JKind User Guide

Table of Contents

[Table of Contents 2](#_Toc505170281)

[1 Introduction 4](#_Toc505170282)

[2 Tool Installation 5](#_Toc505170283)

[3 LUSTRE Language 7](#_Toc505170284)

[3.1 Dataflow Language 7](#_Toc505170285)

[3.2 Syntax Overview 8](#_Toc505170286)

[3.3 Lexical Elements 9](#_Toc505170287)

[3.4 Declarations 10](#_Toc505170288)

[3.4.1 Constant 10](#_Toc505170289)

[3.4.2 Type Definition 11](#_Toc505170290)

[3.4.3 Node 11](#_Toc505170291)

[3.4.4 Function 11](#_Toc505170292)

[3.5 Types 12](#_Toc505170293)

[3.5.1 Enumeration Types 13](#_Toc505170294)

[3.5.2 Array Types 13](#_Toc505170295)

[3.5.3 Record Types 13](#_Toc505170296)

[3.6 Expressions 13](#_Toc505170297)

[3.6.1 ID Expressions 14](#_Toc505170298)

[3.6.2 Literal Expressions 14](#_Toc505170299)

[3.6.3 Cast Expressions 14](#_Toc505170300)

[3.6.4 Call Expressions 15](#_Toc505170301)

[3.6.5 Condact 15](#_Toc505170302)

[3.6.6 Record Expressions, Field Accesses, and Updates 15](#_Toc505170303)

[3.6.7 Array Expressions, Accesses and Updates 15](#_Toc505170304)

[3.6.8 Stream (Previous Value and Arrow) Expressions 16](#_Toc505170305)

[3.6.9 Logical Operators 17](#_Toc505170306)

[3.6.10 Arithmetic Operators 17](#_Toc505170307)

[3.6.11 Relational Operators 17](#_Toc505170308)

[3.6.12 If Then Else Expression 18](#_Toc505170309)

[3.6.13 Tuple Expressions 18](#_Toc505170310)

[4 Command Line Flags 19](#_Toc505170311)

[4.1 Configuration Flags 19](#_Toc505170312)

[4.2 Output Flags 20](#_Toc505170313)

[4.3 Miscellaneous Flags 20](#_Toc505170314)

# Introduction

JKind [1] is an open-source industrial infinite-state inductive model checker for safety properties. Models and properties in JKind are specified in Lustre, a synchronous data-flow language, using the theories of linear real and integer arithmetic. JKind uses SMT-solvers to prove and falsify multiple properties in parallel. A distinguishing characteristic of JKind is its focus on the usability of results. For a proven property, JKind provides traceability between the property and individual model elements. For a falsified property, JKind provides options for simplifying the counterexample in order to highlight the root cause of the failure. In industrial applications, we have found these additional usability aspects to be at least as important as the primary results. Another important characteristic of JKind is that is it designed to be integrated directly into user-facing applications. Written in Java, JKind runs on all major platforms and is easily compiled into other Java applications. JKind bundles the Java-based SMTInterpol solver and has no external dependencies. However, it can optionally call Z3, Yices 1, Yices 2, CVC4, and MathSAT SMT solvers if they are available.

JKind’s source [2] is open and hosted on Github.

# Tool Installation

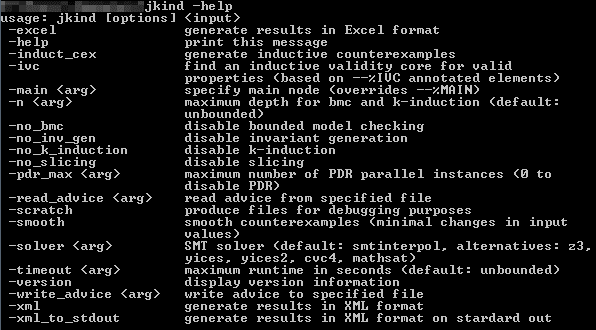
JKind is implemented in Java. This allows it to be installed on multiple platforms with minimal effort. The prerequisite for installing and using JKind are:

* Having Java installed and on the PATH variable for the OS

Installing JKind is accomplished by the following steps:

1. Download the zip containing the latest release from <https://github.com/agacek/jkind/releases>. As of the writing of this document, that is version 4.0
2. Unzip the contents of that zip to the directory of your choice.
3. Add the folder from step #2 to the PATH variable for your OS.

