Wire Routing and Satisfiability Planning

Esra Erdem, Vladimir Lifschitz, Martin D. F. Wong

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA
{esra,vl,wong}@cs.utexas.edu

Abstract. Wire routing is the problem of determining the physical locations of all the wires interconnecting the circuit components on a chip. Since the wires cannot intersect with each other, they are competing for limited spaces, thus making routing a difficult combinatorial optimization problem. We present a new approach to wire routing that uses action languages and satisfiability planning. Its idea is to think of each path as the trajectory of a robot, and to understand a routing problem as the problem of planning the actions of several robots whose paths are required to be disjoint. The new method differs from the algorithms implemented in the existing routing systems in that it always correctly determines whether a given problem is solvable, and it produces a solution whenever one exists.

1 Introduction

Very large scale integrated circuits (VLSI), with millions of transistors and wires on a single silicon chip, are too complex to design without the aid of computers. Advances in integrated circuit technology will result in more complex chips in the near future—it is predicted that there will be over 1 billion transistors and wires on a single chip in about 10 years [13]. As a result, research and development in computer-aided design (CAD) software is very active in both industry and academia.

Routing is an important step in CAD for VLSI circuits [7]. It is the problem of determining the physical locations of all the wires interconnecting the circuit components (transistors, gates, functional units, etc.) on a chip. Since the wires cannot intersect with each other (otherwise resulting in short circuits), they are competing for limited spaces, thus making routing a difficult combinatorial optimization problem. In practice, the routing problem for the whole VLSI chip is decomposed into smaller routing problems [7]. The chip is partitioned into an array of rectangular regions. After determining the connections between adjacent regions, the routing of all the regions are carried out independently. But even for an individual region the problem is computationally difficult. VLSI routing has been shown to be NP-complete [14], and there are many heuristic routing algorithms in the literature [7].

In this paper, we present a new approach to VLSI routing that uses action languages [3] and satisfiability planning [5]. All existing routing systems are based on variations of the sequential maze routing approach using a shortest path algorithm connecting one wire at a time [6, 11]. A major shortcoming of these algorithms is that they cannot guarantee finding a routing solution even when one exists. The new method differs from them in that it is complete: it always correctly determines whether a given routing problem is solvable, and it produces a routing solution whenever one exists.

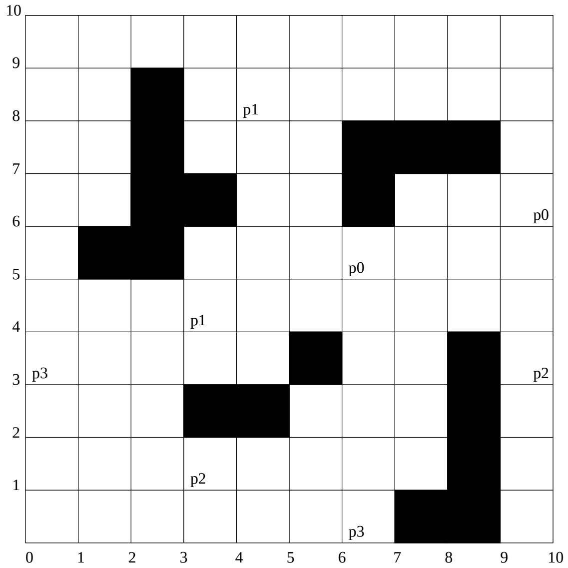

Fig. 1. A routing problem with 4 wires.

Consider, for instance, the routing problem shown in Fig. 1. The wiring space here is a rectangular grid. The goal is to connect 4 pairs of points ("pins")—the two points labeled $p0$, the two points labeled $p1$, and so on—without passing through the obstacles, shown in black. A solution—actually, the solution found by the method proposed in this paper—is given in Fig. 2. If we try to solve this problem by finding first a shortest path between the points labeled $p0$, and then a shortest path between the points labeled $p1$ in the part of the grid that is still available, we will arrive at a partial solution like the one shown in Fig. 3. This partial solution cannot be extended to a complete solution, however, because the points labeled $p2$ cannot be connected without intersecting the first of the two paths selected earlier.

