Dynamic logic of questions
Minica, S.A.

Citation for published version (APA):
Chapter 7
Implementing Querying Strategies

7.1 Implementation for Questioning Actions

We already mentioned two implementations that were used to model NE_p in LG_l: Haskell and Alloy Analyzer. Haskell offers the advantages and versatility of functional programming [25] and Alloy Analyzer [52] was used for characterizing NE_p in LG_l. In this section we present and discuss further details about these implementations. We will show how the Alloy code can be used to define relevant properties of game profiles. And we will illustrate the Haskell implementation for querying local properties of strategy profiles.

A final important aspect in designing good questioning strategies under uncertainty is an account of probabilities. In the final part of this section we present and document the implementation behind probabilistic DELq. One essential element in this extension is an adequate notion of behavioral equivalence and an algorithm for minimizing probabilistic models, defined in the previous chapter.

The connection between questioning theory and implementation tools has proved once more to be a fruitful one.

7.2 Implementation and Illustrative Examples

In this section we will briefly describe the Haskell implementation for local game properties as list comprehension, explain its main functionality and use the code to produce some illustrative examples.

7.2.1 Haskell Implementation

We start by importing basic Haskell functionality for list manipulation, line 3, and some further functionality for generating various combinatorial functions, line 2.

For instance, all possible strategy profiles in the location game can be generated by taking the cartesian product, line 6, of as many lists of all available locations as there are players in the location game.
Chapter 7. Implementing Querying Strategies

module LocGame where
import CombinatoricsGeneration
import List

allgames p l = cartProd \$ take p \$ repeat \[1..l\]
cartProd (set:sets) =
  let cp = cartProd sets in \[x:xs | x <- set, xs <- cp\]
cartProd \[] = \[]
cangames p l = \[x | x <- allgames p l, y<-[1..p-2],
x!!y >= (maximum (take (y) x)), x!!y <= (minimum (drop (y+1) x))\]

As already explained in the text, the main task is to generate a search space that contains only relevant profiles while still allowing for an efficient way of finding Nash equilibria for the location game.

The next block of code gives the implementation of the main four local properties discussed as list comprehension over the canonical profiles.

p1games p l = nub \[x | x <- cangames p l, minimum (map
  \(\langle y -> (payf x y (fromIntegral l) (fromIntegral p))\)
  \[0..(fromIntegral p)-1\]) >= 1 \]
p2games p l = nub \[x | x <- cangames p l, maximum
  (map \(\langle y -> (hght x y)\) \[0..1\]) <=2 \]
p3games p l = nub \[x | x <- cangames p l, maximum
  (map \(\langle y -> (lftd x y (fromIntegral l))\)
  \[0..(fromIntegral p)-1\]) <=1, maximum
  (map \(\langle y -> (rghd x y (fromIntegral l))\)
  \[0..(fromIntegral p)-1\]) <=1 \]
p4games p l = nub \[x | x <- cangames p l,
dycg x (fromIntegral l) (fromIntegral p) == True \]

The rest of the code computes the payoff following the previously explained definitions.

As an illustration of how this implementation approach works we present a representative example in which local properties are used to restrict the relevant search space for Nash equilibria profiles. The example also illustrates the fact that a Nash equilibrium for pure strategies is not always guaranteed to exist in the location game on a line.

*LocGame> cangames 3 5
\[[1,1,1],[1,1,2],[1,1,3],[1,1,4],[1,1,5],[1,2,2],[1,2,3],[1,2,4],[1,2,5],[1,3,3],[1,3,4],[1,3,5],[1,4,4],[1,4,5],[1,5,5],[2,2,2],[2,2,3],[2,2,4],[2,2,5],[2,3,3],[2,3,4],[2,3,5],[2,4,4],[2,4,5],[2,5,5],[3,3,3],[3,3,4],[3,3,5],[3,4,4],[3,4,5],[3,5,5],[4,4,4],[4,4,5],[4,5,5],[5,5,5]\]

