Trapezoidal Generalization of Lustre with Uninterpreted Functions

David Greve

Advanced Technology Center, Rockwell Collins
david.greve@rockwellcollins.com

Abstract. Fuzzing is a form of robustness testing in which random, invalid or unusual inputs are applied to a system while monitoring its overall health. Model-based fuzzing is a fuzzing technique that employs a mathematical model of system behavior to guide the fuzzing process and explore behaviors that would otherwise be difficult to reach by chance. Whereas traditional fuzzing frameworks generate tests randomly, a model-based framework can deduce tests from a behavioral model using a constraint solver. We are developing *FuzzM*, a model-based fuzzing framework that employs Lustre as a modeling language and leverages the JKind model checker as a constraint solver. Because the state space being explored by the fuzzer is often large, the rapid generation of test vectors is crucial. The need to generate tests quickly, however, is antithetical to the use of a constraint solver. Our solution to this problem is to use JKind to generate an initial solution and then to perform trapezoidal generalization of the solution relative to the Lustre specification. Test generation then consists of rapid, repeated, randomized sampling of trapezoidal generalization spaces. Trapezoidal generalizations are ordered, hierarchical conjunctions of linear constraints. They are more expressive than simple intervals but are more efficient to manipulate and easier to sample than generic polytopes. In this paper we describe an approach to the trapezoidal generalizations of JKind counterexamples relative to Lustre models with integer division, remainder, and uninterpreted functions. We demonstrate our approach on a Lustre specification that recognizes permutations of complete integer sequences by generalizing a counterexample sequence satisfying the permutation property to produce a family of permuted sequences that can be rapidly sampled.

1 Motivation

Fuzzing (or fuzz testing) is a form of robustness testing in which random, invalid or unusual inputs are applied while monitoring the overall health of the system.

This work was sponsored by DARPA/AFRL Contract FA8750-16-C-0218. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited

Fuzz testing has been successful in finding bugs. The efficacy of random fuzzing, however, is often limited by even simple software well-formedness checks that are unlikely to succeed on random data. CRC checks on Ethernet packets, for example, are unlikely to be correct by chance thus most packets generated at random are likely to be immediately discarded. Smart fuzzers have emerged that allow the user to specify data format templates while computing essential content such as CRCs programmatically. Random (but well structured) inputs are then constructed by the fuzzer by filling in these data templates.

Model-based fuzzing is an evolution of smart fuzzing that, in addition to data formats, employs a mathematical model of expected system behavior to guide the fuzzing process. Whereas smart fuzzing frameworks construct fuzz tests by simply filling in data structure templates, a model-based framework can deduce tests from a behavioral model using a constraint solver. Testing objectives can be expressed mathematically as logical constraints and those constraints, along with the behavioral model, can be passed to a constraint solver that deduces an input sequence that will cause the model (and, presumably, the device under test) to satisfy the testing objective. Using a constraint solver in this manner enables the creation of high-quality tests capable of targeting deep system behaviors that random testing alone would be unlikely to reach.

The objective of model-based fuzzing is to find bugs in systems whose abstract behavior is described by a model, not to identify bugs in the behavioral system model itself. While progress against traditional test coverage metrics can be measured against the model, the objective of fuzz testing is to explore the behavior of a system beyond the model to search for otherwise unknown vulnerabilities. Because the unknown state space being explored by the fuzzer is likely to be substantially larger than the model state space, effective exploration of that space requires the rapid generation of large numbers of random test vectors. Unfortunately, the need to generate tests quickly is antithetical to the use of a constraint solver. In addition to performance limitations, it can be surprisingly difficult to induce a constraint solver to generate random solutions. Rather, solutions tend to cluster around model and constraint features or near special values like zero. While specially crafted hypotheses can be employed to drive the solver away from previous solutions, the presence of large, complex hypotheses can degrade solver performance even further.

Generalization can be employed to help to meet the objectives of model-based fuzzing by decoupling the constraint solving from test generation. Generalization is a technique that converts a single constraint solution into a set of solutions that cover the state space in the region of the original solution. Given a generalized solution, test generation can be seen as a matter of sampling the solution set. With trapezoidal generalizations, this sampling process is amenable to efficient implementation and is capable of rapidly producing large numbers of high-quality tests from a single solver invocation. The sampling of the solution set can also be randomized, shifting much of the onus of generating desirable, random test distributions from the solver to the sampler.

This paper describes a trapezoidal generalization technique developed for the model-based fuzzing framework *FuzzM*. FuzzM leverages Lustre as a modeling language and the JKind model checker as a constraint solver. JKind produces solutions (counterexamples) that exhibit potentially hard to reach model behaviors while generalization is relied upon to provide the bandwidth needed to explore the state space around those solutions in search of proximate vulnerabilities in the target system. We say that the constraint solver is used to target known behaviors and the generalizer is used to fuzz unknown behaviors. This configuration allows us to amplify JKind solution streams of approximately 1 vector per second into fuzzing streams of nearly 2000 test vectors per second.

2 Lustre

Lustre is a formally defined, declarative, synchronous dataflow language designed for describing reactive systems and for expressing safety properties about such systems.[3]. While a series of specifications for the Lustre language have been published, no reasoning tool appears to support every language feature. Rather, most tools support both some subset and some superset of some version of the official language. Because we are leveraging the JKind model checker, we restrict our discussion to those fragments of the Lustre language supported by JKind.

3 JKind

JKind is an open-source infinite-state model checker for safety properties of synchronous systems. Systems and their properties are expressed in the Lustre language. Verification is based on k-induction and property directed reachability using back-end SMT solvers. A verified property is guaranteed to be true for all runs of the system. A falsified property is reported with an explicit counterexample demonstrating the property violation. In our FuzzM framework, JKind is employed as a constraint solver. Constraints are submitted to JKind as negated properties. When those properties are subsequently falsified, the resulting counterexample is a satisfying instance of the constraint. JKind is used as the back-end model checker for many other projects and tools at Rockwell Collins as well including the AGREE tool and the SIMPAL tool. JKind is designed to be cross-platform, reliable, and easy to extend, with power and performance as secondary goals. While JKind attempts to be largely compatible with pkind and Kind 2, this varies over time due to continued independent developments in each system.

4 Complete Set Challenge

In a recent effort to fuzz a network communication protocol we were faced with the challenge of modeling a transport layer fragmentation and reassembly protocol. The specification required that the receiver be capable of accepting transmission packets in any order and then, once a complete sequence is received, re-

construct the original network packet from the payloads of the individual transmission packets. To accomplish this, the transmitter assigns each fragmented transmission packet a sequence number. After all of the transmission packets have been sent, the receiver uses the sequence numbers to correctly order the packets. Only if the received packets contain a complete set of sequence numbers does reassembly take place,

To fuzz these requirements, we need a model that can cause the solver to generate packets with complete sets of sequence numbers, but appearing in any order. In other words, we need to define a Lustre predicate that can accept packets, in any order, and determine whether or not the sequence numbers from those packets constitutes a complete set of numbers over some interval.

In defining the complete set predicate, we observe that it is necessary for the sum of the sequence numbers to equal the sum of all of the numbers in the desired interval. Such a summation property is easy to express in Lustre. We have also proven that the aforementioned equality of the sums is sufficient to characterize a complete set if the collection of sequence numbers also contains no duplicates. Capturing this uniqueness property in Lustre, however, is not straightforward.