The idea of the new method is to think of each path as the trajectory of a robot moving along the grid lines, and to understand a routing problem as the problem of planning the actions of several robots. In the example above, the problem involves 4 robots. The initial position of Robot 0 is assumed to be (6,5), and its goal is to reach point (10,6) (or the other way around), and similarly for

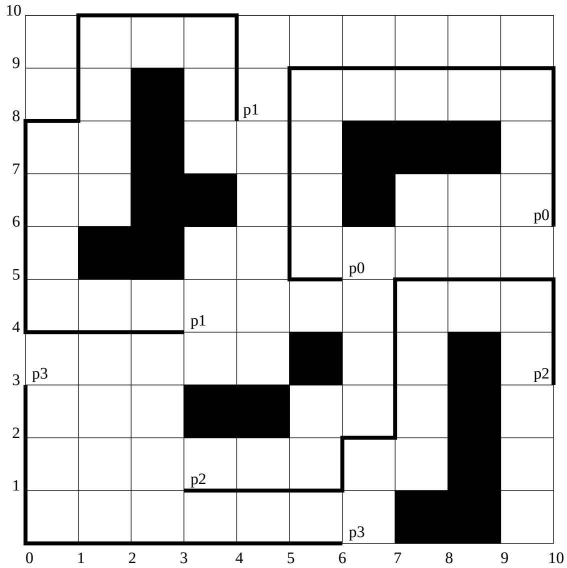

Fig. 2. A solution to the problem from Fig. 1.

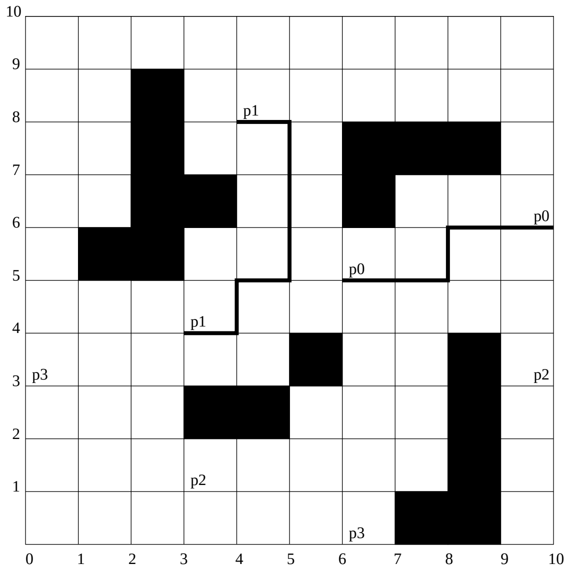

Fig. 3. A partial solution to the problem from Fig. 1. It cannot be extended to a complete solution.

the other robots. The actions that a robot can perform are to move left, right, up or down to the closest grid point, or to do nothing. We describe the effects of these actions in action language $\mathcal{C}$ [4].

The action language $\mathcal{C}$ is based on the theory of causal explanation proposed in [9]. Therefore, the view of causality adopted in $\mathcal{C}$ distinguishes between asserting that a certain fact "holds" and making the stronger assertion that it "is caused" (or "has an explanation").

The language $\mathcal{C}$ has propositions of two kinds: *static laws* of the form

$$\textbf{caused } F \textbf{ if } G$$

and *dynamic laws* of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H.$$

Here F and G are formulas whose atomic components represent fluents. The formula H is of a more general kind: in addition to fluents, it is allowed to contain the names of actions. Syntactically, action names are treated as atomic formulas; an assignment of truth values to action names represents the composite action which is executed by performing concurrently all elementary actions whose names are assigned the value *true*.

In the language $\mathcal{C}$,

(i) an expression of the form

$$U \textbf{ causes } F \textbf{ if } G$$

where U is a propositional combination of elementary action names and F, G are propositional combination of fluent names, stands for the dynamic law

$$\textbf{caused } F \textbf{ if } \mathit{true} \textbf{ after } G \wedge U.$$

(ii) an expression of the form

$$\textbf{nonexecutable } U \textbf{ if } F$$

where U is a propositional combination of elementary action names and F is a propositional combination of fluent names, stands for the dynamic law

$$\textbf{caused } \mathit{false} \textbf{ if } \mathit{true} \textbf{ after } F \wedge U.$$

(iii) an expression of the form

$$\textbf{never } F$$

where F is a propositional combination of fluent names, stands for the static law

$$\textbf{caused } \mathit{false} \textbf{ if } F.$$

These and other abbreviations are introduced in [3].

