## JKind Support for Inductive Datatypes

Modern symbolic state model checking algorithms rely on constraint solvers to returns answers to a series of different queries. Thus the theories that these model checkers can reason about generally depends on the support offered by the underlying constraint solvers that they use. Modern SMT solvers generally have strong support for the theory of infinite domain integers and reals. Many SMT solvers can also reason about bit-vector types as well.

AGREE can use both the JKind and Kind2 model checkers. This restricts the types of variables and constraints in AGREE contracts to be within the theories interpreted by these tools. However, we have found through our use of AGREE, and through feedback given to us by other users, that there is demand for expressing richer datatypes and constraints within AGREE. In particular, users sometimes wish to represent non-linear constraints or constraints over arrays of variables. Examples of models where users wish to represent non-linear constraints are models of non-linear controllers or components that keep track of the mean-squared error of their input signals.

Some examples of applications where users may wish to specify constraints over arrays are modeling serialization/de-serialization code between components, modeling components that process lists of sensor readings, or modeling components that process lists of desired coordinates that a vehicle is meant to navigate to.

The constraint solvers that JKind and Kind 2 use have some support for both non-linear constraints and complex datatypes. In particular, Z3 allows users to express non-linear constraints, but it has very weak guarantees about when it is able to terminate with an affirmative or negative answer to these problems. The general satisfiability problem for non-linear constraints over integers is not decidable. The known sound and complete algorithms for deciding non-linear constraints over reals has poor worst-case complexity and thus is not often implemented in modern constraint solvers.

Both Z3 and CVC4 are able to reason about constraints over array typed variables of arbitrary length. Both of these solvers can also reason about constraints over recursively defined data-types as well. In fact, recently CVC4 has added support for automatically performing induction to prove quantified constraints over expressions of variables of inductively defined types. In order to support the demand of users to reason about array types there are two possible approaches for modeling these constraints:

1. Add support in the model checkers to use the native theory of arrays in their underlying solvers.
2. Add support in the model checkers to use the theory of inductive data types to emulate array constraints.

During the third year of this process we took different approaches between Kind 2 and JKind. Kind 2 implemented support for reasoning about arbitrary length arrays. JKind implemented support for reasoning about inductively defined data types. Both model checkers allow users to specify quantified constraints over these datatypes[1]. In this section we will discuss our experience with supporting inductive datatypes, recursive functions, and quantified expressions in JKind using CVC4. Kind 2's support for arrays is discussed in Section 5. In Section 6 we discuss how we utilize Kind 2's support for arrays in AGREE.

## Implementation

To support inductive datatypes, it was first necessary to make several extensions to the grammar of the JKind input language. Once this was done, there were a number of further challenges that we will discuss.

## Extensions to Lustre Grammar

In order to model recursive functions and inductive datatypes in JKind we took advantage of the new features added to CVC4 to perform induction directly in SMT solvers. Specifically, we use the new `define-fun-rec` and `declare-datatypes` constructs in the SMT-LIB standard. We support a new Lustre grammar element called a "recursive node" that has syntax similar to a traditional Lustre node:

```
recursive:
  'recursive' ID '(' input=varDeclList? ')'
  'returns' '(' output=varDeclList? ')' ';'
  'let'
    (equation)+
  'tel' ';'?
;
```

The main differences between a recursive node and a Lustre node are as follows:

1. A recursive node may include a recursive call to itself in its body. It may also contain a transitively recursive call to itself by calling another recursive node.
2. A recursive node cannot contain any local variables[2].
3. A recursive node can only contain a single output.
4. No temporal operations (`pre` or `->`) are allowed in a recursive node.

Traditional Lustre nodes can only contain a finite number of state variables. They can reference other Lustre nodes, but they cannot contain transitive calls to themselves. These restrictions

[1] Kind2 allows the domain of discourse for quantifiers to only be integers. The support we have added into JKind allows users quantify over any inductively defined type as well as the integer and real types.