While, arguably, one could use an array to maintain a history of sequence numbers, this approach is cumbersome and limited by the fact that JKind supports only fixed, finite arrays. Another, more elegant, approach is to use uninterpreted functions. Assume that the evaluation of an uninterpreted function on each sequence number in turn is known to be equal to the value of a simple, increasing sequence. We then know that the output of the function, being a simple, increasing sequence, contains no duplicates. But if the outputs of the function are all different, then, by virtue of the fact that it is a function, the inputs must be different as well. In this way, we can use uninterpreted functions to identify sequences of numbers that contain no duplicates.

A Lustre specification that combines the aforementioned properties to recognize complete sets of input values is presented in Figure 1.

5 JKind UF Support

We recently added UF support to JKind to enhance the expressive capabilities of the FuzzM framework and to enable solutions to challenges such as complete set problem. A substantial concern when adding new features or theories into JKind is whether they will work generically across the supported solvers and engines. Another issue is that, while solver support is necessary for most theories, it constitutes only a small fraction of the actual coding effort. Most of the work tends to be in transforming the Lustre into a simplified form for the SMT solvers while preserving/handling the new theory. That is, most the work in adding new theory support is 'plumbing' and 'bookkeeping'.

Because SMTInterpol supports UF we were able to support UF with the PDR engine. All of the other SMT-LIB based solvers support UF so implementing this interface allowed us to advantage of this theory with Yices2, Z3, CVC4, and MathSAT. Most of the solvers supported by JKind are SMT-LIB compliant.

```
function unique(x: int) returns (y: int);
function min() returns (y: int);

node main(in: int) returns (complete_set: bool);
var
  pin: int;
  sequence: int;
  sumsq: int;
  sumin: int;
  ssize: int;
  interesting_length: bool;
  not_increasing: bool;
  not_decreasing: bool;
  sum_equiv: bool;
let

  --- Establish arbitrary bounds on the input
  assert(  in < 128);
  assert(-128 <= in);

  --- We do this to keep the solutions
  --- from being completely degenerate.
  not_decreasing = false -> ((in < (pre in)) or (pre not_decreasing));
  not_increasing = false -> ((in > (pre in)) or (pre not_increasing));

  --- Here we use an uninterpreted function to establish an arbitrary
  --- lower bound on the interval and say that all values are greater
  --- than or equal to it.
  assert(min() <= in);

  -- We bias our input stream by the arbitrary minimum to
  -- generate a sequence of non-negative values.
  pin = in - min();

  -- This assertion ensures that out input sequence
  -- contains no duplicates.
  sequence = 0 -> (pre sequence) + 1;
  assert(unique(pin) = sequence);

  -- The sum_equiv predicate ensures the sum of the input sequence is
  -- equal to the sum of a simple arithmetic sequence
  sumsq = (sequence -> sequence + (pre sumsq));
  sumin = (pin -> pin + (pre sumin));
  sum_equiv = (sumsq = sumin);

  -- FuzzM negates the properties it submits to JKind so that the
  -- resulting counterexamples satisfy the desired constraints.
  complete_set = (not (not_increasing and not_decreasing and sum_equiv));

  --%PROPERTY complete_set;

tel
```

Fig. 1. Lustre Complete Set Specification

Early versions of the SMT-LIB standard, however, did not standardize the syntax for describing uninterpreted function instances (counterexamples) and not all solvers support the most recent standard. Reporting solution results, therefore, proved challenging because each solver reported its results differently. Because the solvers often returned expressions that reflected a specific implementation, we opted to write code to evaluate the models reported by the solvers so that JKind can report the results as a simple table of function evaluations as in Table 1.

x	$f(x)$
0	1
3	1
4	2

x	y	$g(x,y)$
0	5	1
3	2	1
4	7	2

Table 1. Examples of JKind's Tabular Reporting of Function Instances

Our implementation supports uninterpreted functions that accept as inputs and produce as outputs values of any primitive or user defined type supported by JKind and they are allowed to return multiple values. Note, however, that during pre-processing JKind reduces complex functions into simpler functions that accept only primitive input types and return only a single primitive result type. JKind also supports functions with no inputs. UF is expected to work seamlessly with JKind's advice, smoothing, and interval generalization capability. UF has not been tested with IVC, but the interaction between UF and quantifiers may cause performance issues with Z3.

The output from our UF-enabled JKind when applied to the complete set model of Figure 1 is shown in Figure 2.

```
==========================================
  JKind 3.0.5-uf
==========================================

There are 1 properties to be checked.
PROPERTIES TO BE CHECKED: [complete_set]

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
INVALID PROPERTY: complete_set || bmc || K = 3 || Time = 0.115s
                           Step
variable                      0     1     2

INPUTS
in                            1     2     0

OUTPUTS
complete_set              false false  true

FUNCTIONS
----------
| min
----------
| 0

--------------------
x         | f
--------------------
1         | 0
2         | 1
0         | 2
```

Fig. 2. JKind Complete Set Solution

6 Generalization

Generalization is a technique that converts a single constraint solution into a set of solutions that cover the state space in the region of the original solution. The FuzzM framework employs generalization to decouple constraint solving from test generation. Given an appropriate generalization, the sampling process can be implemented efficiently and is capable of rapidly producing large numbers of high-quality tests from a single solver invocation. The sampling of the solution set can also be randomized and biased, shifting much of the onus of generating desirable, random test distributions from the solver to the sampler.

The FuzzM framework originally relied on JKind's interval generalization capability. An interval generalization consists of a list of intervals, one for each input at each step of the counterexample. An example interval generalization is given in in Table 2.

Lower Bound	Variable	Upper Bound
$100 <$	x	< 200
$0 <$	y	< 100
$-50 <$	z	< 50

Table 2. Example Interval Generalization

As a volume, an interval generalization looks like a multi-dimensional rectangle. Figure 3 illustrates what the interval generalization of Table 2 looks like as a volume in x-y-z space.

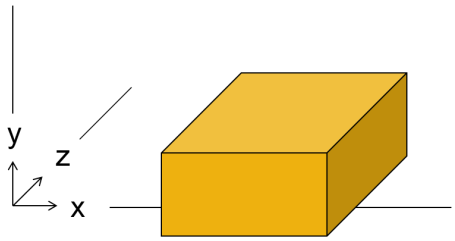

Fig. 3. Rectilinear Generalization Volume

Sampling an interval generalization is trivial: a value for an input is computed by randomly selecting a value from that input's associated interval. Uniform sampling means that every value in the interval is equally likely to be selected. Biased sampling means that values near the interval boundaries are preferred. We use heat maps to visualize the distribution of test vectors generated by the FuzzM framework. Heat maps are generated by color coding (red = more, blue = less) the relative frequencies of solutions from millions of samples from the

generalizations of solutions to thousands of constraint queries. The heat maps presented here are generated from arbitrary Boolean combinations of the three linear constraints of Equation 1 that define a triangular region, as shown in Figure 4.

$$\begin{aligned} 23(y-95) &> -88(x-24) \\ 82(y-2) &> 71(x-21) \\ 106(y-92) &< -47(x+1) \end{aligned} \tag{1}$$

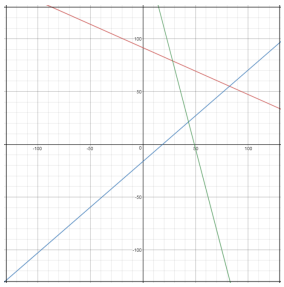

Fig. 4. Graph of Linear Features used to Generate Heat Maps

Figure 5 shows heat maps for uniform (left) and biased (right) sampling of interval generalizations. While JKind's generalization capability did enable a substantial boost in our fuzzing bandwidth, the inherent limitations of rectilinear generalization resulted in sensitivity to the initial solutions and poor alignment between interval boundaries and linearly dependent model features. Artifacts of these issues are visible in the heat maps as clumping in the distributions and the fact that, even with biased sampling, the distributions appear to avoid the linear feature boundaries.

