Automatic Furniture Rearrangement Using PBnJ

Ei Darli Aung

Computer Science Department
University of California, Los Angeles
`eidarli@cs.ucla.edu`

Abstract. This project is a major contribution to PBnJ (Plan B in Java), the instantiation of Plan B in an extension to Java with executable specifications. Plan B checks for violation of method postconditions at runtime. Upon violation, instead of halting the program, it falls back on the specification and finds a model that satisfies the postconditions using a constraint solver based on the dynamic program state on entry to the faulty method. Extensive testing and case studies were performed in order to further improve the usability and expressive power of PBnJ. The integral part of this project is adding an extension to SweetHome3D, an interior design application, using PBnJ.

1 Introduction

Introduced by Professor Millstein and his PhD student, Hesam Samimi, at UCLA, *Plan B* is a novel approach to using specifications for software reliability at runtime. Plan B performs dynamic checking of method postconditions and the declared class invariant. Upon a violation, Plan B simply ignores the method implementation and executes the specification defined in the postconditions. Using a constraint solver, it finds a *model* that satisfies the method postconditions based on the dynamic program state on entry to the faulty method. Instead of halting the program execution upon a contract violation, which is done by traditional contract checking, Plan B lets the program execution continue by updating the appropriate program variables with the result from the solver. The main benefit of this approach is that specification can be used not only as a way of checking correctness of a program implementation but also as a reliable alternative to fall back when there is a problem in execution. Plan B can recover from dynamic contract violations as well as from other runtime exceptions caused by a null pointer deference or an array-index out of bounds.

The implementation of Plan B has been added to Java as an extension, which has been called PBnJ (Plan B in Java). Programmers can express class invariants and method postconditions in a first-order relational logic similar to Alloy [2]. Class invariants and method postconditions can be defined using an optional `ensures` clause on a class declaration and on methods, respectively. Programmers can also write specifications in a procedural style using the `spec` keyword in the header of each method in which they are defined. Specification methods, *spec methods* for short, can invoke other spec methods but not regular Java methods. `ensures` clause can invoke only spec methods.

At compile time, PBnJ translates class invariants and method postconditions into not only Java predicates to be used for dynamic checking, but also relational logic which can be provided to the Kodkod constraint solver [6]. When contract checking fails at runtime or the method terminates with a Java `RuntimeException`, PBnJ automatically invokes the Kodkod solver to search for a model that satisfies the constraints based on the dynamic program state on entry of the method. If a model is found, the result from the solver is translated back to update the appropriate Java variables and fields, and the program execution continues. If Kodkod reports unsatisfiability, PBnJ throws a `ContractViolationException`.

Before a method is executed, PBnJ makes a deep copy of the reachable state from any object whose `old` field is used in the methods postconditions. `old` is a field implicitly created by PBnJ for each object and it represents the state of that object on entry to the method. After the method execution, the postcondition is checked using the deep copy. If the method postcondition is not satisfied, the fallback mechanism takes place. During fallback, the constraint solver can modify the value of any field mentioned the class invariant and the method postcondition by default. PBnJ also provides an optional `modifies fields` clause as a frame condition for programmers to override the default. Using this clause, programmers can limit a set of fields that are intended to be modified in order to improve the performance of constraint solving and to rule out illogical solutions. In addition to the `modifies fields` clause, PBnJ has an optional `modifies objects` clause to allow programmers to limit the set of objects that can be mutable in solving constraints. By default, every object reachable from `this` and formal parameters on entry of the method is considered modiable. However, if `modifies objects` clause is specified, only the set of objects computed dynamically by evaluating the given expression will be considered modifiable while the other objects are considered immutable for the purpose of solving constraints.

PBnJ proposes two types of fallback mechanism: accidental fallback and intentional fallback. Accidental fallback ensures reliability of the program by triggering the fallback process when the method throws a Java `RuntimeException` or ends in a state that violates the postcondition or the class invariant, due to an error in the method implementation. Accidental fallback mechanism automatically recovers from the error using the specification provided. Intentional fallback allows programmers to skip the implementation of tedious but rare cases and rely explicitly on specifications.