[2] We could add this feature later if we would like, but it would increase the complexity of the translation to SMT-LIB

make it easy to inline a Lustre node using relatively simple program transformations. In JKind all nodes called inside of the main program node are inlined before the program is translated into a transition relation described in SMT-LIB. JKind does not inline recursive nodes. Instead the tool directly translates a recursive node definition into a `define-fun-rec` expression in the SMT-LIB queries sent to CVC4. This is why we require restrictions 2-4 described above.

In addition to recursive nodes, we have also added support to define datatypes inductively. This is described by the following excerpt from JKind's grammar:

```
topLevelType: type
    | 'struct' '{' (ID ':' type) (';' ID ':' type)* '}'
    | 'enum' '{' ID (',' ID)* '}'
    | 'induct' '{' inductTerm ('|' inductTerm)* '}'
    ;

inductTerm:
  ID ('(' ID type ')')*
```

Inductive datatype definitions are directly mapped to `declare-datatypes` expressions in SMT-LIB. In Section 4.3 we give some examples of Lustre programs that take advantage of these features. One of these examples models some properties of a hypothetical avionics system consisting of a controller component and a planner component. The planner component sends lists of waypoints to the controller component which decides where the system will navigate to next. The list of waypoints is defined using the new grammar element for inductive datatypes:

```
type Coord = induct {point (x int) (y int)};
type CoordLst = induct {cons (head Coord) (tail CoordLst) | nil };
```

In this example, a recursive function is defined that returns true if and only if a given coordinate is the last element of a given list of coordinates:

```
recursive planEndsInCoord(
  plan : CoordLst;
  dest : Coord)
returns(
  ret : bool);
let
  ret = if (is_nil plan) then
          false
        else if (is_nil (tail plan)) then
          (head plan) = dest
        else
          (planEndsInCoord (tail plan) dest);
tel;
```

In the examples that we have created we generally use recursive functions as predicates to formalize properties. For example, one of the properties in the geofence example we use the

function defined above to express the property: “If we have not received a new geofence, then the current plan in the controller contains only waypoints inside the geofence stored in the mission computer”:

```
not gotFence => (planInFence misFence ctrPlan);
```

In Section 4.3 we describe this example in more detail.

## Specific Challenges

It is generally pretty straightforward to support a new SMT-Solver in JKind as long as the solver properly implements the SMT-LIB standard. We already included support for CVC4 in JKind, but we had to make a few rather invasive changes to the tool to support these new features. These changes were due to a few different design decisions for these new features in CVC4.

The first problem is that CVC4 requires different flags to in order to possibly return satisfiable or unsatisfiable results for certain problems containing quantification. When the solver operates on constraints containing mixed quantifiers and/or recursive functions it uses a finite model finding algorithm to attempt to produce a counterexample. Similarly, it uses a different algorithm to produce unsatisfiable results. The solver requires that the user supply different flags to the tool to tell it which one of these algorithms it should perform. Therefore, JKind needs to call two separate instances of CVC4 with different flags simultaneously in order to determine whether or not a specific query is satisfiable or unsatisfiable[3].

Another problem that we ran into was that in some examples JKind’s BMC process required results from its inductive process in order to make progress on other properties. To illustrate this consider the main node of a Lustre program shown below:

```
node main(n : Nat) returns();
var
  l : Lst;
  odd_in_past : bool;
  prop0 : bool;
  prop1 : bool;
  prop2 : bool;
  prop3 : bool;
  prop4 : bool;
let
  l = (cons n nil) -> (cons n (pre l));
  odd_in_past = (odd n) or (false -> pre(odd_in_past));

  prop0 = forall (m : Nat) . (even (suc m)) = not (even m);
  prop1 = forall (m : Nat) . (odd (suc m)) = not (odd m);
```

