reason is that the `ApronTheoremProver` passes all constraints that are added to an `Abstract1` object from the Apron library (more details in Section 4.3.11) and Apron approximates these constraints to a more imprecise set. This behavior was observed and tested with all available domains. Moreover, the default domain that was tested most was already the most precise one. As opposed to an SMT solver, abstract interpretation can be imprecise but sound in the sense of abstract interpretation, and therefore some constraints are ignored and, respectively, overapproximated. This is what also happens in the upper example. The last constraint is not added to the `Abstract1` object. Consequently, the `ApronTheoremProver` is not unsatisfiable. Currently, there is no option to detect possible approximations. SMT solvers are expected to give precise results about the satisfiability of constraints, so the user expects such behavior and can be misguided by the results of the Apron solver. More examples of constraints that are overapproximated by Apron can be found in the `ApronApiTest` class[28].

A second problem for the applicability of the Apron solver is the model. The rough idea behind finding a suitable variable assignment for the `ApronModel` is the following: The solver uses the possibility of the Apron library to ask for an interval of values for a specific variable. The `Abstract1` object returns an interval of values for which all available constraints are satisfied. The `ApronModel` then returns one value in the interval as a possible result. This works, for example, for these properties: $x = 3, y = 1/2$. If the model is asked for an evaluation of $x + y$, it gives $7/2$ as a possible solution. But there are also cases where this fails. If the user asks for a model of the constraint $x = y+3$, the solver returns $]-\infty, +\infty[$ as a possible solution for $x$ and $y$. The reason is that Apron cannot set the two intervals in relation to each other. The way this is executed is by searching for a solution so that $x \in [a, b]$ for $a$ minimal and $\exists y.x = y + 3$ and $b$ is maximal. The `ApronModel` interprets this in the way that both variables can take any value and decide for zero in both cases.

[28] `https://github.com/sosy-lab/java-smt/blob/314-adding-the-abstract-numeric-domains-of-the-apron-library-as-smt-solver/src/org/sosy_lab/java_smt/solvers/apron/ApronApiTest.java` [30.10.2023].

So the solutions for evaluations of formulas given by the `ApronModel` are sometimes wrong. Implementing a method that calculates such assignments on its own is by far not trivial, and in the worst case, it can need its own solver.

Another problem is that Apron can only guarantee that *isUnsat()* in `ApronTheoremProver` gives a clear result. The opposite does not necessarily mean that a domain is satisfiable. The *isUnsat()* method asks the related `Abstract1` object for *isBottom()*. If the return value is true, this means that the set of abstract values is definitely empty. Consequently, there is no possible value that satisfies all constraints. If *isBottom()* returns false, this means that it is only maybe not empty. So it is not possible to detect satisfiable constraints as *!isBottom()* does not imply SAT but UNKNOWN or SAT. While no case of UNKNOWN could be found during the development and testing of the Apron solver extension, it is still a valid threat that at some point a wrong result could be encountered. This also affects the *isTrue()* method from the `ApronBooleanFormulaManager`. In cases where the *isBottom()* method from Apron is used, the user is informed that *!isBottom()* does not necessarily mean that the constraints are satisfiable.

Furthermore, it should be mentioned that the `ApronBooleanFormulaManger` has very limited functionalities. There are no boolean-valued variables, and the only way to build up boolean formulas is by using the *and()* method. The consequences are that only a subset of the extensive test suite present in JavaSMT could be executed for the Apron solver. All tests that use the *or()*, *not()*, or *xor()* methods were disabled for Apron. Consequently, the users are not able to use the methods. Moreover, these functionalities for Boolean theory are part of the core idea of an SMT solver. SMT solvers are an abstraction of SAT solvers, which are made for evaluating boolean formulas. Therefore, it should be expected for each solver in JavaSMT to be able to handle and create diverse boolean formulas.

Lastly, it should be mentioned that another core component of SMT solvers cannot be supported by the Apron solver. The missing component is the concept of uninterpreted functions (UFs). They don't exist in the Apron library, and there is no straight-forward solution for implementing them in JavaSMT. One of the resulting consequences is discussed in Section 5.1.

# 6. Conclusion

This thesis presented the parallels between SMT and numerical abstract domains. It explained the background of the two concepts and showed how abstract interpretation with numerical abstract domains can be translated into an SMT solver. Moreover, it was argued that the architecture of JavaSMT[3] and the Apron library[21] is similar. They both offer a framework to use different solvers, respectively domains, in the same way and therefore make the switch between the different units easy. We presented how the Apron library may be translated into a quasi SMT solver, based on an example implementation in JavaSMT. The implementation brought different problems to light, which finally showed that the quasi SMT solver cannot fulfill important features of an SMT solver. The overapproximation of the value set causes that sometimes constraints are not added. None of the available numerical abstract domains in Apron were able to provide precise results with respect to unsatisfiable formulas. Contrary to that, SMT solvers do not overapproximate and are able to decide on clear results concerning satisfiability. The imprecise depiction of numerical properties, which is intended in abstract interpretation, for example, for reducing costs, does not go in line with the core concepts of SMT solving. Additionally, it has to be mentioned that two core characteristics of SMT solvers could not be realized for the Apron solver. The first one is the limited possibilities for boolean operations. No other propositional logic operators besides conjunctions could be implemented. The second missing property is the concept of uninterpreted functions, which made an automated evaluation with the help of CPAchecker[8] impossible.