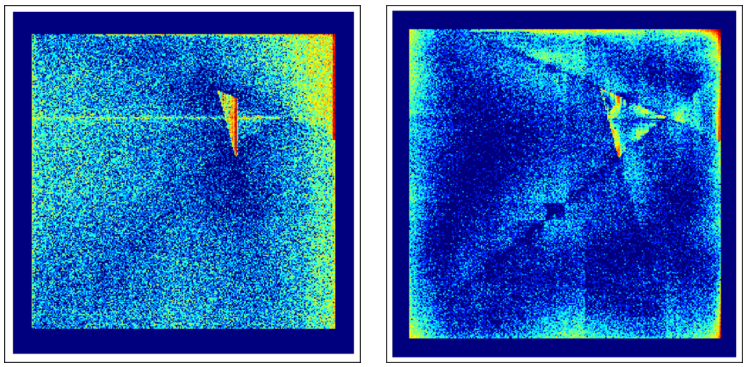

Fig. 5. Uniform and Biased Heat Maps for Interval Generalization

To address these issues we developed a trapezoidal generalization technique that substantially improves the fuzzing quality of the resulting vector set without undue computational penalty[2]. A trapezoidal generalization consists of an ordered list of variables whose interval bounds are rational first-order multivariate polynomials expressed in terms of previous variables, as shown in Table 3.

Lower Bound	Variable	Upper Bound
$100 <$	x	< 200
$3x - 290 <$	y	$< -3x + 970$
$y + x - 250 <$	z	$< -y + 7$

Table 3. Example Trapezoidal Generalization

As a volume, a trapezoidal generalization resembles a multi-dimensional trapezoid. Figure 6 illustrates what the trapezoidal generalization of Table 3 looks like as volume in the x-y-z space.

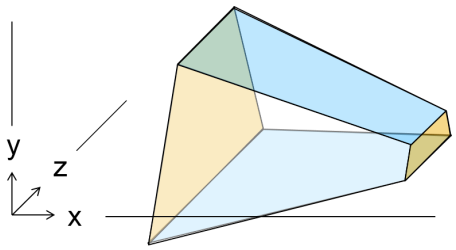

Fig. 6. Trapezoidal Volume

The size of a trapezoidal representation is worst case quadratic and operations over the representation are worst case cubic in the total number of inputs (ie: *model inputs*$*$*unwindings*). Trapezoidal generalizations provide a better approximation of feature boundaries, enhancing boundary value testing, and they are generally larger (in volume) than similar rectilinear generalizations, allowing each counterexample to yield more test vectors.

Sampling a trapezoidal solution space is only slightly more complex than sampling a rectilinear space. The sampling is done sequentially and each input assigned a value selected from the interval obtained by evaluating the upper and lower bound expressions relative to the inputs that have already been assigned. Because each variable bound is expressed in terms of earlier variables, this procedure is guaranteed to yield concrete bounds for each variable choice.

Figure 7 shows uniform (left) and biased (right) heat map images generated using trapezoidal generalizations. Note that in the uniform image the distribution is nearly piece-wise uniform over the entire state space. This is a result of

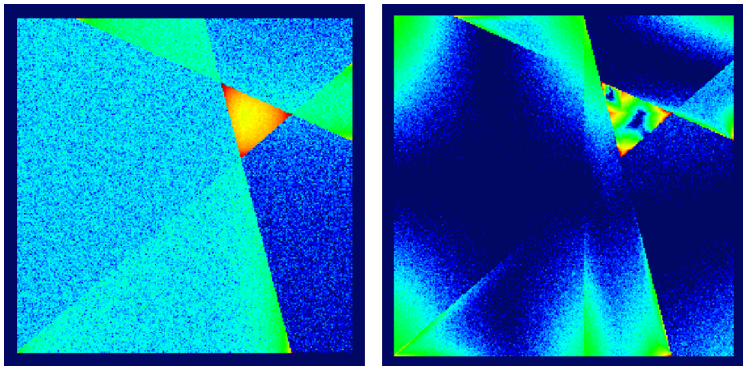

Fig. 7. Uniform and Biased Trapezoidal Heat Maps

larger generalizations that better approximate the linear features of the model. Note, too, that biased solutions tend to cluster near the boundaries and intersections of linear features.

6.1 Generalizing UF

Generalization is a crucial feature of the FuzzM framework. We rely on generalization to randomize solver results, to generate tests with specific statistical distributions, and, above all, to improve performance. When adding new theory support it is therefore essential to consider the impact that the theory will have on generalization. Even if it is possible to generalize around a theory, if the resulting generalization is too restrictive, the generalization process may not provide the flexibility or performance required to make it feasible. An important contribution of this work is in demonstrating a technique for generalizing solutions involving UF instances into a manageable set of linear constraints that offer reasonably large solution spaces.

We illustrate the operation of our algorithm with an example using the the uninterpreted function declared in Figure 8.

```
function f(x: int, y: real, z: bool) returns (a: int);
```

Fig. 8. An uninterpreted function with int, real, and a bool arguments

Table 4 illustrates a counterexample that provides several function instances of $f(x, y, z)$.

x	y	z	$f(x,y,z)$
4	3.0	T	7
4	3.0	F	9
0	3.0	F	7

Table 4. A UF Function Instance

Our objective in generalizing this solution is to replace the constant values in Table 4 with arbitrary symbolic variables, as in Table 5.

x	y	z	$f(x,y,z)$
$x43T7$	$y43T7$	$z43T7$	$f43T7$
$x43F9$	$y43F9$	$z43F9$	$f43F9$
$x03F7$	$y03F7$	$z03F7$	$f03F7$

Table 5. A UF counterexample instance for $f(x,y,z)$

Simply doing this, however, would ignore the fact that the inputs and outputs of an uninterpreted function are constrained by the UF axiom. The UF axiom for $f(x,y,z)$ is shown in Equation 2. It says that the evaluation of two different instances of f must be the same if the same arguments passed to them are the same.

$$\frac{x1 = x2 \wedge y1 = y2 \wedge z1 = z2}{f(x1,y1,z1) = f(x2,y2,z2)} \quad \text{UF Axiom for } f(x,y,z) \tag{2}$$

The contrapositive form of the UF axiom, presented in Equation 6.1, says that if two function instances produce different output values then at least one of the inputs to the instances must be different.

$$\frac{f(x1,y1,z1) \neq f(x2,y2,z2)}{x1 \neq x2 \vee y1 \neq y2 \vee z1 \neq z2} \quad \text{Contrapositive Form}$$

It would be inconsistent with the UF axiom to simply replace the values in each function instance with the variables of Table 5 because it is possible to select values for those variables that violate this axiom. Rather, the variables need to be constrained so that any variable assignment that satisfies the constraint also satisfies the UF axioms.