The semantics of $\mathcal{C}$ defines how a set of propositions describes a "transition system"—a directed graph whose vertices are "states" and whose edges are labeled by "actions." A state is characterized by an assignment of truth values to fluent names, and an action is characterized by an assignment of truth values to action names. See [4] for details.

We use the Causal Calculator[1] (CCALC) to find a plan for the planning problem that corresponds to a wire routing problem. The Causal Calculator uses literal completion [9] to reduce a planning problem described in $\mathcal{C}$ to the problem of finding a satisfying interpretation for a set of propositional formulas, and then passes on these formulas to a satisfiability solver, such as RELSAT [1].

In the next two sections we provide a more detailed description of the new method as it applies to the problem above. Then we show that our approach can handle various kinds of additional routing constraints which ensure that a circuit meets its performance specification: constraints on the lengths of the wires, essential because signal delay through a wire is proportional to its length (Sections 4 and 5), and spacing constraints between the wires, related to the problem of avoiding signal interferences (Section 6).

2 Input and Output of CCALC

As discussed in the introduction, for each pair of points that need to be connected we imagine a robot that travels between these points. The position of Robot `N` is described by the propositional fluents `at_x(N,XC)` ("the x-coordinate of `N` is `XC`") and `at_y(N,YC)`. We also use the expression `at(N,XC,YC)` that is expanded into the conjunction of these fluents by the CCALC macro expansion mechanism. The actions affecting the position of Robot `N` are denoted by expressions of the form `move(N,D)`, where `D` is one of the directions `left`, `right`, `down`, `up`.

To express that the robots' paths don't loop and don't intersect each other, we use the propositional fluent `occupied(N,XC,YC)`—"point `(XC,YC)` has been visited by Robot `N`." Initially, this fluent is true only if `(XC,YC)` is the initial position of Robot `N`. The set of true fluents of this form becomes larger as robots move to new positions.

Fig. 4 shows the CCALC input file representing the routing problem from Fig. 1. The `include` directives in the middle of the file refer to two other files: `obstacles0.t`, describing the shape of the obstacles in this example, and `routing.t`, describing the effects and the executability of actions in the routing domain. Parts of file `routing.t` are discussed in the next section. The number of wires and the size of the grid are represented in that file by the macros `k`, `maxX` and `maxY`. Their numeric values are defined in each particular routing problem.

The description of the planning problem consists of a set of given facts and a goal. The symbol `0:` at the beginning of every fact tells CCALC that the fact is assumed to hold at time 0 (that is to say, is an initial condition). The first fact

[1] `http://www.cs.utexas.edu/users/tag/cc` .

```
:- macros k -> 3;
          maxX -> 10;
          maxY -> 10.

:- include 'obstacles0.t'.
:- include 'routing.t'.

:- plan
facts::
0: (occupied(N,XC,YC) ->> at(N,XC,YC)),
0: at(0,6,5),
0: at(1,3,4),
0: at(2,3,1),
0: at(3,0,3);
goal::
12..17: (at(0,10,6) && at(1,4,8) && at(2,10,3) && at(3,6,0)).
```

Fig. 4. Input file for the problem from Fig. 1

characterizes the initial value of `occupied(N,XC,YC)`.[2] We could have replaced this conditional by an equivalence, but there is no need to do this, because file `routing.t` declares `occupied(N,XC,YC)` to be a fluent false by default. The other facts give the initial positions of the robots. The symbol `12..17` in the goal instructs CCALC to try first to find a plan of length 12; if there is no such plan then try length 13, and so on, up to 17. In the case of the routing problem, the length of a plan corresponds to the maximum of the lengths of the wires.