When I first joined the PBnJ project, the initial implementation of the PBnJ compiler was finished and the paper written by Hesam Samimi and Professor Millstein had been submitted to ECOOP for initial review. However, the tool was not ready for a practical use. Extensive testing and case studies were needed to investigate what needed to be improved.

After initial testing (Section 2) on a Priority Queue was done, an extension was added to SweetHome3D, an interior design application, with the use of PBnJ's intentional fallback mechanism (Secion 3). Section 3 also includes the experimental results from employing PBnJ fallback mechanism to rearrange furniture automatically.

The PBnJ compiler is available at `http://www.cs.ucla.edu/~hesam/planb`. An extended version of SweetHome3D with features mentioned in this paper is also available at the same location.

2 Initial Testing

After reading the initially submitted paper, the first thing I wanted to do was to explore the expressive power of PBnJ. I started out with a class which has a PriorityQueue of `Nodes` as a field. PriorityQueue is a class taken from the Java library. Its `size()` method is modified to have the `spec` keyword in the method header. A new spec method `getQueue()` is added to return a set of the queue's elements. `Node` is a user-defined class with an integer `value` field. Specifications were added to the `addToQueue()` method in Figure 1 in order to make sure that values in all Node instances of the priority queue are positive and that the number of elements in the queue is no more than a fixed number(3 in this experiment). If the constraints are not satisfied, it is specified to rollback to the original queue state.

There are two main objectives in creating these specifications with a Priority Queue. The first one is to test specifying constraints on different levels of abstraction: on the Priority Queue and on the value of the elements inside the Priority Queue. The second objective is to test the ability to rollback to the original state if the constraints are not satisfied. They are simple specifications, but since these types of constraints were not covered before, they exposed bugs in the PBnJ compiler. The main problem was that the priority queue was left with incorrect elements after rolling back to the old queue state when the constraints were not satisfied. For example, after a new Node with the value -5 has been added to the queue with the existing values of 10, 20, and 30, it should rollback to the old queue having the values of 10, 20 and 30. It is because the `ensures` clause in the `addToQueue()` method specifies that all elements in the priority queue must be positive and the queue stays unchanged otherwise. Since the `modifies fields` clause does not include `Node.value`, the solver cannot replace the value -5 with some positive number in order for the constraints on the left side of `||` in the `ensures` clause to be satisfied. Therefore, when a `Node` with a negative number is added to the queue, the solver is forced to make `unchanged()` true. Because of a bug in the PBnJ implementation, the fallback mechanism left the the queue in the above example with the values of -5, 10 and 20, deleting the largest value from the queue, instead of the last item added to the queue.

The specifications in this example might not look very realistic at a glance. However, the logic can be applied to any collection type object with elements of a user-defined type. In some situation, it might even make sense to impose a restriction on all the elements in a collection to maintain certain property (e.g: all values must be positive), or to limit the size of the collection, or more importantly, to fall back to the original state with no modifications to the data when the constraints are not satisfied.

```
class priorityQueueTest{
  PriorityQueue<Node> pq;
  spec public boolean isInLimit(){
    return (pq.size() <= 3);
  }
  spec public boolean allPositive(){
    return all Object n : this.pq.getQueue() |
                ((Node)n ==null || ((Node)n).value > 0);
  }
  spec public boolean unchanged(){
    return this.pq.size() == this.old.pq.size() &&
                   this.pq.getQueue() == this.old.pq.getQueue();
  }
  public void addToQueue(Node newNode)
    modifies fields PriorityQueue.size, PriorityQueue.queue
    ensures ((isInLimit() && allPositive() &&
              this.pq.size() == (this.old.pq.size() + 1))
            || unchanged()){
        pq.add(newNode);
  }
  public static void main(String args) {
    priorityQueueTest pqt = new priorityQueueTest();
    pqt.add(new Node(10));
    pqt.addToQueue(new Node(20));
    pqt.addToQueue(new Node(30));
    pqt.addToQueue(new Node(-5));
  }
}
```