To do this, we start with the unique function instances provided in the counterexample. Presumably those instances satisfy the UF axiom because they were generated by the constraint solver. For each numeric input to the function, we sort the list of values associated with that input. For numeric inputs we use the

resulting list to identify an appropriate linear relationship between the pairs of values in the list. We then constrain the generalized variables associated with each instance by substituting them into those linear relationships as shown in Tables 6 and 7.

cex	0	$<$	4	$\leq$	4
x	$x03F7$	$<$	$x43T7$	$\leq$	$x43F9$

Table 6. Ordering constraints on generalizations of x

cex	3.0	$<=$	3.0	$<=$	3.0
y	$y43T7$	$<=$	$y43F9$	$<=$	$y03F7$

Table 7. Ordering constraints on generalizations of y

For Boolean inputs we use the resulting list to identify an appropriate equality relationship between the pairs of values in the list. We then generate a set of constraints by substituting the generalized Boolean variables from Table 5 into those equality relationships[1], as in 8.

cex	F	$=$	F	$\neq$	T
z	$z43F9$	$=$	$z03F7$	$\neq$	$z43T7$

Table 8. Equality constraints on the generalizations of z

Any solution for the generalization variables that satisfies the constraints resulting from this process is guaranteed to preserve the ordering (or equivalence) relationship that existed in the original solution. That ordering (equivalence) relationship ensures that the inputs to any two distinct function instances in Table 5 will differ by at least one input, which will always be sufficient to satisfy the UF axiom regardless of the values chosen for function outputs.

7 Trapezoidal Generalization in Lustre

We now describe how trapezoidal generalization is performed in Lustre. We begin with an overview of the Lustre syntax upon which our generalization algorithm operates. We then present an overview of the various data structures and methods used in the generalization process, how the process is initialized, and how symbolic simulation is performed to compute the final generalization result.

7.1 Supported Lustre Syntax

```
program: (node | function)* EOF ;

node:
  'node' ID '(' varDeclList? ')'
  'returns' '(' varDeclList? ')' ';'
  ('var' varDeclList ';' )?
   'let' ( equation | property | assertion )* 'tel'
  } ;

function: 'function' ID '(' varDeclList? ')'
          'returns' '(' varDeclList? ')' ';'
  ;

varDeclList: varDecl (';' varDecl)* ;

varDecl: ID ':' type ;
type: ('int' | 'bool' | 'real') ;

property: '--%PROPERTY' ID ';' ;
assertion: 'assert' expr ';'   ;
equation: ID '=' expr ';'      ;

expr: ID                                                  # idExpr
    | (INT | REAL | BOOL)                                 # constantExpr
    | 'real'  '(' expr ')'                                # castExpr
    | 'floor' '(' expr ')'                                # castExpr
    | 'pre' expr                                          # preExpr
    | expr '->' expr                                      # arrowExpr
    | 'not' expr                                          # notExpr
    | '-' expr                                            # negateExpr
    | expr ('*' | '/' | 'div' | 'mod' | '+' | '-') expr # arithmeticExpr
    | expr ('<' | '<=' | '>' | '>=' | '=' | '<>') expr  # relationExpr
    | expr ('and' | 'or' | 'xor' | '=>' ) expr           # binaryExpr
    | 'if' expr 'then' expr 'else' expr                   # ifThenElseExpr
    | ID '(' (expr (',' expr)*)? ')'                      # callExpr
;
```

Fig. 9. Lustre Grammar supported by Trapezoidal Generalization

Generalization can be applied to any Lustre specification supported by JKind. However, we rely on methods from JKind to pre-process the original Lustre model into a more primitive form semantically equivalent to the original but easier to work with. As a result, the syntax upon which our generalization algorithm operates is a subset of the syntax supported by JKind proper. The grammar for that subset of Lustre is given in Figure 9

The primitive Lustre types are Booleans (`bool`), unbounded integers (`int`) , and reals (`real`). In JKind, reals are interpreted as rationals so we use the terms 'real' and 'rational' interchangeably. Lustre also supports user defined record and array types and provides syntax for accessing and updating such structures. Instances of complex user defined data-types, however, are reduced to instances of the primitive data types during pre-processing. Since JKind supports only finite arrays, even array instance are expanded to their constituent elements while array accesses are transformed into if-then-else expressions. Records and tuples are similarly affected and multiple-value assignments are reduced to multiple single assignments. Pre-processing is also applied to functions. If a function has as an argument a complex data type, the signature of the function is extended to accept each primitive element of that data type. If a function returns a complex data type (or tuple), new function names are introduced for each primitive type. JKind supports the built-in $real()$ function to cast integer values into real values, but it does not support $int()$. Rather it supplies a built-in $floor()$ function for that purpose. We support two temporal operators, `pre` and `->` (arrow), a conditional if-then-else expression, and a number of Boolean and arithmetic operators including integer division and modulus. After pre-processing the Lustre specification is reduced to a single node. The body of that node contains only variable declarations, equations, assertions, and properties. Any call expression remaining after pre-processing is a call to a function.

7.2 Data Structures for Generalization

In explaining the generalization process we begin with an overview of the basic data structures we employ. The primitive Lustre numeric types are reflected in `Number`, which may be unbounded integers or reals (rationals). More generally any Lustre primitive value can be captured using `LustreValue` which includes Booleans.

```
type Number      = ( int | real );
type LustreValue = ( Number | bool );
```

Objects of type `LustreTypeName` allow us to manipulate Lustre type names explicitly and the method `isBool()` allows us to distinguish between numeric and Boolean types.

```
type LustreTypeName = {
  isBool();
};
```

We assume that the Lustre syntax parses into AST objects and that expressions map into objects that extend `Expr`. Again, the `isBool()` method gives us

the ability to distinguish between numeric and Boolean expressions. The generalization method operates recursively over the expression syntax. It accept an integer argument, the current unwinding step, and returns an instance of the generalization type, `GenType`. The ID expression is used to represent Lustre variable and function names and the Lustre literal expressions implement methods to translate their representations into primitive type values.

```
type Expr = {
  GenType generalize(int);
  bool isBool();
};

type ID extends Expr;
type LITERAL extends Expr = { LustreValue getValue(); }
type INT  extends LITERAL;
type BOOL extends LITERAL;
type REAL extends LITERAL;
```

Model inputs generalize into `Variable`s. The new variable constructor (allocator) takes a `LustreTypeName` and a `LustreValue`. The primitive value supplied to the `Variable` constructor is available via the `cex` field. Technically variables also have names and there is a total ordering among them. However, these capabilities are not used by the algorithms discussed in this paper.

```
type Variable = {
  LustreValue cex;
  Variable newVar(LustreTypeName,LustreValue);
};
```

The generalization process produces a result of type `GenType` which will always be either a Polynomial or a Trapezoid. As with variables, the value of the generalization evaluated at the counterexample is stored in the `cex` field. An abstract constructor, `TorP()` allows generalizations to be constructed based on the type of the given `Variable` or `Number`.

```
GenType : type = {
  LustreValue cex;
  GenType TorP(Variable);
  GenType TorP(Number);
}
```