Given this input file, CCALC reports that there is no solution of length 12, 13 or 14, and then produces a plan:

```
0.  at_y(0,5)  at_y(1,4)  at_y(2,1)  at_y(3,3)
    at_x(0,6)  at_x(1,3)  at_x(2,3)  at_x(3,0)

ACTIONS: move(0,left) move(1,left) move(2,right) move(3,down)

1.  at_y(0,5)  at_y(1,4)  at_y(2,1)  at_y(3,2)
    at_x(0,5)  at_x(1,2)  at_x(2,4)  at_x(3,0)

ACTIONS: move(0,up) move(1,left) move(2,right)

2.  at_y(0,6)  at_y(1,4)  at_y(2,1)  at_y(3,2)
    at_x(0,5)  at_x(1,1)  at_x(2,5)  at_x(3,0)

ACTIONS: move(0,up) move(1,left) move(2,right) move(3,down)

. . . . . . . . . . . . . .
```

[2] In CCALC input files, the propositional connectives are denoted by `->>` (implication), `&&` (conjunction), `++` (disjunction) and `-` (negation).

```
ACTIONS: move(0,down) move(1,down)

14.  at_y(0,7)  at_y(1,9)  at_y(2,3)  at_y(3,0)
     at_x(0,10)  at_x(1,4)  at_x(2,10)  at_x(3,5)

ACTIONS: move(0,down) move(1,down) move(3,right)

15.  at_y(0,6)  at_y(1,8)  at_y(2,3)  at_y(3,0)
     at_x(0,10)  at_x(1,4)  at_x(2,10)  at_x(3,6)
```

This is the solution shown in Fig. 2. RELSAT took 59 seconds to find it. (In our experiments, we used an UltraSPARC that has 124 MB main memory, runs SunOS 5.5.1, and has a 167 MHz CPU.)

3 The Routing Domain

In file `routing.t`, the effect of action `move(N,right)` is described by the proposition[3]

```
move(N,right) causes at_x(N,X)
              if at_x(N,XC) && X is XC+1 && XC < maxX.
```

(the execution of this action when `at_x(N,XC)` holds for some `XC` that is not at the right boundary of the grid makes `at_x(N,XC+1)` true). There is no need to postulate that the y-coordinate of Robot `N` and the coordinates of the other robots remain the same, because the coordinates of robots are declared to be "inertial"—they don't change their values if there is no evidence that they do. There is no need to say that `at_x(N,XC)` becomes false: the action affects this fluent indirectly, because the uniqueness of the x-coordinate of a robot is postulated in `routing.t` in the form

```
caused -at_x(N,XC)  if at_x(N,XC1) && -(XC=XC1).
```

(whatever value the x-coordinate of Robot `N` currently has, there is a cause for it not to have any other value).[4]

When a robot is on the right edge of the grid, it cannot move right:

```
nonexecutable move(N,right) if at_x(N,XC) && XC>=maxX.
```

Similar postulates describe moves in other directions.

We prevent robots from hitting obstacles by postulating

```
never at(N,XC,YC) && blocked(XC,YC).
```

[3] For the syntax of action language $\mathcal{C}$ as used in the input language of CCALC, see [3, Section 6].

[4] The solution to the frame problem and ramification problem incorporated in $\mathcal{C}$ and CCALC is based on the ideas of [12, 2, 15, 9].

Here `blocked(XC,YC)` is a macro defining the shape of the obstacles.

Fluent `occupied(N,XC,YC)` is characterized by the propositions

```
caused occupied(N,XC,YC) if at(N,XC,YC).
caused occupied(N,XC,YC) after occupied(N,XC,YC).
```

(the set of points visited by Robot `N` includes its current position and all points it had visited by the previous time instant). Using this fluent, we can say that paths of different robots don't intersect:

```
never occupied(N,XC,YC) && occupied(N1,XC,YC) && (N < N1).
```

and that a robot never visits the same point twice:

```
caused false if at(N,XC,YC) after occupied(N,XC,YC) && -at(N,XC,YC).
```

The last conjunctive term is necessary to allow a robot not to move; without it, all robots would make the same number of moves, and all paths would have equal lengths.