Fig. 1: Specifications on a Priority Queue with elements of type `Node`. The `Node` class (not shown) includes an integer `value` field. `getQueue()` is a spec method added to the PBnJ version of PriorityQueue class and it returns a set of the queue's elements. The `modifies fields` clause prevents the solver from changing the integer value stored in each element of the queue, but instead only the size and the queue array of the priority queue, forcing the solver to make `unchanged()` true.

3 Case Study

3.1 SweetHome3D

After initial testing was done, I looked into several open source Java applications to use as case studies for PBnJ. The winner was SweetHome3D [5], an interior design application implemented in Java. Users can create rooms, add or rearrange pieces of furniture manually in a 2D plane and view the results in a 3D view. It is the best choice for a case study because it is a fairly popular open source application with beautiful 2D and 3D images and it has an interesting limitation which can be addressed using PBnJ while showcasing the main features of PBnJ.

While it is not physically possible to have a piece of furniture overlapping the other, SweetHome3D does not attempt to move the pieces of furniture when they are overlapped. When a new piece of furniture needs to be added to a room with many pieces of furniture which have already been placed, the user will potentially need to move every single piece manually to make room for the new piece. This limitation can be addressed using PBnJ's intentional fallback mechanism to automatically rearrange pieces of furniture to remove overlaps. Details on how this is implemented with different criteria as well as other enhancement added to SweetHome3D are explained below.

1. When the user adds a new piece of furniture, the application will make sure that no pieces overlap and that pieces are not moved too far away from their original positions while keeping their relative position to other pieces. It will also make sure that furniture pieces will not be placed at random places. If the user overlaps a new piece on the right side of an existing piece, the new piece should be placed next to the existing piece only on its right side. This feature can be implemented easily by relying on the PBnJ intentional fallback mechanism, instead of a manual implementation, which can be cumbersome and error prone.
2. When moving pieces of furniture to make room for the newly added piece, the application will first attempt to move an interfering piece along the X-axis by modifying just the x coordinate of the center of the piece. If the attempt fails with UNSAT, it will try to modify both x and y coordinates of the center of the piece. For example, the user might want to squeeze a chair between two chairs which have already been placed next to each other. In this case, the existing chairs should slide horizontally to each side of the new chair. Implementing this feature requires the use of nested fallbacks in PBnJ.
3. When adding a new piece which overlaps an existing piece of furniture, the existing piece will be moved but the new piece will be pinned down to where it is being placed. It is because when the user adds a piece to a certain location, the user might not want the piece to be moved to somewhere else.
4. By default, the position of any piece can be modifiable if necessary during auto-rearrangement. However, the user can select a piece and mark it as

unmodifiable so that the piece will not be part of auto-rearrangement. Moreover, a piece will be considered for moving only when it is necessary even if it is being marked *modifiable*. A benefit of limiting the number of pieces to consider moving is to improve the constraint solving time by making the search space smaller. This can be done easily in PBnJ with the use of `modifies objects` clause.

5. When moving furniture pieces, the application will place them only within a specified area. The area can be defined by the user dynamically via a GUI. This makes sure that no pieces of furniture will be moved out of a room just to make a space for the newly added piece.
6. All of the above constraints are also applied to situations in which the user moves around the existing pieces of furniture.
7. A menu option to enable/disable automatic rearrangement of furniture using PBnJ fallback mechanism. Since the constraint solver gives a solution out of all possible solutions, the furniture pieces might end up in places where the user would not want them in some cases. In such situations, having this menu option will reduce the user's frustration from having to move things back to where they were.