[3] Ideally the solver would internally try to solve a query using both the finite model finding algorithm and the induction algorithm in parallel. We do not think that the onus should be on the model checker to figure out what flags the solver should be called with. We believe that he desire to perform well on certain metrics in solver competitions might be harming the general usability of some academic solvers.

```
  prop2 = forall (m : Nat) . (odd m) = (not (even m));

  prop3 = forall (m : Nat) . (in m l) => (odd m) => odd_in_past;
  prop4 = forall (m : Nat) . (in m l) => (not (even m)) => odd_in_past;

  --%PROPERTY prop0;
  --%PROPERTY prop1;
  --%PROPERTY prop2;
  --%PROPERTY prop3;
  --%PROPERTY prop4;

tel;
```

The program has a single natural number input. At every time instance the program places the current value of its input into a list. It has keeps track of whether or not any of the inputs it received in the past were odd. The fifth property (`prop4`) states that if any of the numbers in the list are not even, then the program received an odd input in the past. To prove this property a few lemmas about the relationship between the odd and even recursive functions must first be proved. These lemmas are the first three properties. Another property about, which is almost identical to the fifth property, must also be proved. This is the fourth property.

When the BMC engine tries to falsify the fifth property it must prove that it is impossible to violate this property within a single step from the initial states. However, for this example CVC4 is not able to return an unsatisfiable result to this query unless it has first proved that if an odd input was received in the initial state, then an odd input was received in the past.

In order to make progress JKind solves each property individually in the order in which they appear in the program. If a property is proved true than it is asserted in the BMC queries sent to the solver so it can may be more likely to make progress on other properties.

## 1.1 Examples

In this section we discuss two of the examples that we created to exercise the new features supporting expressions of quantified inductive data types and recursive functions in JKind. These two examples were created based on specific problems that we wanted to solve with our tools on other projects.

### 1.1.1 List filtering

The following example illustrates how one could perform various “filtering” operations on lists and then prove that their results meet some well-formedness criteria.

```
type Lst = induct {cons (head int) (tail Lst) | nil };

recursive filter(lst : Lst; high : int; low : int) returns (ret : Lst);
let
  ret = if (is_nil lst) then
```

```
          nil
        else
          if (head lst) >= high or (head lst) <= low then
            (filter (tail lst) high low)
          else
            (cons (head lst) (filter (tail lst) high low));
tel;

recursive contains(x : int; lst : Lst) returns (ret : bool);
let
  ret = if (is_nil lst) then
          false
        else if (head lst) = x then
          true
        else
          (contains x (tail lst));
tel;

recursive increase(x : int; lst : Lst) returns (ret : Lst);
let
  ret = if (is_nil lst) then
          nil
        else
          (cons ((head lst) + x) (increase x (tail lst)));
tel;

recursive some_neg(lst : Lst) returns (ret : bool);
let
  ret = if (is_nil lst) then
          false
        else if (head lst) < 0 then
          true
        else
          (some_neg (tail lst));
tel;

recursive sum(lst : Lst) returns (ret : int);
let
  ret = if (is_nil lst) then
          0
        else
          (head lst) + (sum (tail lst));
tel;

node main(input : int) returns ();
var
  prop0 : bool;
  prop1 : bool;
  prop2 : bool;
  prop3 : bool;
  list : Lst;
  prop4 : bool;
let
  list = if (input >= 10) or (input <= -5) then nil -> pre(list) else (cons
input nil) -> (cons input pre(list));

  prop0 = forall (x : int) .
```

```
     (x >= 10) or (x <= -10) => not (contains x list);

  Prop1 = forall (x : int; lst : Lst; high : int; low : int) .
    (x >= high) or (x <= low) => not (contains x (filter lst high low));

  Prop2 = forall(x : int; y : int; lst : Lst; high : int; low : int) .
    (x >= high + y) or (x <= low + y) => not (contains x (increase y (filter
lst high low)));

  Prop3 = forall(x : int; high : int; low : int; lst : Lst) .
    (x >= high) or (x <= low) => not (contains x (filter (cons x lst) high
low));

  Prop4 = forall(l : Lst) .
    (sum l) < 0 => (some_neg l);

  --%PROPERTY prop0;
  --%PROPERTY prop1;
  --%PROPERTY prop2;
  --%PROPERTY prop3;
  --%PROPERTY prop4;
tel;
```