4 Bus Routing

A bus is a set of wires, each connecting a source pin and a sink pin, where the source pins are all adjacent and the sink pins are all adjacent. In bus routing, given several pairs of points on a rectangular grid, we want to find a configuration of a bus such that all wires are of the same length: we want the signal delays through all wires to be equal. The need to express the equality of the lengths is the main special feature of bus routing problems.

A bus routing problem, along with its solution found by CCALC, is displayed in Fig. 5. The input file for this program is shown in Fig. 6. The equality of the lengths of all paths is expressed there by the proposition

```
caused false if at(N,XC,YC) after at(N,XC,YC).
```

The run time of RELSAT in this example is 36 seconds.

In some cases, a bus routing problem has no solution but becomes solvable if we relax the condition on the lengths of wires. For instance, with the configuration of obstacles shown in Fig. 7, it is impossible to connect all pairs of pins by paths of the same length, but there is an "approximate solution" in which the lenghts of wires do not differ by more than 2 (Fig. 8).

To find an "approximate solution" for the problem presented in Fig. 8 using CCALC, we add a constraint telling that robots always move until their goals have been achieved. Then we can instruct CCALC to look for paths whose lengths are between 11 and 13 by replacing the goal in Fig. 6 with

```
((11: at(0,8,4) ++ 12: at(0,8,4) ++ 13: at(0,8,4)) &&
 (11: at(1,8,5) ++ 12: at(1,8,5) ++ 13: at(1,8,5)) &&
 (11: at(2,8,6) ++ 12: at(2,8,6) ++ 13: at(2,8,6))).
```

If the goal is modified in this way (and file `obstacles1.t` is modified to reflect the configuration of obstacles in Fig. 7), CCALC generates the approximate solution shown in Fig. 8. The run time of RELSAT is 36 seconds.

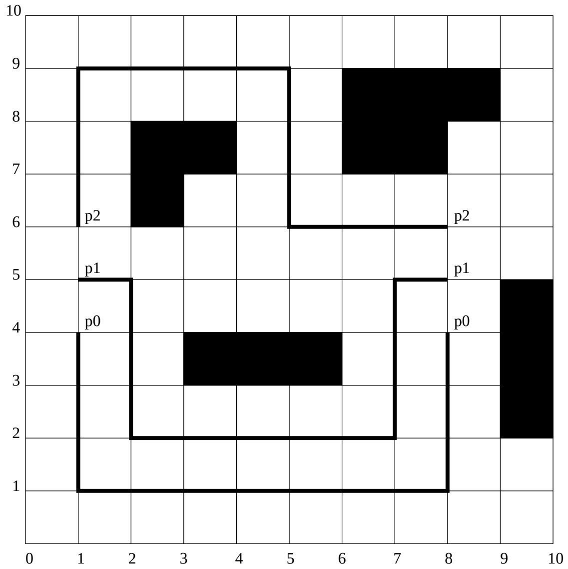

Fig. 5. A bus routing problem. The wires are required to have the same length.

```
:- macros k -> 2;
          maxX -> 10;
          maxY -> 10;
          maxLength -> 15.

:- include 'obstacles1.t'.
:- include 'routing.t'.

% robots always move
caused false if at(N,XC,YC) after at(N,XC,YC).

:- plan
facts::
0: (occupied(N,XC,YC) ->> at(N,XC,YC)),
0: at(0,1,4),
0: at(1,1,5),
0: at(2,1,6);
goal::
12..maxLength: (at(0,8,4) && at(1,8,5) && at(2,8,6)).
```

Fig. 6. Input file for the problem from Fig. 5

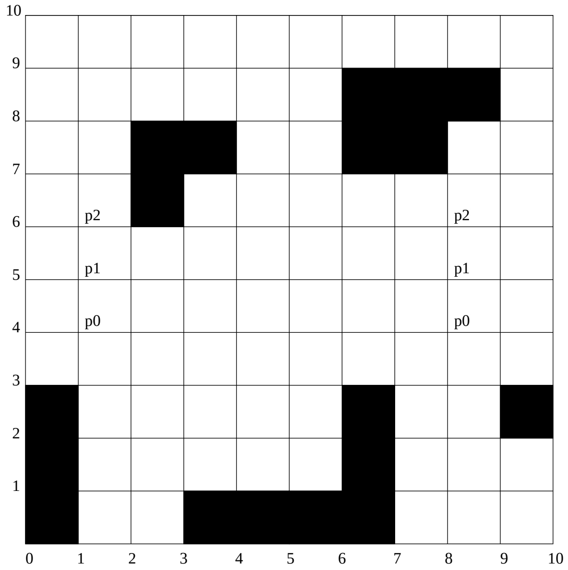

Fig. 7. A bus routing problem that has no precise solution.