The installation can be tested by executing ***jkind –help*** from the command line. Successful installation will provide the following result:



JKind comes packaged with the SMTInterpol solver, and it uses it by default. The tool is also able to use Z3 [3], Yices1 and Yices2 [4], CVC4 [5], and MathSAT [6] solvers. It looks for specific environment variables that point to the installations for its supported solvers.

To add a solver, users must perform these steps:

1. Download the solver of choice.
2. Unzip the contents of the solver into a directory.
3. Create an environment variable called <SOLVER>\_HOME that points to the directory used the previous step. This is:
   * CVC4\_HOME for CVC4
   * Z3\_HOME for Z3
   * MATHSAT\_HOME for MathSAT
   * YICES\_HOME for Yices1
   * YICES2\_HOME for Yices2
4. JKind will automatically look for the “bin” directory under <SOLVER>\_HOME when the solver is activated.

# LUSTRE Language

## Dataflow Language

JKind accepts input from the *synchronous dataflow language* Lustre. Lustre is a *synchronous* *dataflow* language used for programming real-time systems. A *dataflow* language consists of a set of *equations* that assign *variables* in which a variable can be computed as soon as its *data dependencies* have been computed. As an example, consider a system that computes the values of two variables, X and Y, based on four inputs: a, b, c, and d:



Figure 11: A dataflow model and its associated set of equations

This diagram is to be read left-to-right, with the inputs "flowing" through the system of operators to create the outputs at the right side. The diagram can be represented more concisely as a set of equations, as shown at right. We name the inputs to the dataflow model *input variables* and all variables that are computed by the model *state variables.*

As the basis of a high-level programming language, the dataflow model has several merits:

* It is a completely functional model without side effects. This feature makes the model well-suited to formal verification and program transformation. It also facilitates reuse, as a module will behave the same way in any context into which it is embedded.
* It is a naturally parallel model, in which the only constraints on parallelism are enforced by the data-dependencies between variables. This allows for parallel implementations to be realized, either in software, or directly in hardware.

Dataflow models can be either *synchronous* or *asynchronous*. In an asynchronous dataflow model, the outputs of the system are continually recomputed depending on the inputs to the system. In the synchronous model, however, real-time is broken into a sequence of instants in which the model is recomputed. The synchronous model is better suited to translation into a programming language, as it more naturally matches the behavior of a computer program. Therefore, all of the dataflow-style languages adopt some form of this approach.

The variables in a dataflow model are used to label a particular computation graph; they are not used as constraints. Therefore, it is incorrect to view the equations as a set of constraints on the model: a set of equations such as {X = 2a/Y, Y = X + d} does not correspond to an operator network because X and Y mutually refer to one another. Put another way, there is no way to arrange the variables from left to right such that each can be computed. This is shown in Figure [12](#fig:dataflow_model_cyclic), where the bold red-lines indicate the cyclic dependencies. Such a system may have no solution or infinitely many solutions, so cannot be directly used as a deterministic program. If viewed as a graph, these sets of equations have *data dependency cycles*, and are considered incorrect.



Figure 12: A Dataflow Model with Cyclic Dependencies

However, in order for the language to be useful, we must be able to have mutual reference between variables. To allow benign cyclic dependencies, a *delay operator* (prev) is added. The operator returns the value of an expression, delayed one instant. For example: {X = 2a + Y; Y = (prev(X, 1)) + d} defines a system where X is equal to 2a plus the current value of Y, while Y is equal to the *previous* value of X (with value in the initial instant set to 1) plus the current value of d. Systems of equations of this form always have a single solution. The delay operator is also the mechanism for recording state about the model. For example, we can construct a counter over the natural numbers by simply defining the equation: x = prev(x+1, 1).

Finally, some notion of selection is added to assignment expressions. In Lustre, this is simply an if/then/else statement. From these elements, at its core, a dataflow program can be viewed as simply a set of input variables and assignment equations of the form {X0 = E0, X1 = E1, ..., Xn = En} that must be acyclic in terms of data dependencies.