*LocGame> p1games 3 5
\[[1,1,1],[1,1,2],[1,1,3],[1,1,4],[1,1,5],[1,2,2],[1,2,3],[1,2,4],[1,2,5],[1,3,3],[1,3,4],[1,3,5],[1,4,4],[1,4,5],[1,5,5],[2,2,2],[2,2,3],[2,2,4],[2,2,5],[2,3,3],[2,3,4],[2,3,5],[2,4,4],[2,4,5],[2,5,5],[3,3,3],[3,3,4],[3,3,5],[3,4,4],[3,4,5],[3,5,5],[4,4,4],[4,4,5],[4,5,5],[5,5,5]\]
We end this section with additional relevant code output. This includes examples of representative profiles satisfying P1 to P4 properties for the following locations and players values \( l = p + 2, l = p + 3, \) and in general \( p < l:\)

\[
\begin{align*}
&[2,3,4], [2,3,5], [2,4,4], [2,4,5], [2,5,5], [3,3,3], [3,3,4], [3,3,5], [3,4,4], [3,4,5], [4,4,4], [4,4,5], [5,5,5])
\end{align*}
\]

\*LocGame> p2games 3 5  
\([1,1,2], [1,1,3], [1,1,4], [1,1,5], [1,2,2], [1,2,3], [1,2,4], [1,2,5], [1,3,3], [1,3,4], [1,3,5], [1,4,4], [1,4,5], [1,5,5], [2,2,3], [2,2,4], [2,2,5], [2,3,3], [2,3,4], [2,3,5], [2,4,4], [2,4,5], [2,5,5], [3,3,3], [3,3,4], [3,3,5], [3,4,4], [3,4,5], [3,5,5], [4,4,5], [4,5,5])

\*LocGame> p3games 3 5  
\([1,2,4], [1,3,4], [1,3,5], [2,2,4], [2,3,4], [2,3,5], [2,4,4], [2,4,5])

\*LocGame> p4games 3 5  
\([]\)

As discussed in the main text, using local properties allows for faster and correct searching strategies using queries to an oracle, but without breaking the threshold of efficiency. An efficient solution also will have to use a characterization of Nash equilibrium by local properties but additionally will have to make essential use of fragments of strategy profiles.
7.2.2 Alloy Analyzer Implementation

We proceed now towards presenting the Alloy Analyzer implementation that provides a characterization of Nash equilibrium by local properties and tests this by logical entailment in a specified scope.

We start by importing predefined Alloy modules for arithmetic operations over integer numbers line 1, used for computing payoff values, and predefined operations over orders, used to compare location signatures, line 2, and player signatures, line 3, in a strategy profile.

Next we introduce the basic signatures needed to represent and reason about location games. The signatures for locations, line 5, respectively for players, line 7, are atomic signatures without component fields representing the fact that they do not have any relevant internal structure.

The game signature, lines 9 to 13 has some internal structure represented by three fields: a sequence of locations, a sequence of players and a function from players to locations encoding choices made in the game, for each player.

The local properties defined and studied in the main text are going to be introduced as facts over the previously described game signatures. The first property, line 15, captures the canonical profiles. Any game is isomorphic up to permutations of players to some canonical game. The constraints over game signatures that are implementing this condition make use of the order relation over players and their choices represented by the Locations field. Intuitively this says that the player’s choices are ordered weakly increasing.

The following blocks of code are dedicated to game constraints implementing the local properties P1 to P4. For the sake of conceptual clarity we split them in two clusters. The first will contain static properties which describe the characteristics of a favorable strategy profile instance, and the second one will capture
dynamic properties or legal ways in which a profile can behave in terms of changes in the field representing choice functions.

The first fact, line 20, implements property P1. Intuitively, this requires that in any game signature any player has a payoff greater than or equal with a unit. This relies on a definition of the payoff values in the game as fractional numbers which will be introduced later. Because the payoff can be a fractional number and Alloy only has predefined functionality for integers the payoff comparisons will require utilities which are slightly more elaborate.

```
20 fact { --P1 : Minimum Payoff
21   all g : Game | all x : Player |
22   payoff_gt_3d[payoff_n_tot[g,x],1,payoff_d_tot[g,x],1]
23   or (div[payoff_n_tot[g,x],payoff_d_tot[g,x]] = 1 &&
24     rem[payoff_n_tot[g,x],payoff_d_tot[g,x]] = 0)
25 }
26
27 fact p2 { --P2 : Maximum Height
28   all g : Game | all i,j,k : Player |
29     i != j && i != k && j != k && g.Choice[i] == g.Choice[j] =>
30     g.Choice[k] != g.Choice[j]
31 }
32
33 fact p3 { --P3 : Maximum Distance
34   all g : Game | all x,y : g.Locations.inds -
35     {x : Int | some y : Location | some z : Player |
36     g.Locations.idxOf[y] == x && g.Choice[z] = y} | x != y-1
37 }
```