Numeric Lustre expressions generalize into Polynomials (`Poly`). A `Poly`'s evaluation under the counterexample is stored in the `cex` field as a Number. It is possible to construct a `Poly` from a constant `Number` or from a numeric `Variable`. Constant polynomials are recognized by `isConst()`. A polynomial is divisible (`|`) by an integer if all of its coefficients are divisible by that integer. Standard arithmetic operations over polynomials are supported as are multiplication and division by constants.

```
type Poly extends GenType = {
  Number cex;
  Poly poly(Number);
  Poly poly(Variable);
  bool isConst();
  bool (Poly | int);
  Poly (Poly + Poly);
  Poly (Poly - Poly);
  Poly (- Poly);
  Poly (Number * Poly);
  Poly (Poly / Number);
}
```

Boolean Lustre expressions generalize into Trapezoids (`Tzoid`) which are symbolic expression involving linear relations over polynomials. A `Tzoid`'s evaluation under the counterexample is stored in the `cex` field as a Boolean. It is possible to construct a `Tzoid` from a constant `bool` or from a Boolean `Variable`. The relational operations on Polynomials produce Trapezoids. Trapezoids can be combined via conjunction ($\wedge$) and disjunction ($\vee$) and they can be negated using the negation operation ($\sim$).

```
type Tzoid extends GenType = {
  bool cex;
  Tzoid tzoid(bool);
  Tzoid tzoid(Variable);

  // Boolean Combinations of Trapezoids
  Tzoid (Tzoid /\ Tzoid);
  Tzoid (Tzoid \/ Tzoid);
  Tzoid (~ Tzoid);

  // Linear Relations over Polynomials
  Tzoid (Poly <  Poly);
  Tzoid (Poly <= Poly);
  Tzoid (Poly >  Poly);
  Tzoid (Poly >= Poly);
  Tzoid (Poly =  Poly);
  Tzoid (Poly <  Number);
  Tzoid (Poly <= Number);
  Tzoid (Poly >  Number);
  Tzoid (Poly >= Number);
  Tzoid (Poly =  Number);
}
```

Function signatures are stored in an `FSigType`. This data structure contains the Lustre type names of the various arguments (`arg`) and of the return value (`value`). An `FSigMapType` maintains a collection of function signatures, indexed by a function name.

```
type FSigType = {
  LustreTypeName arg[int];
  LustreTypeName value;
};

type FSigMapType = FSigType[ID];
```

An uninterpreted function instance is represented using an `FInstType`. This data structure contains the Lustre values of the various arguments (`arg`) and of the return value (`value`). An `FInstMapType` is a collection of lists of function signatures, indexed by a function name.

```
type FInstType = {
  LustreValue value;
  LustreValue arg[int];
};

type FInstMapType = FInstType[ID][int];
```

A function instance generalization is represented using an `FGenType`. This data structure contains generalization data structures representing the various arguments (`arg`) and the return value (`value`). An `FGenMapType` is a collection of `FGenType`s indexed by a list of `LustreValue`s that represent concrete argument values applied to the function. The method `getNthArgs(int)` returns the list of generalizations associated with the function's n'th argument. `FGenMapTypeMap` is a collection of collections of function generalizations (`FGenMapType`) indexed by a function name.

```
type FGenType = {
  GenType value;
  GenType arg[int];
};

type FGenMapType  = FGenType[LustreValue[]] {
  GenType[] getNthArgs(int);
}

type FGenMapTypeMap = FGenMapType[ID];
```

Finally we have a polymorphic len() operator that returns the length of a list or collection.

7.3 Algorithm Initialization

The initial state of the system is completely determined by the counterexample. The counterexample includes assignments to all of the system inputs at each step of the model unwinding. It also includes tables enumerating all relevant uninterpreted function evaluations. Evaluating the Lustre model on the counterexample will satisfy all of the assertions in the model in every step of the unwinding and will falsify the property in the finals step of the unwinding. The generalization process begins by generalizing the initial state of the system, both the inputs and the uninterpreted functions.

The Global State There a several global variables used in the generalization process that must be initialized and maintained properly to ensure the correctness of our results. The global `cache`, when it hits, maps variable ID's at different steps of the unwinding to a generalization result. It is used to store the initial generalization of the system inputs as well as to cache intermediate generalization results for the sake of efficiency. Information about how each uninterpreted function instance has been generalized is stored in a the global table `FGenMap` that can be accessed when UF instances are encountered in the course of the generalization process. Finally, the global Trapezoid `G`, initially True, is used to accumulate invariants that arise during the generalization of certain expressions.

```
global GenType     cache[int][ID];
global FGenType    FGenMap[ID][LustreValue[]];
global Tzoid       G = tzoid(true);
```

Initializing System Inputs Generalizing the initial system inputs involves iterating over all of the system inputs in each step of the unrolling specified by the original counterexample. At each step, each input is associated with a unique, new variable. Upon creation, variables are given a value that corresponds to the value of the input at that specific step of the unrolling in the original counterexample. Binding variables to their original value from the counterexample allows us to later compute the value of arbitrary expressions under the original counterexample, a feature that is crucial in maintaining the satisfiability of our final generalization result. After allocation, each variable is stored in the `cache` as an appropriate generalization based on it's Lustre type. The constructor TorP(Variable) will construct either a Trapezoid for Boolean variables or a Polynomial for numeric variables.

Initializing Uninterpreted Functions UF generalization begins by creating fresh variables for each input and output of each unique instance of each function, a la Table 5. Note that, just as with system inputs, the variables created here are associated with the values of the values they generalize in the counterexample. The global `FGenMap` data structure being created is indexed first by a function name and then by a list of values that correspond to the arguments of one of

```
procedure initializeInputs(LustreValue cex[int][ID]) {
  global GenType  cache[int][ID];      // the global cache
  for (i=0;i<cex.len();i++) {          // For each step of the unrolling ..
    foreach (ID v,type in Input) {     // For each input ..
      var <- newVar(type,cex[k][v])    // .. allocate a new variable
      cache[i][v] <- TorP(var)         // Cache the appropriate generalization
    }
  }
}
```

Fig. 10. Generalization Initialization Procedure

the known function instances and returns a generalized function instance that contains the generalizations for each of the inputs and the outputs.

```
procedure generalizeUF(FInstType finstListMap[ID][int]) {
  global FGenType FGenMap[ID][LustreValue[]]; // The global FGen data structure
  foreach (FSigType fsig in Functions) {      // For each function (signature) ..
    ID fname <- fsig.name
    FInstType[] finstList <- finstListMap[fname] // get all fname instances
    FGenType[LustreValueType[]] genMap <- {}
    foreach (FInstType finst in finstList) {  // For each function instance ..
      FGenType fgen;
      for (n=0;n<finst.arg.len();n++) {        // For each argument position ..
        LustreValueType argn <- finst.arg[i]  // argument value
        LustreTypeName  type <- fsig.arg[i]   // argument type
        Variable avar <- newVar(type,argn)    // .. allocate a fresh variable
        fgen.arg[i] <- TorP(avar)             // use appropriate generalization
      }
      LustreValueType fval  <- finst.value    // function instance value
      LustreTypeName  ftype <- fsig.value     // function return type
      Variable fvar <- newVar(ftype,fval)     // allocate a fresh return variable
      fgen.value <- TorP(fvar)                // use appropriate generalization
      genMap[finst.arg] <- fgen               // Add to the generalized instance map
                                              // keyed by all argument values
    }
    FGenMap[fsig.name] <- genMap              // Update the global data structure
  }
}
```

Fig. 11. Generalization of UF Instances

Having generalized the function instances, we now compute the invariants essential to satisfy the UF axioms and add them to the set of global invariants. The following procedure implements that algorithm outlined in Section 6.1, sorting the generalized arguments by their value in the counterexample and then imposing that same ordering on the generalized variables.