## Syntax Overview

Before describing the details of the language, we provide a few general notes about the syntax. In the syntax notations used below, syntactic categories are indicated by monospace font. Grammar productions enclosed in parenthesis ('( )') indicate a set of choices in which a vertical bar ('|') is used to separate alternatives in the syntax rules or ' ..' is used to describe a range (e.g. ('A'..'Z')). Any characters in single quotes describe concrete syntax: (e.g.: '+', '-', '=>', '''). Note that the last example is the concrete syntax for a single quote. Examples of grammar fragments are also written in the monospace font. Sometimes one of the following characters is used at the beginning of a rule as a shorthand for choosing among several alternatives: 1) The '\*' character indicates repetition (zero or more occurrences) and '+' indicates required repetition (1 or more occurrences). 2) A '?' character indicates that the preceding token is optional.

JKind’s Lustre grammar is built in ANTLR and can be referenced at <https://github.com/agacek/jkind/blob/master/jkind-common/src/jkind/lustre/parsing/Lustre.g4>

The rest of this document will post relevant snippets from this grammar, explaining

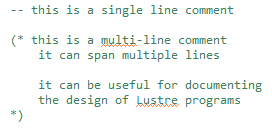
## Lexical Elements

JKind supports single line and multi-line comment styles. The grammar elements for comments are shown in Figure 1.



Figure - Comment grammar elements

Single line comments begin with two adjacent hyphens and span to the end of a line. Here is an example. Multi-line comments begin with “(\*” and end with “\*)” and may span multiple lines. Both are shown below:



The grammar for identifiers in JKind are shown in Figure 2.



Figure - JKind identifier grammar

This means that valid identifers will begin with a upper or lowercase letter, followed by 0 or more letters, numbers, or underscore characters

**Note:** While valid grammatically, users should not use the ‘~’ character in identifiers. These characters are reserved for the tool’s internal processes.

The grammar for JKind’s literals are shown in Figure 3.

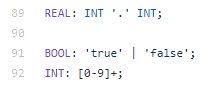
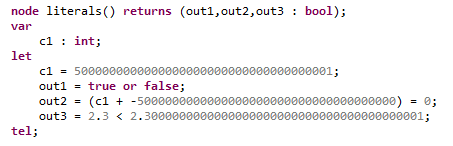


Figure - JKind literal grammar

Shown in the following example are some Lustre literals in use.



As noted in Section 3.4, JKind uses infinite precision datatypes for integers and reals. It is able to accurately represent the constant **c1** in the above example. The tool is also able to determine that **out2** and **out3** evaluate to true.

## Declarations

In JKind, a Lustre program is a collection of constant, type, node, and function declarations. The grammar elements for these declarations are shown in Figure 4.

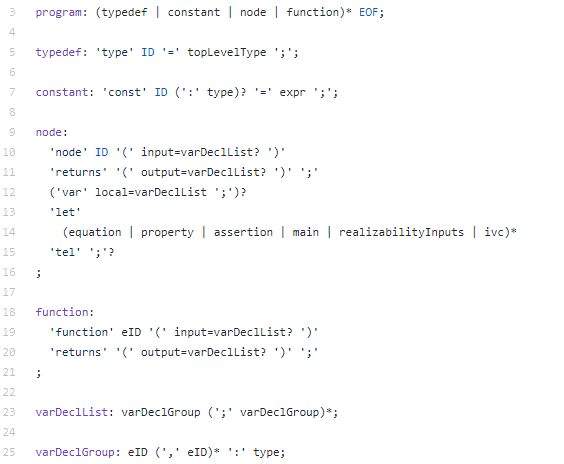
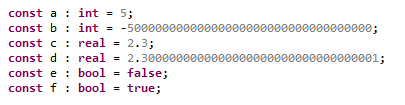


Figure - Lustre program declarations

The entry point of JKind’s analysis is a node. If there are more than one node, JKind will consider the entry point to be the node annotated with the --%MAIN declaration, or will use the last node in the file as the main node. As a result, to do any meaningful analysis, a Lustre program must have at least one node defined. The remaining elements (constants, type definitions, and functions) can be used to specify a Lustre program for analysis. Each element is described in the following sections.

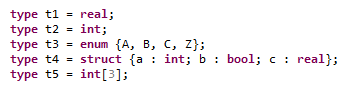
### Constant

A constant declaration is a useful way to define values that are commonly referred to throughout a Lustre program.



### Type Definition

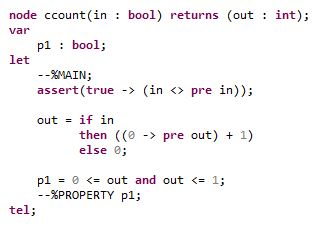
A type definition is a way to alias primitive or existing types. A type definition is also the only way to use record and enumeration types. The following type definitions show several type definitions for both aliasing and defining records and enumerations.