3.2 Implementation

Limitations and challenges In order to make sure that the exiting SweetHome3D code was compatible with the PBnJ compiler, I compiled it with the PBnJ compiler with no modifications. It resulted in many compiler errors. Some of them were because of the bugs in the PBnJ implementation while some of them were from Polyglot Java 5.0 extension [3], an extensible compiler framework for Java, on which PBnJ is implemented. Some of the significant errors include not being able to handle generics properly, not being able to recognize `.class` of any object, not being able to evaluate the value of an enum type variable in a `switch` statement, and the lack of support for static initializer blocks in Polyglot. All of the issues were fixed except the last one, for which I created a work-around by replacing the static initializer block with a method. The method is invoked from the class constructors only if the static field being initialized in the initializer block is `null`.

It was also necessary to add new specification methods to some of the java.util libraries. For example, in order to properly capture the meaning of quantifying over an ArrayList object in relational logic, `toPBJSet()` spec method was added to java.util.ArrayList class. This method returns the set of elements inserted to the ArrayList.

Currently, PBnJ does not have support for float data type simply because Kodkod cannot handle floats. However, SweetHome3D uses float for all furniture attributes required for the spec methods. As a work-around, I created an integer version for each of those attributes in parallel to the original float version. Whenever the original float version of the variable gets updated, the integer version is also updated with the rounded value of the original float value. The integer values are used in the specifications and passed to the constraint solver. The

```
private void doFallback(){
 try{
     moveAlongXaxis(); // A
 }
 catch(ContractViolationError e0){
    try{
        moveAlongAnyAxis();  // B
    }
    catch(ContractViolationError ae){
       try{
           moveAlongXaxis_relaxed();  // C
        }
        catch(ContractViolationError e){
           try{
              moveAlongAnyAxis_relaxed();  // D
           }
           catch(ContractViolationError e1){
                System.out.println(e1.getMessage());
}}  }   }  }
```

Fig. 2: Illustration of different fallbacks using try/catch blocks

results from the solver are then set back to the float version of the variable so that the rest of the application can execute without any modifications.

SweetHome3D provides four different kinds of unit for furniture attributes on GUI: centimeter, millimeter, inch and meter. The values used in calculations are however in `cm`. With an average width of furniture being 400 cm, the numbers passed to the constraint solver make up a large search space. This leads to a slow solving time and a frequent crash with the out-of-memory exception, as more and more pieces of furniture are being added. To scale this issue well, the values of furniture attributes are divided by a fixed constant(15 in this experiment) before being passed to the solver, and multiplied by the same constant to get the actual value back to use. An average fallback time was significantly reduced to less than a second after using this optimization technique.

Specifications This section describes how PBnJ specifications are used to implement the features laid out in Section 3.1.

Figure 2 shows `doFallback()` method which invokes different fallback methods depending on the satisfiability of the previous case. The method is invoked from the SweetHome3D existing methods for adding a new piece of furniture and moving an existing piece of furniture if the *auto-rearrange furniture* menu option is checked. If the menu option is unchecked, the original SweetHome3D methods for adding and moving pieces of furniture are invoked.

```
public void moveAlongXaxis()
  modifies fields HomePieceOfFurniture.i_x
  modifies objects objectsToBeMoved()
  ensures notOverlapped() && notTooFar() && withinRange()
        && keepRelativePositionInSurrounding() { }

public void moveAlongAnyAxis()
  modifies fields HomePieceOfFurniture.i_x, HomePieceOfFurniture.i_y
  modifies objects objectsToBeMoved()
  ensures notOverlapped() && notTooFar() && withinRange()
        && keepRelativePositionInSurrounding() { }

public void moveAlongXaxis_relaxed()
  modifies fields HomePieceOfFurniture.i_x
  modifies objects objectsToBeMoved()
  ensures notOverlapped() && notTooFar_relaxed() && withinRange()
        && keepRelativePositionInSurrounding() { }

public void moveAlongAnyAxis_relaxed()
  modifies fields HomePieceOfFurniture.i_x, HomePieceOfFurniture.i_y
  modifies objects objectsToBeMoved()
  ensures notOverlapped() && notTooFar_relaxed()  && withinRange()
        && keepRelativePositionInSurrounding() { }
```

Fig. 3: Four empty methods which rely on the PBnJ intentional fallback mechanism to search for a model that satisfies their postconditions. The spec methods invoked in the `ensures` clauses are illustrated in Figure 6.