Based on the outlined findings, we argue that the numerical abstract domain cannot work as a viable SMT solver without further work on the mentioned problems.

# Bibliography


[1] Gianluca Amato and Francesca Scozzari. "JGMP: Java Bindings and Wrappers for the GMP Library". In: *SoftwareX* 23 (2023). URL: https://www.sciencedirect.com/science/article/pii/S2352711023001243.

[2] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Science of Computer Programming* 72.1-2 (2008), pp. 3–21. URL: https://www.sciencedirect.com/science/article/pii/S0167642308000415.

[3] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. "JavaSMT 3: Interacting with SMT Solvers in Java". In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 195–208. URL: https://doi.org/10.1007/978-3-030-81688-9_9.

[4] Haniel Barbosa et al. *CVC5 at the SMT Competition 2022*. 2022. URL: https://smt-comp.github.io/2022/system-descriptions/cvc5.pdf.

[5] Clark Barrett, Pascal Fontaine, and Tinelli. "The SMT-LIB Standard: Version 2.6". In: 2021. URL: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[6] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*. Springer, 2018. URL: https://doi.org/10.1007/978-3-319-10575-8_11.

[7] Clark Barrett et al. "CVC4". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer. 2011, pp. 171–177. URL: https://doi.org/10.1007/978-3-642-22110-1_14.

[8] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer. 2011, pp. 184–190. URL: https://link.springer.com/chapter/10.1007/978-3-642-22110-1_16.

[9] Franck Cassez and Anthony M Sloane. "ScalaSMT: Satisfiability Modulo Theory in Scala (Tool Paper)". In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. 2017, pp. 51–55. URL: https://doi.org/10.1145/3136000.3136004.

[10] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An Interpolating SMT Solver". In: *International SPIN Workshop on Model Checking of Software*. Springer. 2012, pp. 248–254. URL: https://doi.org/10.1007/978-3-642-31759-0_19.

[11] Alessandro Cimatti et al. "The MathSAT5 SMT Solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 93–107. URL: https://doi.org/10.1007/978-3-642-36742-7_7.

[12] David R Cok. "jSMTLIB: Tutorial, Validation and Adapter Tools for SMT-LIBv2". In: *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer. 2011, pp. 480–486. URL: https://doi.org/10.1007/978-3-642-20398-5_36.

[13] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: *Journal of Logic and Computation* 2.4 (1992), pp. 511–547. URL: https://doi.org/10.1093/logcom/2.4.511.

[14] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 1977, pp. 238–252. URL: https://doi.org/10.1145/512950.512973.

[15] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 1978, pp. 84–96. URL: https://doi.org/10.1145/512760.512770.

[16] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340. URL: https://doi.org/10.1007/978-3-540-78800-3_24.

[17] Bruno Dutertre. "Yices 2.2". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 737–744. URL: https://doi.org/10.1007/978-3-319-08867-9_49.

[18] Laurent Fousse et al. “MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding”. In: *ACM Transactions on Mathematical Software (TOMS)* 33.2 (2007). URL: https://dl.acm.org/doi/abs/10.1145/1236463.1236468.

[19] Marco Gario and Andrea Micheli. “PySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms”. In: *SMT Workshop*. Vol. 2015. 2015. URL: https://www.mikand.net/papers/pysmt.pdf.

[20] Antti EJ Hyvärinen et al. “OpenSMT2: An SMT Solver for Multi-Core and Cloud Computing”. In: *Theory and Applications of Satisfiability Testing–SAT 2016: 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings 19*. Springer. 2016, pp. 547–553. URL: https://doi.org/10.1007/978-3-319-40970-2_35.

[21] Bertrand Jeannet and Antoine Miné. “APRON: A Library of Numerical Abstract Domains for Static Analysis”. In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 661–667. URL: https://doi.org/10.1007/978-3-642-02658-4_52.

[22] Jiahong Jiang et al. “Block-Wise Abstract Interpretation by Combining Abstract Domains with SMT”. In: *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings 18*. Springer. 2017, pp. 310–329. URL: https://doi.org/10.1007/978-3-319-52234-0_17.

[23] Antoine Miné. “The Octagon Abstract Domain”. In: *Higher-Order and Symbolic Computation* 19 (2006), pp. 31–100. URL: https://doi.org/10.1007/s10990-006-8609-1.

[24] Aina Niemetz et al. “BTOR2, BtorMC and Boolector 3.0”. In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 587–595. URL: https://doi.org/10.1007/978-3-319-96145-3_32.

[25] Jan Peleska, Elena Vorobev, and Florian Lapschies. “Automated Test Case Generation with SMT-Solving and Abstract Interpretation”. In: *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer. 2011, pp. 298–312. URL: https://doi.org/10.1007/978-3-642-20398-5_22.

[26] Philipp Rümmer. “A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2008, pp. 274–289. URL: https://doi.org/10.1007/978-3-540-89439-1_20.

[27] Roberto Sebastiani and Patrick Trentin. “OptiMathSAT: A Tool for Optimization Modulo Theories”. In: *Journal of Automated Reasoning* 64.3 (2020), pp. 423–460. URL: https://doi.org/10.1007/978-3-319-21690-4_27.