### Node

A node definition is the primary way of programming specific behaviors. A node accepts inputs (in the form of variable declarations, shown on line 25) and assigns local and output variables through assignment equations.

The following example defines a node that takes a single boolean input and counts how many consecutive steps it has been true.



This node has been marked with the main node annotation (--%MAIN) to tell JKind it is the entry point for its analysis. Next, an assertion is declared that ensures that the input will never be equal to its previous value. Next, two equations are defined. The first assigns the out variable. The second defines a local variable p1, which captures an invariant property. Finally, the --%PROPERTY annotation is provided to instruct JKind to verify that p1 is always true.

Analyzing this node in JKind shows that property p1 is valid. It is valid because the assertion ensures that the input will never be true for more than a single step. This causes the counter to always reset to zero every other step, ensuring the output will always be less than or equal to 1.

### Function

JKind also provides support for uninterpreted functions in Lustre. Uninterpreted functions can be used to represent an unspecified computation that behaves as a true mathematical function. That is, given the same input(s) the function provides the same output(s). The following example shows three function declarations that JKind supports.



Note: In the above image, fn2 has no arguments and returns a single output of type integer. This can be used to represent an unspecified constant in JKind Lustre files. These can be used to model values that are known to be constant but whose explicit value is unknown.

## Types

The grammar elements for the Lustre types that JKind supports are shown below.

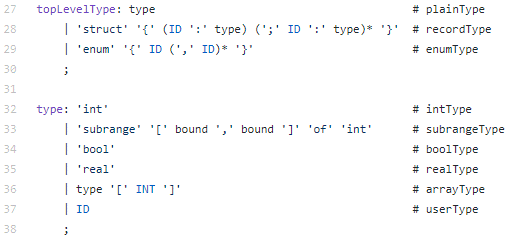


Figure - JKind's Lustre type grammar

JKind supports primitive types of int (Shown on line 32 of Figure 5), boolean (line 34), real (line 35), subrange of integer (line 33) and enumeration (defined on line 29, referenced on line 37).

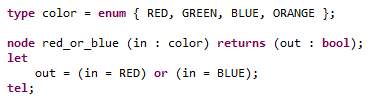
JKind supports composite types of records (defined on line 28, referenced on line 37) and array (line 36).

JKind also supports type aliasing. A type alias must be defined (line 27) and referenced (line 37) similar to enumeration and record types.

**Note**: Lustre uses unbounded, infinite range integers and rational numbers. Similarly, all floating point numbers are approximated by rational numbers. **This means that** **JKind results are not guaranteed to be sound with respect to system implementations** **that use machine data types, such as signed and unsigned 32-bit integers or floating point representations.** We expect that future versions of JKind will support bit-level integers, as these are widely supported by solvers. On the other hand, floating point solvers are currently immature, so it is likely that reals will be used for the foreseeable future. If exact floating point behavior (including rounding and truncation) are important to your verification problem, JKind may provide incorrect answers.

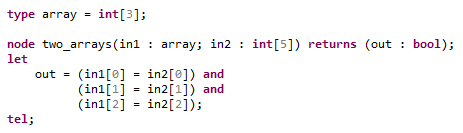
### Enumeration Types

Enumeration types must be specified in type definitions, which can then be referenced when typing variables. The following example shows how to define an enumeration type and how to reference the declared type, in bold.



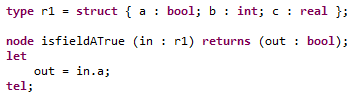
### Array Types

Array types may be defined using type aliasing and referenced, similar to enumeration types. Array types may also be defined inline when typing a variable. The following example shows how to do the former on variable in1 and the latter on variable in2.



### Record Types

Similar to enumeration types, record types must be specified in type definitions, which can be referenced when typing variables. The following example shows how to declare a record structure and type a variable using it.



## Expressions

The Lustre expressions elements supported by JKind are shown in the grammar snippet in Figure 6.