There are four method invocations which trigger the PBnJ fallback mechanism in `doFallback()` at lines marked A, B, C and D. All four methods, whose specifications are shown in Figure 3, are intentionally left empty with postconditions which make sure that there are no overlaps, that the new position of each piece is not too far away from its original position, that all pieces of furniture are in the specified range, and that the new piece and the other pieces in its surrounding maintain the same relative position with respect to each other before and after the move. The only differences among these four methods are how far each piece can be moved from its original position and which coordinates can be modified. `modifies fields` clause is used to limit which coordinates can be modified in performing fallback. Using this annotation also improves the performance of constraint solving by limiting the search space.

`moveAlongXaxis()` has the strictest specification which allows modifications to only the X coordinate of the center of each piece and requires pieces to be moved only up to half of its width or its depth as specified in `notTooFar()`. By allowing only X coordinates to be mutable, pieces of furniture will be placed next to each other horizontally if the constraints are satisfied, which is as expected in most cases in terms of user experience. If the constraints in this method are too strict to be satisfiable, PBnJ will throw an assertion error which can be caught to trigger another fallback event to solve for a model with less strict constraints. The second fallback method `moveAlongAnyAxis()` allows modifications to either X or Y coordinates of the center of pieces if necessary, while keeping the same movable distance criteria as the first fallback method. If the solver cannot find any model which satisfies this second set of constraints, the third set of constraints in `moveAlongXaxis_relaxed()` is considered. `moveAlongXaxis_relaxed()` allows modifications to X coordinates only, but allows each piece to be moved as far as its width or its depth as defined in `notTooFar_relaxed()`. In a similar style, if the third set of constraints is not satisfiable, the last set of constraints in `moveAlongAnyAxis_relaxed()`, which allows both X and Y coordinates to be modified and allows each piece to be moved up to its full width and/or its depth, is passed to the solver. If even the last set of constraints is not satisfiable, no pieces will get moved. An example of automatic rearrangement of furniture using the PBnJ fallback mechanism is illustrated in Figure 4 with the screenshots from the application.

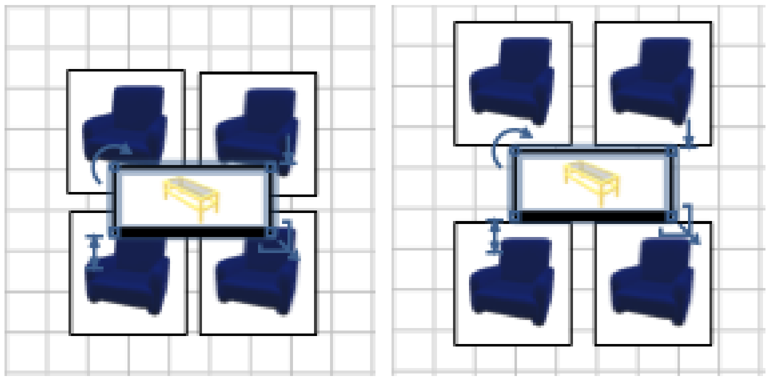


Fig. 4: (a) Four chairs and a coffee table overlapping the chairs. (b) PBnJ's fallback mechanism automatically rearranges the furniture.