The second fact, line 27, implements property P2. Intuitively, this requires that never more than two players occupy the same location in a game profile. This is a straightforward translation of the first order formula expressing the tower height property. Only quantification over players and their choices together with equality and inequality are needed in order to express this fact.

The third fact, line 33, implements property P3. Intuitively, this says that the maximum distance between occupied locations in a game is strictly smaller than two. This is not a straightforward translation form the first order formula expressing this property. Although such a translation is possible it is much easier encoded as a property of locations. For this the indexes of locations in the sequence are used to perform simple arithmetic on their values. First the empty locations in a game are selected by means of a set comprehension expression. Next the condition that no difference between any two numbers in this set is one is imposed. Here and in other places later on it is important to have index values as integers in order to perform basic arithmetical operations, the ordering of locations alone would not be enough for this.

```
39 fact p4 { --P4 : Proximal Move
40   all x,y : Game | all z : Player |
41   not udev [x, y, z]
```

The third fact, line 33, implements property P3. Intuitively, this says that the maximum distance between occupied locations in a game is strictly smaller than two. This is not a straightforward translation form the first order formula expressing this property. Although such a translation is possible it is much easier encoded as a property of locations. For this the indexes of locations in the sequence are used to perform simple arithmetic on their values. First the empty locations in a game are selected by means of a set comprehension expression. Next the condition that no difference between any two numbers in this set is one is imposed. Here and in other places later on it is important to have index values as integers in order to perform basic arithmetical operations, the ordering of locations alone would not be enough for this.
The next fact, line 39, implements property P4. This is the only property describing the dynamics of the game. Intuitively, it says that no player has an incentive for proximal move or unilateral deviation with one unit from his choice in the considered strategy profile. This being a dynamic property, is implemented in Alloy Analyzer in declarative style, by a predicate taking as input the precondition game, the postcondition game and the relevant player. The following three predicates provide an implementation for this as follows:

\[
\begin{align*}
\text{pred devleft } [g,g', p : \text{Player}] & \{ \\
g.\text{Locations} == g'.\text{Locations} \\
g.\text{Players} == g'.\text{Players} \\
g'.\text{Choice}[p] == \text{prev}[g.\text{Choice}[p]] \\
\text{all } x : \text{Player} & | p != x \Rightarrow g'.\text{Choice}[x] == g.\text{Choice}[x] \\
\text{payoff}_\text{gt}_3d[\text{payoff}_\text{ntot}[g', p], \text{payoff}_\text{ntot}[g, p], \\
\text{payoff}_\text{dtot}[g', p], \text{payoff}_\text{dtot}[g, p]] \\
\} \\
\text{pred devright } [g,g', p : \text{Player}] & \{ \\
g.\text{Locations} == g'.\text{Locations} \\
g.\text{Players} == g'.\text{Players} \\
g'.\text{Choice}[p] == \text{next}[g.\text{Choice}[p]] \\
\text{all } x : \text{Player} & | p != x \Rightarrow g'.\text{Choice}[x] == g.\text{Choice}[x] \\
\text{payoff}_\text{gt}_3d[\text{payoff}_\text{ntot}[g', p], \text{payoff}_\text{ntot}[g, p], \\
\text{payoff}_\text{dtot}[g', p], \text{payoff}_\text{dtot}[g, p]] \\
\} \\
\text{pred udev } [g,g', p : \text{Player}] & \{ \\
\text{devleft } [g,g', p] \text{ or devright } [g,g', p] \\
\}
\end{align*}
\]

The predicate at line 44 defines a left unit unilateral deviation with incentive. It specifies the fact that the Locations and Players fields in the pre and post game signatures remain unchanged and the fact that the choice of the player given as a parameter is changed from its initial location to the previous one in the order, line 47. In the same time, this change is unilateral, that is, all the other choices in the strategy profile remain unchanged, line 48.