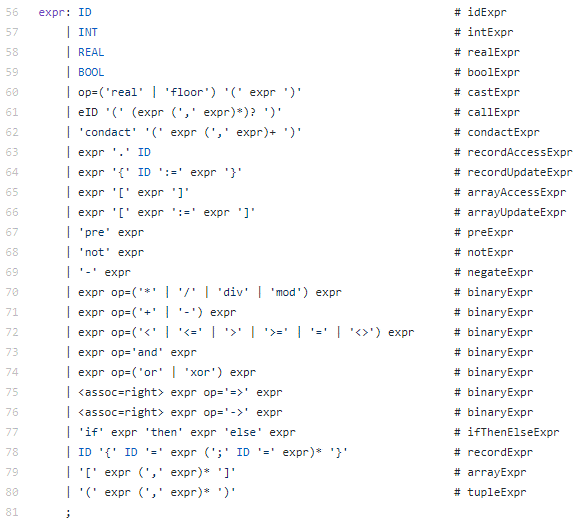


Figure - JKind's Expression grammar

### ID Expressions

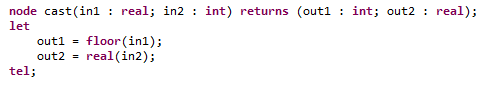
ID expressions are used to reference variables, enumeration values, and record field elements by name.

### Literal Expressions

Literal expressions are used to express boolean literals of true/false, integer literals 0 to Infinity, and real literals 0.0 to Infinity. Negative literals are obtained by using the unary negation operator, shown in line 69 of Figure 6.

### Cast Expressions

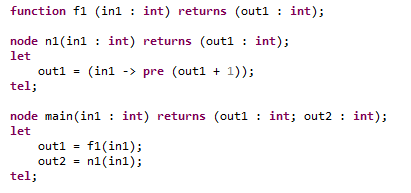
JKind’s Lustre grammar supports the use of casting expression. Cast expressions can be used to convert between integer datatypes and real datatypes, and vice-versa. An example of using the cast operators follows.



**Note**: Real to integer casting uses the floor rounding method (i.e. 1.54 becomes 1, -2.54 rounds to -3), hence the name “floor”.

### Call Expressions

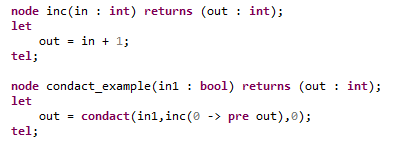
Lustre programs can contain both nodes and functions, which can be called by equations. Call expressions require the user to identify the ID of the function being called, and the input arguments that are being passed to the node or function. The following example shows how two calls are made in the node named main. The first call is to the function f1, with in1 as the argument. The second call is made to node n1, with variable in1 passed as the argument.



### Condact

Condact expressions (a portmanteau of conditional activation) are used to clock node call expressions. A clocked node computes local and output streams when the clock expression is true, and holds (they do not change) the previous value of local and output streams when false.

The following example illustrates the use of a condact expression to clock a simple node that increments an integer variable, named ***inc***.



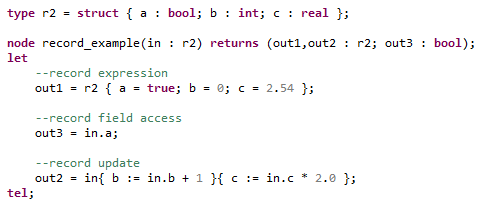
In the node ***condact\_example***, the condact expression is used to clock a call to node ***inc***. The first argument to the condact, ***in1***, is the expression that clocks the node. The second argument is the node call expression that is being clocked. The remaining expressions are the initial values of the called node’s outputs if the clocking signal is false on the initial step. Since the ***inc*** node has a single integer output, a single argument, 0, is provided.

The following shows a single trace for the input and outputs of the condact\_example node to illustrate the semantics of the condact expression.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Time Instant** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **in1** | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| **Out** | 1 | 1 | 2 | 3 | 3 | 3 | 3 | 4 | 5 | 5 |
|  |  |  |  |  |  |  |  |  |  |  |

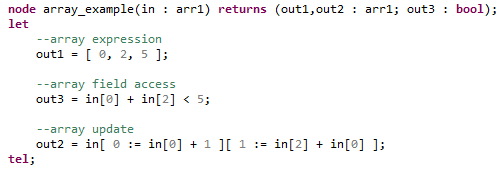
### Record Expressions, Field Accesses, and Updates

Record expressions are used to create a new instance of a record variable. Record field access expressions are used to access field elements of a record. Record update expression are used to perform incremental updates to an existing record. The following example demonstrates the use of record expressions, record access expressions, and record update expressions.