7.4 Generalization Algorithm

Unlike rectilinear generalization, which requires multiple interval simulations to compute, a trapezoidal generalization is performed in a single symbolic simulation of the Lustre model. Generalization is performed depth-first, evaluating the inputs to each expression before using those results to generalize the expression itself. At the top level, the generalization process is driven by a specific property and any assertions contained in the Lustre model. Note that all assertions are assumed to be relevant in constraining the counterexample, even if they might

```
procedure constrainUF (FInstType finstListMap[ID][int]) {
  global FGenType FGenMap[ID][LustreValue[]]; // The global FGen data structure
  global Tzoid G;                             // The global invariants
  foreach (FSigType fsig in Functions) {      // For each function (signature) ..
    for (n=0;n<fsig.arg.len();n++) {          // For each function argument ..
      argnList <- FGenMap.getNthArgs(n)       // Get all n'th generalized arguments
      argnList <- argnList.sort(.cex)         // Sort all n'th arguments in ascending
                                              // order by cex
      LustrueTypeName type = fsig.arg[n]
      if type.isBool() {                      // for a Boolean argument
        for (i=1;i<argnList.len();i++) {      // starting with the second instance ..
          Tzoid vi <- argnList[i]             // Current variable
          Tzoid vm <- argnList[i-1]           // Previous variable
          if (vm.cex == vi.cex) {             // Compare cex values
            G <- G /\ (vm = vi)               // Ensure always that vm = vi
          } else {
            G <- G /\ (vm = ~vi)              // Ensure always that vm = ~vi
          }
        }
      } else {                                // for a Numeric argument
        for (i=1;i<argnList.len();i++) {      // starting with second instance ..
          Poly vi <- argnList[i]              // Current variable
          Poly vm <- argnList[i-1]            // Previous variable
          if (vm.cex < vi.cex) {              // Compare cex values
            G <- G /\ (vm < vi)               // Ensure always that vm < vi
          } else {
            G <- G /\ (vm <= vi)              // Ensure always that vm <= vi
          }
        }
      }
    }
  }
}
```

Fig. 12. Generate UF Global Invariants

be provably irrelevant to the truth of the property. The final result is computed as the conjunction of all of the following:

- Each assertion evaluated at each step of the unwinding
- The negation of the property evaluated in the final step
- The final global state after all of these evaluations

The top level algorithm for the generalization process is given in Figure 13.

```
Trapezoid generalizeProperty(ID property, int k) {
  global Trapezoid G;                 // The global constraint
  for (i=0;i<k;i++) {                 // For each step of the unrolling ..
    foreach (a: Expr in Assertions) { // For each assertion ..
      A <- a.generalize(i)            // generalize the assertion at step i
      G <- G /\ A                     // intersect w/global constraint
    }
  }
  P <- property.generalize(k-1)       // generalize the property at step k-1
  return G /\ ~P                      // The final result is the intersection
                                      // of the negation of the property with
                                      // the final global constraint
}
```

Fig. 13. Generalization Algorithm

The symbolic evaluation of Lustre expressions required for generalization is performed by extending one of the generic evaluation visitors patterns provided

by the JKind framework. The visitor maps recursively over the syntax tree defined by the expression grammar. We describe the behavior of the generalizer using production rules based on that grammar. In the following diagrams, the symbol $\xrightarrow{\mathcal{T}}$ denotes result of applying the generalization method at unwinding step T and any global side effect of applying a given production rule are denoted by actions appearing to the right of the rule.

Temporal Operators The Lustre language can express both temporal behaviors and properties through the use of its temporal operators. Because a Lustre specification may involve temporal properties, a single JKind counterexample may span multiple time steps. We say that multiple time steps in a JKind counterexample coresponds to multiple "unrollings" of the Lustre model. At each unrolling, the inputs and outputs of the Lustre model may assume new values because they represent the state of the system at different time steps.

The semantics of the binary `'->'` operator are such that it evaluates to its left hand side expression at time step 0 and to its right hand side expression in any subsequent time step. The unary `pre` operator returns the value of its argument evaluated in the previous time step. We say that a `pre` operator is "guarded" if it appears in the right hand side of an `->` operator. An "unguarded pre" operators may require the evaluation of an expression in a time step before 0. Unguarded pre operators are not supported.

$$\frac{\mathcal{T} > 0 \quad E \xrightarrow{\mathcal{T}-1} P}{(\texttt{`pre'}\ E) \xrightarrow{\mathcal{T}} P}$$

$$\frac{\mathcal{T} = 0 \quad E1 \xrightarrow{\mathcal{T}} P1}{(E1\ \texttt{`->'}\ E2) \xrightarrow{\mathcal{T}} P1} \qquad \frac{\mathcal{T} > 0 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1\ \texttt{`->'}\ E2) \xrightarrow{\mathcal{T}} P2}$$

Variables Every variable appearing in the Lustre model is either an input, output, or a local variable. If the variable is an output or a local variable, it also has a single defining equation in the body of the Lustre node. If a variable is an input, our initialization procedure ensures that it will have a binding in the global cache. If a variable is not an input, the Lustre language ensures that it is defined by an equation. If a variable is not already bound in the cache, we compute its value by evaluating its defining equation. After evaluating the equation, we update the cache with the computed value for the variable. This caching of computed values substantially improves generalization performance.

$$\frac{R = cache[T\mathcal{T}][ID] \quad R \neq \bot}{ID \xrightarrow{\mathcal{T}} R}$$

$$\frac{cache[\mathcal{T}][ID] = \bot \quad (ID\ \texttt{`='}\ E\texttt{`;'}) \quad E \xrightarrow{\mathcal{T}} R}{ID \xrightarrow{\mathcal{T}} R} \quad \text{cache}[\mathcal{T}][\text{ID}] \leftarrow \text{R}$$

if then else If the type of the if-then-else expression is Boolean, we can logically combine the condition with the two branches. However, since we have no way to represent conditional polynomials, we must restrict our final result to reflect only either the true of false branch of the condition if the type is numeric. To do this, we conjoin either the condition or its negation (whichever was true under the counterexample) with the global invariant and return the generalization of the selected branch.