The predicate at line 53 defines a right unit unilateral deviation with incentive. It specifies the fact that the Locations and Players fields in the pre and post game signatures remain unchanged and the fact that the choice of the player given as a parameter is changed from its initial location to its successor in the order, line 56. In the same time, this change is unilateral, that is, all the other choices in the strategy profile remain unchanged, line 57.

An additional condition in both predicates is that the change comes with an incentive, that is the payoff value in the postgame is greater than the payoff value in the pregame, lines 49 and 58.

The last predicate in the block, line 62, puts together the previous ones to
7.2. Implementation and Illustrative Examples

define a left or right unit unilateral deviation with incentive. This is afterward
directly used to define the P4 property.

```haskell
fun payoff_n_tot [g : Game, i : Player] : Int {
  let th = tower_height[g, where[g, i]],
  thl = tower_height_left_limit[g, i],
  thr = tower_height_right_limit[g, i],
  dl = left_dis_int_half[g, where[g, i]],
  dr = right_dis_int_half[g, where[g, i]],
  fl = rem[left_dis_int[g, where[g, i]],2],
  fr = rem[right_dis_int[g, where[g, i]],2] |
  let oone = add[1,add[dl,dr]] | 
    #left_of_occ[g,where[g,i]] != 0 &&
    #right_of_occ[g,where[g,i]] != 0 =>
    add[ mul[oone, mul[add[th,thl],add[th,thr]]],
      add[mul[fl,mul[th,thr]],mul[fr,mul[th,thl]]]]
  else
    (#left_of_occ[g,where[g,i]] == 0 =>
    add[ mul[(1+left_dis_int[g,where[g,i]]+dr),add[th,thr]],
      mul[fr,th]]
    else
      (#right_of_occ[g,where[g,i]] == 0 =>
    add[ mul[(1+right_dis_int[g,where[g,i]]+dl),add[th,thl]],
      mul[fl,th]]
    else
      add[1, add[dl,dr]]
  )
}
```

The definition of both local properties P1 and P4 have made use of the payoff
values for specific choices in the game. So we have to introduce the way in which
this value is computed. Due to the fact that Alloy Analyzer is designed to handle
only integer numeric types and the payoff is a fractional number we will have to
represent the payoff value as a pair of integers.

The function at line 66 takes a game signature and a player and returns the
integer representing the numerator of the fraction that computes the payoff value
for the given player in the given game. The computation proceeds according to
the game definition by considering four relevant cases and using the splitting of
location payoff units according to the distances on the line and number of stacked
players and their proximal neighbors.

The function at line 93 takes a game signature and a player and returns the
integer representing the denominator of the fraction that computes the payoff value
for the given player in the given game. Again the computation proceeds
according to the game definition by considering four relevant cases and using the
tower heights of the relevant locations in the game.

```haskell
fun payoff_d_tot [g : Game, i : Player] : Int {
  let th = tower_height[g, where[g, i]],
```
Chapter 7. Implementing Querying Strategies

\[
\text{thl} = \text{tower\_height\_left\_limit}[g,i], \\
\text{thr} = \text{tower\_height\_right\_limit}[g,i] | \\
\text{#left\_of\_occ}[g,\text{where}[g,i]] != 0 \&\& \text{#right\_of\_occ}[g,\text{where}[g,i]] != 0 \\
\Rightarrow \text{mul}[\text{th}, \text{mul}\{\text{add}[\text{th},\text{thl}], \text{add}[\text{th},\text{thr}]\}]
\]

\[
\text{else} \\
\text{(#left\_of\_occ}[g,\text{where}[g,i]] == 0 \Rightarrow \\
\text{mul}[\text{th}, \text{add}[\text{th},\text{thr}] \} \\
\text{else} \\
\text{(#right\_of\_occ}[g,\text{where}[g,i]] == 0 \Rightarrow \\
\text{mul}[\text{th}, \text{add}[\text{th},\text{thl}] \} \\
\text{else} \\
\text{th}
\]

Computing the fractional payoff value is only one aspect of analyzing the game from a static perspective. Another important aspect is comparison between payoff values, this can analyze the game from a dynamic perspective.