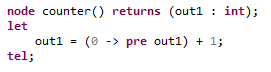
### Array Expressions, Accesses and Updates

Array expressions are used to create a new instance of an array variable. Array field access expressions are used to access elements of an array. Array update expressions are used to perform incremental updates to an existing array. The following example demonstrates the use of record expressions, record access expressions, and record update expressions.



### Stream (Previous Value and Arrow) Expressions

**Arrow Expression**. The arrow expression evaluates to the value of the expression of the left hand side of the arrow on the initial step. Otherwise, it evaluates to the value of the expression on the right hand side of the arrow. The arrow expression is used with the pre expression to specify the value of previous variables on the initial step. In the following example, variable out1 is 0+1 = 1 on the step (uses the left hand side of the arrow expression inside the parenthesis) and is pre out1 + 1 (uses the right hand side of the arrow expression inside the parenthesis) on subsequent steps.



**Previous Value Expression**. A previous value expression evaluates to the value of its argument on the previous time step. It is good practice to combine pre expressions with arrow expressions to ensure that the overall expression is defined on the initial step.

In the above example, in the initial instant out1 is equal to 0. In all subsequent instants, out1 is equal to the previous value of out1 + 1. If we examine the evolution of out1 over a time window of ten steps, it is relatively straightforward to see.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Time Instant** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **out1** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

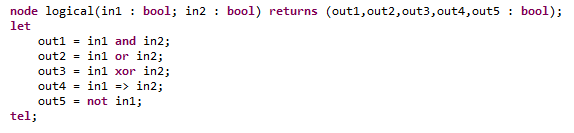
The arrow (->) operator is the stream initialization operator. Given an expression x -> y, in the initial instant in time, the value is equal to x. In all subsequent instants, it is equal to y. So, suppose we have:



Then, in the first instant in time, out2 will be assigned "false" and in every other instant in time, it will be assigned to the value of in1.

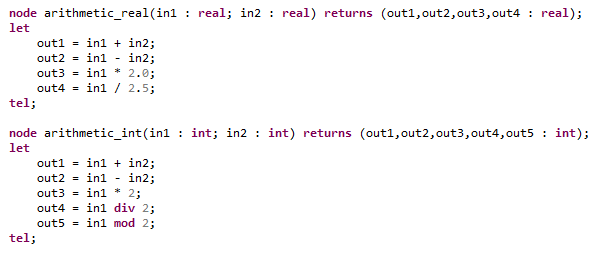
### Logical Operators

JKind’s Lustre supports basic logical operators of and, or, xor, => (logical implication), and not (logical negation). The following example shows how each can be used.



### Arithmetic Operators

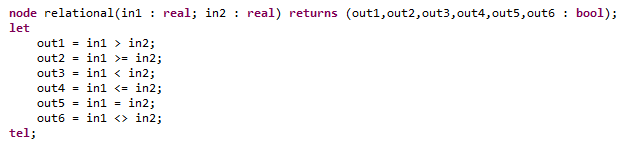
JKind supports arithmetic operators of + (addition), - (subtraction), \* (multiplication), / (real division), div (integer division), and mod (integer modulus). The following example shows two nodes that show the allowed usage of arithmetic operators on both real and integer typed variables.



### Relational Operators

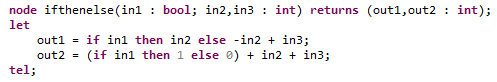
JKind supports relational operators > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to) for integer and real variables. Further, it supports = (equality) and <> (inequality) for all types.

The following example demonstrates the use of the relational operators within JKind.



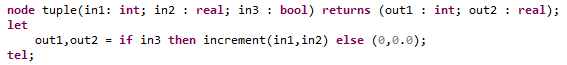
### If Then Else Expression

JKind supports the use of an in-expression if-then-else. This can be used as a conditional expression for variable assignment, etc.