$$\frac{E1.isBool() \quad E1 \xrightarrow{\mathcal{T}} R1 \quad E2 \xrightarrow{\mathcal{T}} R2 \quad E3 \xrightarrow{\mathcal{T}} R3}{(\texttt{`if'}\ E1\ \texttt{`then'}\ E2\ \texttt{`else'}\ E3) \xrightarrow{\mathcal{T}} (R1 \cap R2) \cup (\sim R1 \cap R3)}$$

$$\frac{\neg E1.isBool() \quad E1 \xrightarrow{\mathcal{T}} R1 \quad R1.cex \quad E2 \xrightarrow{\mathcal{T}} P2}{(\texttt{`if'}\ E1\ \texttt{`then'}\ E2\ \texttt{`else'}\ E3) \xrightarrow{\mathcal{T}} P2}\ \mathrm{G} \leftarrow \mathrm{G} \cap \mathrm{R1}$$

$$\frac{\neg E1.isBool() \quad E1 \xrightarrow{\mathcal{T}} R1 \quad \neg R1.cex \quad E3 \xrightarrow{\mathcal{T}} P3}{(\texttt{`if'}\ E1\ \texttt{`then'}\ E2\ \texttt{`else'}\ E3) \xrightarrow{\mathcal{T}} P3}\ \mathrm{G} \leftarrow \mathrm{G} \cap \sim \mathrm{R1}$$

Rational Division and Multiplication We only support the generalization of linear models. For rational division this means that the denominator must generalize to a constant polynomial. To ensure this we create a constant poly using the value of the denominator evaluated at the counterexample. We then add a global constraint that asserts that the denominator expression is always equal to that value, effectively linearizing the result around that solution.

$$\frac{\begin{array}{l} E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \\ d = P2.cex \quad D = poly(d) \quad R = \langle P2 = D \rangle \end{array}}{(E1\ \texttt{`/'}\ E2) \xrightarrow{\mathcal{T}} P1/d}\ \mathrm{G} \leftarrow \mathrm{G} \cap \mathrm{R}$$

The linear model restriction also means that we only support the multiplication of Polynomials by a constant. If neither argument generalizes to a constant polynomial, we create a constant poly using the value the multiplicand evaluated at the counterexample. We then add a global constraint that asserts that the multiplicand is always be equal to that value, effectively linearizing the result around that solution.

$$\frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \quad P1.isConst() \quad m = P1.cex}{(E1\ \texttt{`*'}\ E2) \xrightarrow{\mathcal{T}} m * P2}$$

$$\frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \quad P2.isConst() \quad m = P2.cex}{(E1 \ \texttt{`*'} \ E2) \xrightarrow{\mathcal{T}} m * P1}$$

$$\frac{\begin{array}{l} E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \quad \neg isConst(P1) \quad \neg isConst(P2) \\ m = P1.ctx \quad M = poly(m) \quad R = \langle P1 = M \rangle \end{array}}{(E1 \ \texttt{`*'} \ E2) \xrightarrow{\mathcal{T}} m * P2} \ \mathrm{G} \leftarrow \mathrm{G} \cap \mathrm{R}$$

div* and *mod The behaviors of *div* and *mod* are generally under-specified for negative numbers. JKind adopts an interpretation of their behavior that is consistent with that of its various back-end solvers. In this interpretation, the result of the modulus operation is always positive, even for negative divisors. Under JKind, the following properties of `'div'` and `'mod'` are both theorems:

```
node main(D, N : int) returns (ok1, ok2 : bool);
let
 assert D <> 0;
 ok1 = (D*(N div D) + (N mod D)) = N;
 ok2 = N mod D >= 0;
 --%PROPERTY ok1;
 --%PROPERTY ok2;
tel;
```

Fig. 14. JKind `div` and `mod` properties

Integer division and modulus cannot be accurately modeled using rational division unless the numerator is divisible by the denominator. In this case, we simply perform the division or modulus and return the result. If not, a different approach is required. As implied by Figure 14, for a constant divisor D and appropriate values of k and m, any integer N can be expressed as in Equation 3 and Equation 4.

$$N = D * k + m \tag{3}$$

$$0 <= m <= |D| \tag{4}$$

This implies Equation 5 and Equation 6.

$$N \ \texttt{div} \ D = k \tag{5}$$

$$N \ \texttt{mod} \ D = m \tag{6}$$

So our approach to generalizing `div` and `mod` is to introduce two new variables, k and m, decompose their input according to Equation 3, constrain m a la Equation 4, and return a result using Equation 5 or Equation 6 as appropriate.

Again, we only support `div` and `mod` by constants. To enforce this we generate a global constraint on the denominator to ensure that it is always equal to its value at the counterexample.

$$\frac{\begin{array}{l} E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \\ d = P2.cex \quad D = poly(d) \\ P1 \mid d \\ R = \langle P2 = D \rangle \end{array}}{(E1 \texttt{ `div` } E2) \xrightarrow{\mathcal{T}} P1/d} \text{ G} \leftarrow \text{G} \cap \text{R}$$

$$\frac{\begin{array}{l} E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \\ d = P2[\,] \quad D = poly(d) \\ P1 \mid d \\ R = \langle P2 = D \rangle \end{array}}{(E1 \texttt{ `mod` } E2) \xrightarrow{\mathcal{T}} poly(0)} \text{ G} \leftarrow \text{G} \cap \text{R}$$

$$\frac{\begin{array}{l} E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \\ n = P1.cex \quad d = P2.cex \qquad D = poly(d) \quad D_{abs} = poly(|d|) \\ P1 \nmid d \\ k' = newVar(int, n \texttt{ div } d) \quad K = poly(k') \\ m' = newVar(int, n \texttt{ mod } d) \quad M = poly(m') \\ R = \langle P1 = d * K + M \rangle \cap \langle P2 = D \rangle \cap \langle 0 <= M \rangle \cap \langle M < D_{abs} \rangle \end{array}}{(E1 \texttt{ `div` } E2) \xrightarrow{\mathcal{T}} K} \text{ G} \leftarrow \text{G} \cap \text{R}$$

$$\frac{\begin{array}{l} E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2 \\ n = P1.cex \quad d = P2.cex \qquad D = poly(d) \quad D_{abs} = poly(|d|) \\ P1 \nmid d \\ k' = newVar(int, n \texttt{ div } d) \quad K = poly(k') \\ m' = newVar(int, n \texttt{ mod } d) \quad M = poly(m') \\ R = \langle P1 = d * K + M \rangle \cap \langle P2 = D \rangle \cap \langle 0 <= M \rangle \cap \langle M < D_{abs} \rangle \end{array}}{(E1 \texttt{ `mod` } E2) \xrightarrow{\mathcal{T}} M} \text{ G} \leftarrow \text{G} \cap \text{R}$$

***real*() and *int*()** No special processing is required to cast an integer to a real.

$$\frac{E \xrightarrow{\mathcal{T}} P}{(\texttt{`real' `('}E\texttt{`)'}) \xrightarrow{\mathcal{T}} P}$$

We treat `floor(x)`, on the other hand, as though it were (x `div` 1). Which is to say, we deconstruct x as $x = z' + q'$ where z' is an integer and q' is a real such that $0.0 \leq q' < 1.0$.

$$\frac{\begin{array}{l} E \xrightarrow{\mathcal{T}} P \\ n = P.cex \\ z' = newVar(int, n \texttt{ div } 1) \quad Z = poly(z') \\ q' = newVar(real, n \texttt{ mod } 1) \quad Q = poly(q') \\ R = \langle P = Z + Q \rangle \cap \langle 0.0 <= Q \rangle \cap \langle Q < 1.0 \rangle \end{array}}{(\texttt{`floor' `(' }E\texttt{ `)'}) \xrightarrow{\mathcal{T}} Q} \text{ G} \leftarrow \text{G} \cap \text{R}$$