The function at line 110 takes four integers as arguments, representing, in order, the nominator of the first fraction, the nominator of the second fraction, the denominator of the first fraction and the denominator of the second fraction, and returns true if the first fraction is greater then the second. The fractions are representing payoff values and the comparison proceeds up to the third decimal value. Note that for a maximum column height of two the first two decimals should already be enough in order to decide the highest value.

The computation uses predefined alloy functionality for integer arithmetic like multiplication, integer division and division reminder. These were imported in the beginning from the util/integer module and are used to encode the result of the fractional division as a rational number. This is needed because Alloy does not have a predefined data structure for floating numbers.

For instance, when comparing 500 350 with 500 351 the result will be 1.42857 > 1.42450 and will be obtained by considering the first three decimal positions. This provides enough precision to be meaningful in the context of the location game.
7.2. Implementation and Illustrative Examples

The following code blocks are dedicated to various computation for payoff value. There is nothing remarkable about the functions, they proceed as expected to capture the game definition. We include them here for the sake of completeness but we will only briefly describe them without too much details.

```haskell
fun where [g : Game, p : Player] : Location {
  g.Choice[p]
}

fun tower_height [g : Game, l : Location] : Int {
  #{x : Player | g.Choice[x] = l}
}
```

The function at line 126 takes a player and a game and returns its chosen location. At line 130, the function takes a location and a game signature and returns the number of players that have chosen the location. The function at line 134 takes a player and a game and returns the tower height of its right limit, similarly, line 139, takes a player and returns the tower height of its left limit.

```haskell
fun tower_height_right_limit [g : Game, i : Player] : Int {
  where[g,i] = right_limit[g, where[g,i]] => 0
  else tower_height[g, right_limit[g, where[g,i]]]
}

fun tower_height_left_limit [g : Game, i : Player] : Int {
  where[g,i] = left_limit[g, where[g,i]] => 0
  else tower_height[g, left_limit[g, where[g,i]]]
}
```

Next function, at line 144, takes a location and returns half of its left distance as an integer number, similarly, line 148, takes a location and a game and returns half of its right distance as an integer number.

```haskell
fun left_dis_int_half [g : Game, l : Location] : Int {
  div[ left_dis_int[g, l],2 ]
}

fun right_dis_int_half [g : Game, l : Location] : Int {
  div[ left_dis_int[g, l],2 ]
}
```

The entire left distance is computed at line 152, the function takes a location and a game and returns its left distance as a set of locations including limit locations. Analogously, line 158, takes a location and returns its right distance as set of locations including limits.

```haskell
fun left_dis [g : Game, l : Location] : Location {
  no left_of_occ[g, l] => { x : Location | gt[x, left_limit[g, l]]
    && lt[x,l] } + left_limit[g, l]
}
```
Chapter 7. Implementing Querying Strategies

In some contexts it is useful to consider the distance without the ending locations. The function at line 164 takes a location and returns its left distance as integer excluding limit locations. Analogously for the right side, the function at line 168 takes a location and returns its right distance as integer again by excluding the limit locations.

In the case of a location game on a line there are two locations with a special behavior because they are the line endpoints. The function at line 172 takes a location and returns its left limit or the least location i.e. the left endpoint of the line or the minimal location. Symmetrically, the function at line 177, takes as input a location and a game and returns its right limit or the greatest location, i.e. the right endpoint of the line or the maximal location.