One of the postconditions that all four methods in Figure 3 ensure is retaining the relative position of a piece with respect to other pieces. For example, if a chair is on the right side of a table, it should stay on the same side after the auto-rearrangement. The initial version of `keepRelativePosition()` maintains the relative position of a piece with respect to every other piece on the plane. The drawback of this approach is that, as too many pieces are getting added, the constraints become too strict trying to keep the same relative position of the new piece to the pieces, including the ones which are so far away from it. As shown

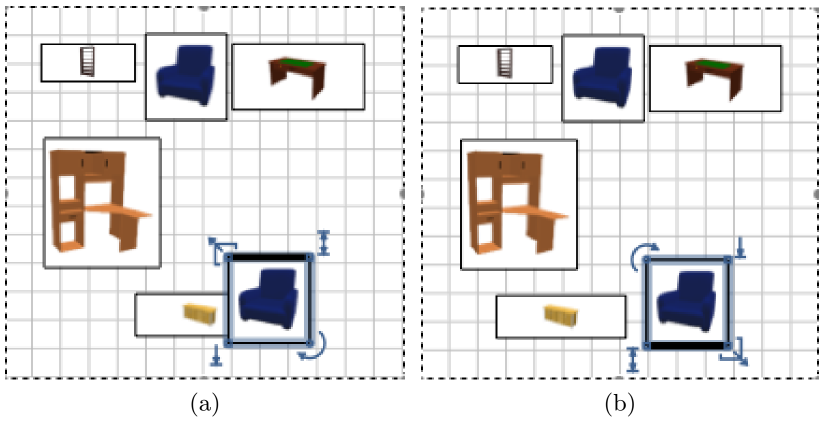


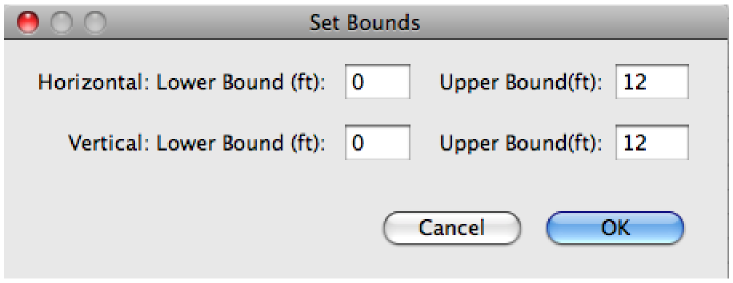
Fig. 5: The selected arm-chair is the piece being added and its position is not modifiable. (a) An UNSAT situation with the initial `keepRelativePosition` method. It is unsatisfiable because every piece of furniture maintains the relative position of its center to the center of every other piece, and thus, moving the dresser to the left will violate its relative position constraint with respect to the center of the arm-chair at the top. (b) The same set of furniture pieces which gets auto-rearranged after using `keepRelativePositionInSurrounding()`.

in Figure 5(a), in some cases, it becomes an annoyance to the user, who expects to get the dresser moved to the left when the arm chair at the bottom is being added, but ends up with the overlapped situation because moving the dresser to the left will violate the relative position of its center to the center of the arm chair at the top. It is obvious that the situation will get worse if more and more pieces are added. As implemented in `keepRelativePositionInSurrounding()` in Figure 6, it makes more sense to enforce this constraint only on the pieces that are *sufficiently close* to the piece that is being added. Piece A is considered *sufficiently close* to piece B if an edge of A is in the radius of B's width or B's depth from the center of B. An algorithm to calculate all pieces sufficiently close to the area being overlapped starts from the piece that is overlapping an existing piece and branches out to all pieces that are *sufficiently close* to the current piece. The algorithm continues until all pieces are scanned.