Uninterpreted Functions The generalization of uninterpreted function occurrences is driven by the values of the function arguments evaluated at the counterexample and the generalized function instances generated during initialization. First the argument list to the function is generalized. The evaluation of the argument list at the counterexample is then used to index into the global function generalization map to locate the generalization specific to this specific function instance. A global invariant is then generated that constrains each of the generalized inputs to this function occurrence to be equal to the generalized inputs from the original function instance (thus inheriting the constraints on those inputs). Finally the generalized function instance value is returned as the generalization result.

$$\frac{\begin{array}{l} E_1 \ \cdots \ E_n \xrightarrow{\mathcal{T}} P[\,] \\ V[\,] = P[\,].cex \\ FGen = FGenMap[V[\,]] \\ R = \forall i : \langle P[i] = FGen.arg[i] \rangle \end{array}}{(ID \texttt{ `(' } E_1 \ \cdots \ E_n \texttt{ `)' }) \xrightarrow{\mathcal{T}} FGen.value} \text{ G} \leftarrow \text{G} \cap \text{R}$$

Addition and Subtraction The Lustre addition and subtraction expressions are interpreted as addition and subtraction of polynomials.

$$\frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`+'} \ E2) \xrightarrow{\mathcal{T}} P1 + P2} \qquad \frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`-'} \ E2) \xrightarrow{\mathcal{T}} P1 - P2}$$

$$\frac{E1 \xrightarrow{\mathcal{T}} P1}{(\ \texttt{`-'} \ E1) \xrightarrow{\mathcal{T}} (-P1)}$$

Constants Literal constants are processed based on their type. Numeric constants are translated into constant Polynomials and Boolean constants are translated into constant Trapezoids.

$$\frac{E = (INT|REAL|BOOL) \quad P = TorP(E.getValue())}{E \xrightarrow{\mathcal{T}} P}$$

Logical Operators The Lustre logical operators are interpreted as their corresponding operators over trapezoidal regions.

$$\frac{E1 \xrightarrow{\mathcal{T}} R1 \quad E2 \xrightarrow{\mathcal{T}} R2}{(E1 \ \texttt{`and'} \ E2) \xrightarrow{\mathcal{T}} R1 \cap R2} \qquad \frac{E1 \xrightarrow{\mathcal{T}} R1 \quad E2 \xrightarrow{\mathcal{T}} R2}{(E1 \ \texttt{`=>'} \ E2) \xrightarrow{\mathcal{T}} {\sim}R1 \cup R2}$$

$$\frac{E1 \xrightarrow{\mathcal{T}} R1 \quad E2 \xrightarrow{\mathcal{T}} R2}{(E1 \ \texttt{`or'} \ E2) \xrightarrow{\mathcal{T}} R1 \cup R2} \qquad \frac{E1 \xrightarrow{\mathcal{T}} R1}{(\texttt{`not'} \ E1) \xrightarrow{\mathcal{T}} {\sim}R1}$$

$$\frac{E1 \xrightarrow{\mathcal{T}} R1 \quad E2 \xrightarrow{\mathcal{T}} R2}{(E1 \ \texttt{`xor'} \ E2) \xrightarrow{\mathcal{T}} ({\sim}R1 \cap R2) \cup (R1 \cap {\sim}R2)}$$

Relational Operators The Lustre relational operators over numeric expressions become primitive linear relations over polynomials.

$$\frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`<'} \ E2) \xrightarrow{\mathcal{T}} \langle P1 < P2 \rangle} \qquad \frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`<='} \ E2) \xrightarrow{\mathcal{T}} \langle P1 <= P2 \rangle}$$

$$\frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`>'} \ E2) \xrightarrow{\mathcal{T}} \langle P1 > P2 \rangle} \qquad \frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`>='} \ E2) \xrightarrow{\mathcal{T}} \langle P1 >= P2 \rangle}$$

$$\frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`='} \ E2) \xrightarrow{\mathcal{T}} \langle P1 = P2 \rangle} \qquad \frac{E1 \xrightarrow{\mathcal{T}} P1 \quad E2 \xrightarrow{\mathcal{T}} P2}{(E1 \ \texttt{`<>'} \ E2) \xrightarrow{\mathcal{T}} {\sim} \langle P1 = P2 \rangle}$$

8 Complete Set Solution

Running the model and counterexample through the generalizer produces the result shown in Figure 15.

```
(
 (1) <= |UF_unique(1)_arg0_#10| <= (1) &&
 |UF_unique(1)=0_#9| = (0) &&
 (0) <= |UF_unique(0)_arg0_#8| < (|UF_unique(1)_arg0_#10|) &&
 |UF_unique(0)=2_#7| = (2) &&
 |UF_unique(2)_arg0_#6| = (-|UF_unique(1)_arg0_#10| - |UF_unique(0)_arg0_#8| + 3) &&
 |UF_unique(2)=1_#5| = (1) &&
 (-128) <= |UF_min()=0_#4| < (125) &&
 in[0] = (|UF_unique(1)_arg0_#10| + |UF_min()=0_#4|) &&
 in[1] = (-in[0] + 2*|UF_min()=0_#4| - |UF_unique(0)_arg0_#8| + 3) &&
 in[2] = (-in[0] - in[1] + 3*|UF_min()=0_#4| + 3)
)
```

Fig. 15. Generalized Complete Set Solution

Note that the generalization is just a conjunction of linear constraints on variables that either appear as inputs to the (unrolled) model or were generated as part of the generalization process. The `|UF..|` variables are auxiliary variables generated to generalize UF function instances or their arguments. We have two UF functions: $f(x)$ and $min()$. The auxiliary variable names are constructed to reflect the initial JKind solution for those functions. The most important thing to observe is that this generalization provides a recipe for creating new sequences for the model input (in[0], in[1], and in[2]) by selecting new values for the various variables, especially `|UF_min()=0_#4|` which is a generalization of the output of the $min()$ function. A little algebra and variable renaming reveals that the parameterized solution set specified by the generalization is as shown in Figure 16.

```
in[0] = min() + 1
in[1] = min() + 2
in[2] = min() + 0
```

Fig. 16. Simplified Generalization

Note that in the original solution $min()$ was 0. We can now see that by using the generalization we can generate a family of different "complete set" solutions by choosing different values for $min()$. Unfortunately the generalization doesn't allow us to permute the sequence. The ability to describe such behavior would require a generalization that is strictly more expressive than our current trapezoidal representation.

9 Conclusion

We have described a generalization technique for Lustre developed to support the performance needs of model-based fuzzing. Our generalization technique em-

ploys trapezoidal solution sets that can be computed with reasonable space and time bounds and then sampled efficiently even over integer domains. Our generalization techniques have been extended to support integer division, modulus, and uninterpreted functions. Finally, we demonstrated the utility of our approach on a complete set example and showed how, given a single permuted solution, we could generalize the solution to generate an entire family of such solutions.

References

1. David Greve. Trapezoidal Generalization of Boolean Circuits.
2. David Greve and Andrew Gacek. Trapezoidal Generalization over Linear Constraints.
3. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sept 1991.