Another auxiliary function, line 182, takes a location and returns the occupied locations to its left. Symmetrically, at line 186, the function takes a location and returns the occupied locations to its right. Analogously the functions at lines 190 and 194 take as input the signatures of a location and a game and return the free locations to the left and to the right, respectively.
7.2. Implementation and Illustrative Examples

fun right_of_occ [g : Game, l : Location] : Location {
    { x : Location | gt[x,l] && some {y : Player | g.Choice[y] = x} }
}

fun left_of_free [g : Game, l : Location] : Location {
    { x : Location | lt[x,l] && no {y : Player | g.Choice[y] = x} }
}

fun right_of_free [g : Game, l : Location] : Location {
    { x : Location | lt[x,l] && no {y : Player | g.Choice[y] = x} }
}

The final predicate, line 198, used to capture the game dynamics records arbitrary unilateral deviation with incentive. This is completely analogous to the predicate characterizing the P4 property.

The difference is that this time the change of choice or the deviation in location is an arbitrary one, it is not restricted to proximal locations but can be any location in the game. This allows the predicate to be used in the characterization of a Nash equilibrium strategy profile.

pred ne [g,g' : Game, p : Player] {
    g.Locations == g'.Locations
    g.Players == g'.Players
    g'.Choice[p] != g.Choice[p]
    all x : Player | p != x => g'.Choice[x] == g.Choice[x]
    payoff_gt_3d[payoff_n_tot [g', p], payoff_n_tot [g, p],
        payoff_d_tot [g', p], payoff_d_tot [g, p]]
}

Based on this characterization the entailment between the local properties and Nash equilibrium can be tested using a finite scope in Alloy Analyzer.

There are some additional constraints that do not correspond directly to the local properties discussed so far but are needed in order to make the signatures behave in the desired way. We include them here for the sake of completeness.

The first one, line 207, requires that players and locations are always part, i.e. fields, in a game. The second one, line 210, requires that games always have the same component fields. This is useful for comparing games.

fact{
    all x : Player | some g : Game | x in g.Players.elems
    all x : Location | some g : Game | x in g.Locations.elems
    all x, y : Game | x.Locations==y.Locations && x.Players == y.Players
}

fact{
    all g : Game | all x,y : Location | gt[x,y] =>
        g.Locations.idxOf[x] > g.Locations.idxOf[y]
    all g : Game | not g.Locations.hasDups
    all g : Game | all x,y : Player | gt[x,y] =>
Finally, a requirement that makes many of the calculations easier is that the order over the Player and Location signatures and the sequence indexing are bijective, line 213. In this way sequence indexes can be used to perform basic arithmetic operations on locations in the game.

This useful correspondence is also illustrated in Figure 7.1 which is the Alloy representation of the example in Figure 6.1 already discussed on page 164.

For instance the run at line 222 checks if the predicates implementing local properties, are consistent, that is, it tries to generate an example satisfying all the local properties. More concretely, an instance obtained by running a predicate is an assignment that makes all the facts of the model true. These can be explicit facts, in our implementation the fact paragraphs at lines 15, 20, 27, 33, 39, 207 and 213, but also implicit facts contained in the declarations used for the defined signatures and their component fields, like for instance the requirement at line 12, specifying the fact that the players choose a unique location.

The variables assigned in an instance contain the sets associated with the signatures, the relations associated with the fields and the arguments of the predicates. The output details for running the predicate are included in the following output block. The run specifies a limited scope in which Alloy tries to find such an example. The scope is a multi-dimensional space of test cases in which each dimension is a bound for the variables in the constraint. In this case the scope specifies a bound of five for the signature representing game locations, and the number of players is limited to four so that the game conditions are captured.

A final clause specifies the maximum integer value, in this case ten as needed in the previously explained payoff computing function.

When the analysis finds an instance satisfying the specifications this can be visualized as a graph, as illustrated below in Figures 7.2 and 7.1.

The provided output contains useful information about the structure of the signatures considered and about the syntactic structure of the clause generated and fed to the SAT solver:

```plaintext
> Executing "Run run$1 for 5 but 10 int, 4 Player"
Sig this/Player scope <= 4
Sig ordering/Ord scope <= 1
Sig open$3/Ord scope <= 1
Sig this/Location scope <= 5
Sig this/Game scope <= 5
Sig this/Location forced to have exactly 5 atoms.
Sig this/Player forced to have exactly 4 atoms.
Sig this/Location == [[Location$0], [Location$1], [Location$2], [Location$3], [Location$4]]
```
7.2. Implementation and Illustrative Examples

Sig this/Player == [[Player$0], [Player$1], [Player$2], [Player$3]]
Sig this/Game in [[Game$0], [Game$1], [Game$2], [Game$3], [Game$4]]
Sig ordering/Ord == [[ordering/Ord$0]]
Sig open$3/Ord == [[open$3/Ord$0]]
Solver=sat4j Bitwidth=10 MaxSeq=5 SkolemDepth=1 Symmetry=20
1407216 vars. 375 primary vars. 6253223 clauses. 1262515ms.
Instance found. Predicate is consistent. 8212566ms.