### Tuple Expressions

JKind does not provide a tuple type, but does allow for tuple expressions. Tuple expressions can be used to complete conditional node/function calls, such as in the example shown below:



# Command Line Flags

JKind provides numerous user flags to modify the tools inputs, outputs, and operational characteristics. The available flags for JKind can be obtained by executing the binary at the command line with the ***-help*** flag. By doing so the tool will output the information in Table 1.

|  |
| --- |
| usage: jkind [options] <input>  -excel generate results in Excel format  -help print this message  -induct\_cex generate inductive counterexamples  -inv\_gen\_level <arg> invariant generator level for more and more  invariants (default: 0, alternatives: 1, 2, 3)  -ivc find an inductive validity core for valid  properties (based on --%IVC annotated elements)  -main <arg> specify main node (overrides --%MAIN)  -n <arg> maximum depth for bmc and k-induction (default:  unbounded)  -no\_bmc disable bounded model checking  -no\_inv\_gen disable invariant generation  -no\_k\_induction disable k-induction  -no\_slicing disable slicing  -pdr\_max <arg> maximum number of PDR parallel instances (0 to  disable PDR)  -read\_advice <arg> read advice from specified file  -scratch produce files for debugging purposes  -smooth smooth counterexamples (minimal changes in input  values)  -solver <arg> SMT solver (default: smtinterpol, alternatives:  z3, yices, yices2, cvc4, mathsat)  -timeout <arg> maximum runtime in seconds (default: unbounded)  -version display version information  -write\_advice <arg> write advice to specified file  -xml generate results in XML format  -xml\_to\_stdout generate results in XML format on stardard out |

Table 1 - JKind user flags

The rest of this section will discuss each flag and how it modifies JKind’s operation.

## Configuration Flags

Table 2 identifies all of the flags that modify JKind’s analysis.

|  |  |
| --- | --- |
| ***Flag*** | ***Description*** |
| ***-solver*** | Selects the SMT solver that JKind uses for bounded model checking (BMC), k-induction, and invariant generation engines. By default the solver is SMTInterpol, other supported options are Z3, Yices, Yices2, CVC4, and MathSAT. |
| ***-no\_bmc*** | Disables the bounded model checking (BMC) engine. *Note: this will prevent the k-induction engine from proving or disproving properties.* |
| ***-no\_inv\_gen*** | Disables the invariant generation engine. |
| ***-no\_k\_induction*** | Disables the k-induction engine. |
| ***-no\_slicing*** | Disables slicing of Lustre variables that do not affect specified properties. |
| ***-pdr\_max*** | Sets the maximum number of property directed reachability (PDR) engines. Default is 2. Setting to 0 disables PDR. |
| ***-n*** | Sets the maximum depth for bounded model checking and k-induction. Default is unbounded. |
| ***-timeout*** | Sets the maximum runtime for the analysis engines in seconds. Default is unbounded. |
| ***-smooth*** | Use the smoothing algorithm to minimize variable changes in counterexamples. Note: this flag requires the selected solver to be Z3 or Yices. |
| ***-main*** | Specify a main node to override the --%MAIN annotation found in the Lustre file. |
| ***-ivc*** | Identify the inductive validity core for valid properties. This requires the user to identify the IVC elements. |
| ***-write\_advice <filename>*** | Writes analysis information regarding proven invariants and user properties to an external file so it may be reused on subsequent verification runs. |
| ***-read\_advice <filename>*** | Reads and uses previous analysis run information in current run from specified filename. |
| ***-inv\_gen\_level*** | Sets the level of invariant generation. Default is 0 (basic invariant generation). Setting to 1 is suitable for properties with long-running timers and when using physical types to partition variables and constants of the same type e.g. speed, acceleration, etc. Setting to 2 is suitable for long-running timers when not using specific physical types. Setting to 3 is suitable when the other levels are not working. Increasing the inv\_gen\_level generates for the most of the models more and more invariants. The idea is to increase the level until obtaining the proof automatically. |

Table - JKind analysis configuration flags

## Output Flags

Table 3 identifies all of the flags that modify JKind’s output and presentation of results.

|  |  |
| --- | --- |
| ***Flag*** | ***Description*** |
| ***-excel*** | Write counterexamples into a Microsoft Excel spreadsheet. |
| ***-induct\_cex*** | Output counterexamples from the k-induction engine. |
| ***-xml*** | Output analysis results in XML format. |
| ***-xml\_to\_stdout*** | Output analysis results in XML format to the standard out device. |

Table - JKind output flags

## Miscellaneous Flags

Table 4 identifies miscellaneous flags.

|  |  |
| --- | --- |
| ***Flag*** | ***Description*** |
| ***-help*** | Prints the usage message found in Table 1. |
| ***-scratch*** | Produces the files passed to the SMT engines for debugging purposes. |
| ***-version*** | Outputs tool’s current version to the command line. |

Table - Miscellaneous JKind flags