The `withinRange()` spec method in Figure 6 makes sure that each piece is within the specified horizontal and vertical boundaries which can be set dynamically by the user through the GUI shown in Figure 7. It is necessary to add/substract half of the width and the depth of `p1` in the method in order to make sure that the whole piece completely resides in the region and that it can be moved as far as its outer edge touches the boundary, on the other hand. Current implementation allows the user to specify any number from 0ft to 40ft using the GUI for the best performance.

```
spec public boolean notOverlapped(){
  return  all HomePieceOfFurniture p1 : this.temp_furniture |
          all HomePieceOfFurniture p2 : this.temp_furniture |
              ( p1 == p2 ||
         (abs(p1.getIX() - p2.getIX()) >=
                ((p1.getIWidth() + p2.getIWidth())/2))
      || (abs(p1.getIY() - p2.getIY()) >=
                ((p1.getIDepth() + p2.getIDepth())/2)));
}
spec private boolean notTooFar(){
  return all HomePieceOfFurniture p: this.temp_furniture |
          ((abs(p.getIX() - p.old.getIX()) <= p.getIWidth()/2)
     && (abs(p.getIY() - p.old.getIY()) <= p.getIDepth()/2));
}
spec private boolean notTooFar_relaxed(){
  return all HomePieceOfFurniture p: this.temp_furniture |
          ((abs(p.getIX() - p.old.getIX()) <= p.getIWidth())
     && (abs(p.getIY() - p.old.getIY()) <= p.getIDepth()));
}
spec private boolean keepRelativePositionInSurrounding(){
  return all HomePieceOfFurniture p1: this.overlappedFurn |
         all HomePieceOfFurniture p2: this.overlappedFurn |
         (( compare(p1.getIX(), p2.getIX()) ==
                   compare(p1.old.getIX(),p2.old.getIX()) ) &&
          ( compare(p1.getIY(), p2.getIY()) ==
                   compare(p1.old.getIY(),p2.old.getIY()) ));
}
spec public boolean withinRange(){
  return all HomePieceOfFurniture p1 : this.temp_furniture |
         ( p1.getIX() >= (hor_lower_bound + (p1.getIWidth()/2)) &&
          p1.getIX() <= (hor_upper_bound - (p1.getIWidth()/2)) &&
          p1.getIY() >= (ver_lower_bound + (p1.getIDepth()/2)) &&
          p1.getIY() <= (ver_upper_bound - (p1.getIDepth()/2)));
}
```

Fig. 6: The spec methods used in the Figure 3. `this.temp_furniture` is an ArrayList of current pieces of furniture in the home. `this.overlappedFurn` is an ArrayList of furniture pieces that are sufficiently close to the overlapped region with the new piece in it. The `compare` method returns -1 if the first argument is less than the second argument and 1 in any other cases. The `getIX` and `getIY` methods return the coordinates, rounded to the nearest integer, of the center of a piece of furniture.



Horizontal: Lower Bound (ft): 0 Upper Bound(ft): 12

Vertical: Lower Bound (ft): 0 Upper Bound(ft): 12

Fig. 7: The *Set Bounds* dialog box for setting the horizontal and vertical range to move pieces of furniture within.

During fallback, instead of considering all pieces of furniture to be modifiable, it is an obvious improvement in performance to just consider the pieces that are necessary to move. As an enhancement to SweetHome3D, I implemented an option to let the user click on pieces of furniture to explicitly specify which ones cannot be moved as part of the auto-rearrangement upon an overlap. This can be useful in a situation where the user is satisfied with the placement of certain pieces of furniture—for example, a dining set—and does not want them to be part of auto-rearrangement. By default, position of every piece of furniture is modifiable except the new piece because the user will expect the new piece to stay in the same place where he/she puts in. In order to attempt to modify only the pieces that are both marked modifiable and necessary to move, all I needed to do is to include `modifies objects objectsToBeMoved()` in the empty methods in Figure 3. `objectsToBeMoved()` returns the set of objects that are both marked modifiable and sufficiently close to the area that is being overlapped. PBnJ evaluates the `modifies objects` clause dynamically and passes the resulting set of objects to the solver as modifiable while keeping the other objects as immutable.

Benefits Auto-rearrangement of furniture with all the criteria mentioned before would be tedious and error prone to implement manually. However, using PBnJ, the programmer can just write the required postconditions and explicitly rely on its intentional fallback mechanism to solve the problem.