The output specifies first the variables used to formulate the constraint, these can be explicitly defined signatures like the location and players in the game or predefined signatures for orders and sequences.

For each signature used there is a bound that defines the multi-dimensional space of test cases in which an exhaustive search for assignments of variables satisfying the constraint is going to be performed.

Figure 7.1: An instance of the LG model projected over the Game signature

The way Alloy Analyzer works can be regarded as a constraint solver for the relational logic behind the Alloy syntax. The concrete steps are rather those of a compiler: first the constraint is formulated using the signatures and fields describing the model and the defined predicates, next this is translated in a boolean formula over the specified scope, this translation uses standard techniques like skolemization and the fact that the scope is finite to convert formulae containing quantifiers to disjunctions over the range of the scope. Also, various strategies to improve performance by reducing the search space based on symmetries between all possible variable assignments are also employed at this stage.
The final step is to feed the obtained boolean formula to a ‘off the shelf’ SAT-solver that tries to find a satisfying assignment, which is then translated back in the Alloy relational logic and used to construct and visualize the relational structure representing the instance. The last three lines in the previous output contain information about the SAT-solver used, the number of variables and clauses, and the solving parameters and total time. If needed some of these parameters can be changed as the analysis requires. See \[52\] for a detailed description of the analysis process and the theoretical foundations of its logical background.

Besides running a predicate in order to find an example, Alloy Analyzer can also check an assertion in order to find a counterexample. In this case the analysis will also perform an exhaustive search inside the multi dimensional space of test cases specified by the scopes of signatures and relevant parameters. The difference is now that the searching involves a refutation, that is, it tries to find an assignment that makes the facts of the model true but the assertion false. If no such instance is found then the assertion is valid within the scope.

For instance the following assertion can be used to justify the entailment from Corollary 6.3.6 within a limited scope.

```plaintext
assert char {
  all g, g' : Game, p : Player | not ne[g, g', p]
}
```

```plaintext
check char for 5 but 4 Player, 10 int
```

In this case the scopes bounds are specified as in the case of the previous predicate as can be seen in the following output:
7.2. Implementation and Illustrative Examples

$ Executing "Check char for 5 but 10 int, 4 Player"

Sig this/Player scope <= 4
Sig ordering/Ord scope <= 1
Sig open$3/Ord scope <= 1
Sig this/Location scope <= 5
Sig this/Game scope <= 5
Sig this/Location forced to have exactly 5 atoms.
Sig this/Player forced to have exactly 4 atoms.
Sig this/Location == [[Location$0], [Location$1], [Location$2],
[Location$3], [Location$4]]
Sig this/Player == [[Player$0], [Player$1], [Player$2], [Player$3]]
Sig this/Game in [[Game$0], [Game$1], [Game$2], [Game$3], [Game$4]]
Sig ordering/Ord == [[ordering/Ord$0]]
Sig open$3/Ord == [[open$3/Ord$0]]
Solver=sat4j Bitwidth=10 MaxSeq=5 SkolemDepth=1 Symmetry=20
1527795 vars. 389 primary vars. 6833561 clauses. 707222ms.
No counterexample found. Assertion may be valid. 171107ms.

The last three lines contain again information about the SAT solver parameters, and variables and clauses used. These are also as explained before. The only new content is in the last line, this time the goal is to find counterexamples for the assertion. The analyzer reports that no counterexample has been found for the assertion. This is not a result that implies that the assertion is valid for any scope, it just implies validity inside the bounded scope considered and gives some confidence about the general case as many of the remaining models might share the same properties.