The recursive node filter receives as input a list and two integers representing a high and low bound. The function produces a list that only contains elements from the original list that are between the high and low bound. The main node of the program has a single integer input. It also contains a variable of type `Lst` that contains all previous inputs whose values range between -5 and 10. The first property of the program asserts that no value greater than or equal to 10 or less than or equal to -10 are contained in the list at any time. The remaining properties prove more general invariants of the filtering function over all lists. For example, the third property states that if you were to increase every value of a filtered list by some arbitrary amount, then no item in the list that is outside of the filtered range plus this amount is contained in the list.

These types of properties are of interest for applications that perform some sort of processing on lists of data. For example, performing some sort of signal processing on lists of sensor readings or transforming lists of coordinates from one type of units to another.

## Geofencing

The following Lustre program models an avionics system containing two components: a “Mission Planner” and a “Controller.” The system contains two inputs: a “fence” and a “destination.” The role of the mission planner is to generate a “plan” which is a set of waypoints within the last fence received by the mission planner. The final waypoint of the plan is the last destination received by the mission planner.

```
type Coord = induct {point (x int) (y int)};
type CoordLst = induct {cons (head Coord) (tail CoordLst) | nil };
type Fence = induct {corners (top int) (bottom int) (left int) (right int)};
```

```
recursive coordInFence(
  fence : Fence;
  coord : Coord)
returns(
  ret: bool);
let
  ret = if (x coord) <= (right fence) and
            (x coord) >= (left fence) and
            (y coord) <= (top fence) and
            (y coord) >= (bottom fence) then
              true
        else
             false;
tel;

recursive planEndsInCoord(
  plan : CoordLst;
  dest : Coord)
returns(
  ret : bool);
let
  ret = if (is_nil plan) then
          false
        else if (is_nil (tail plan)) then
          (head plan) = dest
        else
          (planEndsInCoord (tail plan) dest);
tel;

recursive planInFence(
  fence : Fence;
  coords : CoordLst)
returns(
  ret: bool);
let
  ret = if (is_nil coords) then
          true
        else
          (coordInFence fence (head coords)) and
          (planInFence fence (tail coords));
tel;

node abs (x : int) returns (ret : int);
let
  ret = if x < 0 then
          -1 * x
        else
          x;
tel;

node min(x : int; y : int) returns (ret : int);
let
  ret = if (x < y) then x else y;
tel;

node main(
```

```
  dest : Coord;
  fence : Fence;
  misPlan : CoordLst;
  location: Coord;
  gotFence : bool;
  gotDest : bool;
  newPlan : bool)
returns ();
var
  prop0 : bool;
  prop1 : bool;
  prop2 : bool;
  prop3 : bool;
  lemma0 : bool;
  misFence : Fence;
  misDest : Coord;
  locInMisFence : bool;
  nextPoint : Coord;
  ctrPlan : CoordLst;
  curXDist : int;
  preXDist : int;
  curYDist : int;
  preYDist : int;
  nextPointInMisFence : bool;
  xDistToFence : int;
  yDistToFence : int;
let
  --BEGIN MISSION PLANNER VARIABLES
  misFence = if (gotFence) then fence else ((corners 0 0 0 0) ->
pre(misFence));
  misDest = if (gotDest) then dest else ((point 0 0) -> pre(misDest));

  --BEGIN CONTROLLER VARIABLES
  ctrPlan = nil ->
                  (if pre(newPlan) then
                    pre(misPlan)
                  else if (is_cons (pre ctrPlan)) and
                          pre(location) = (head (pre ctrPlan)) then
                    (tail (pre ctrPlan))
                  else
                    pre(ctrPlan));


  nextPoint = (point 0 0) ->