Referring back to the nested fallbacks in Figure 2, it is worth mentioning that there are several benefits to have such different fallback scenarios with different level of constraints. In the best case, the solver has to deal with the smallest possible search space and therefore, it is able to return the result in the fastest solving time. On average, fallback does not occur four times for all four method calls. Any of the first three takes much less time than the last set of constraints to solve for a model. Therefore, if only the last set of constraints was implemented, it would have taken more time to solve for a model on average. Lastly, it also improves the user experience by placing the new piece near the existing ones as close as possible without being overlapped.

Table 1: Fallback overhead including copying, contract checking, conversion to Kodkod, Kodkod's translation to SAT and SAT solving time using MiniSat [1] of different fallback events on rearranging various number of pieces of furniture in a 15ftx15ft room. The statistics were taken on a machine with Mac OS X version 10.5.8 running on 2.4 GHz Intel Core 2 Duo with 4 GB of Memory. A=`moveAlongXaxis()`, B=`moveAlongAnyAxis()`, C=`moveAlongXaxis_relaxed()`, D=`moveAlongAnyAxis_relaxed()`

Number of Pieces	A	B	C	D
2	.127s	.171s	.137s	.188s
3	.178s	.275s	.189s	.324s
4	.239s	.365s	.236s	.380s
5	.324s	.489s	.297s	.533s
6	.356s	.671s	.377s	.934s
7	.470s	.887s	.516s	.946s
8	.580s	1.174s	.838s	1.330s
9	.657s	1.383s	.693s	1.675s

Currently, PBnJ does not have support for this style of nested fallbacks automatically. The programmer has to invoke different methods individually using nested try/catch blocks. In the future, it would be nice to have a PBnJ feature which lets the user to specify the sets of constraints with different level of strictness all in one shot for one method. It will also speed up the total fallback time by doing the exact same procedures for subsequent fallbacks only one time, such as making a deep copy of the reachable state from any object whose old field is used in the method postconditions.

3.3 Experimental results

Total method invocation time for fallback varies depending on the bounds set by the user, number of furniture pieces involved in the constraints, and the type of methods for which fallback mechanism is triggered. The average amount of time it takes to solve for a model for each method is listed in Table 1. If fallback mechanism for all four methods needs to be triggered, the total fallback overhead will be the combination of overhead for each method (A+B+C+D). In general, although the response time to rearrange the furniture is not instantaneous, it is not too slow to disrupt the user experience.

This extension to SweetHome3D is a great example of how PBnJ can be used to easily implement a tedious feature without having to worry about handling corner cases. PBnJ can be used in any other existing software to ensure reliability of the software and to recover from runtime errors and exceptions.

More examples of applications into which PBnJ has been incorporated can be found in [4].

4 Conclusion

This project plays an important role in making PBnJ mature. An extension added to SweetHome3D using PBnJ has been mentioned in [4] as a good application for PBnJ intentional fallback mechanism and the use of `modifies objects` clause. Many bugs and issues of PBnJ were exposed and fixed due to testing and the case study done in this project, resulting in a much more improvement in usability and the expressive power of PBnJ than before.

5 Acknowledgments

Many thanks to Professor Millstein and Hesam Samimi for giving me a great opportunity to be part of their PBnJ project and be a co-author of the Plan B paper [4].

References

1. N. Een and N. Sorensson. MiniSat. http://minisat.se.
2. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
3. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.
4. H. Samimi, E. D. Aung, and T. Millstein. In *ECOOP '10, European Conference on Object-Oriented Programming, Maribor, Slovenia, June 21-25, 2010, Proceedings*. Springer, 2010.
5. SweetHome3D. http://www.sweethome3d.eu.
6. E. Torlak. *A constraint solver for software engineering: Finding models and cores of large relational specifications.* Ph.D. dissertation, Massachusetts Institute of Technology, 2009.