              if (is_nil ctrPlan) then
                pre(location)
              else
                (head ctrPlan);

  --BEGIN MISSION PLANNER GUARANTEES

  --The mission planner guarantees that it always provides a plan
  --that falls within the current geofence

  assert (planInFence misFence misPlan);

  --if we got a new fence we need a new plan
```

```
  assert gotFence => newPlan;

  --if we got a new destination we need a new plan

  assert gotDest => newPlan;

  --We never send a blank plan

  assert (newPlan => not (is_nil misPlan));

  --If we have a plan it always has the distination in it

  assert not (is_nil misPlan) => (planEndsInCoord misPlan misDest);

  --BEGIN CONTROLLER GUARANTEES

  --the controller is always initially at point 0,0
  assert( (location = (point 0 0)) -> true);

  --The controller guarantees that it always stays within the same distance
  --or moves towards its next coordinate

  curXDist = 0 -> (x location) - (x pre(nextPoint));
  preXDist = 0 -> (x pre(location)) - (x pre(nextPoint));

  curYDist = 0 -> (y location) - (y pre(nextPoint));
  preYDist = 0 -> (y pre(location)) - (y pre(nextPoint));

  assert preXDist <= 0 => curXDist <= 0 and curXDist >= preXDist;
  assert preYDist <= 0 => curYDist <= 0 and curYDist >= preYDist;

  assert preXDist >= 0 => curXDist >= 0 and curXDist <= preXDist;
  assert preYDist >= 0 => curYDist >= 0 and curYDist <= preYDist;

  --BEGIN SYSTEM ASSUMPTIONS

  --We assume that if we get a new geofence then the current
  --destination lies within this fence

  assert(gotFence => (coordInFence fence misDest));

  --BEGIN SYSTEM VARIABLES

  locInMisFence = (coordInFence misFence location);
  nextPointInMisFence = (coordInFence misFence nextPoint);

  --BEGIN PROPERTIES

  --if we are not in the fence then:
  --1. The fence is new.
  --2. We were previously not in the fence.

  lemma0 = true -> (is_nil ctrPlan) => location = pre(location);
  prop0 = true -> pre(locInMisFence and not gotFence) => locInMisFence or
gotFence;
```

```
  --The controller's plan is always within the mission computer's fence
  --unless we just received a new fence

  prop1 = not gotFence => (planInFence misFence ctrPlan);

  --The destination is at the end of the plan (if we did not just get a new
one)

  prop2 = not gotDest => (not (is_nil ctrPlan)) => (planEndsInCoord ctrPlan
misDest);

  --If we are outside the fence, then we do not move away from the fence

  xDistToFence = (min( abs((x location) - (left misFence)), abs((x location)
- (right misFence))));
  yDistToFence = (min( abs((y location) - (top misFence)), abs((y location) -
(bottom misFence))));

  prop3 = true -> not (pre(locInMisFence or gotFence) or gotFence) =>
    (locInMisFence or
    (xDistToFence <= pre(xDistToFence) or yDistToFence <=
pre(yDistToFence)));

  --%PROPERTY lemma0;
  --%PROPERTY prop0;
  --%PROPERTY prop1;
  --%PROPERTY prop2;
  --%PROPERTY prop3;

tel;
```

The controller guarantees that the location given by the controller either remains in the same location or moves towards the next coordinate in the given plan. The plan is modeled as a list of x,y coordinates of arbitrary length, and the fence is modeled as a rectangle. The program models the following properties:

1. If the previous location was within the fence, then the next location is in the fence unless the system recently received a new fence
2. The plan being executed by the controller is always within the most recent fence received by the mission computer, except in the instant that a new fence is received.
3. The mission computer's last received destination is the last coordinate in the plan being executed by the controller, except in the instant that a new destination is received.
4. If the location is outside of the fence and no new fence was received recently, then the Manhattan distance from the perimeter of the fence either remains the same or decreases.