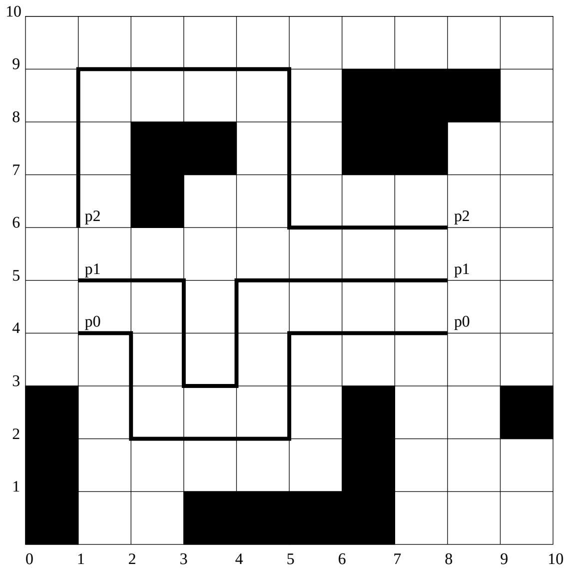

Fig. 8. An approximate solution to the problem from Fig. 7. The differences between the lengths of wires are limited by 2.

5 Restricting the Lengths of Wires

A wire routing problem may involve constraints on the lengths of some of the wires—that is to say, on signal delays through them. The approach to wire routing proposed in this paper allows us to express such constraints by simple changes in the goal condition of the planning problem.

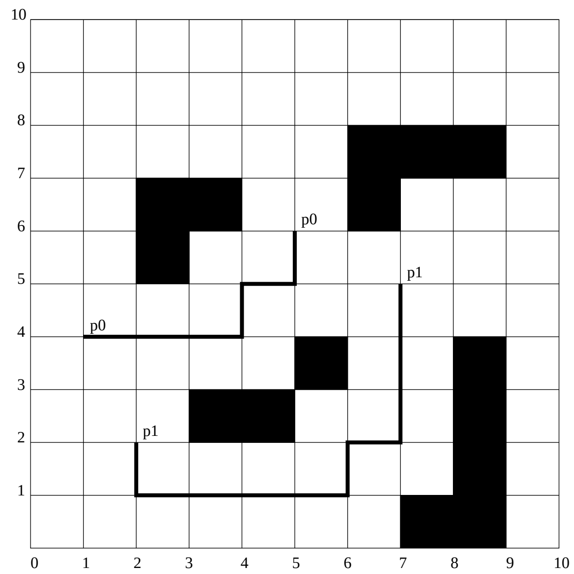

Fig. 9. A solution to a routing problem with 2 wires.

Consider, for instance, the solution to a routing problem with 2 wires shown in Fig. 9. This solution was found by CCALC when the problem was described as in Fig. 10. The length of Wire 1 in this solution is 10. To instruct CCALC to find a solution with the length of this wire limited by 8, we replace the goal in Fig. 10 with

```
8: at(1,7,5),
10: at(0,5,6).
```

The solution found by CCALC after this change is shown in Fig. 11.

We can also restrict the total length of all wires—a parameter that measures the overall quality of the solution. To this end, we need to introduce auxiliary fluents `length(N,L)` ("the current length of the path of Robot `N` equals `L`"). We assume that initially this length is 0, and postulate that `move(N,D)` causes it to increase by 1. Then the requirement that the combined length of Wires 0 and 1 be limited by `maxTotalLength` can be expressed by

```
:- macros k -> 1;
          maxX -> 10;
          maxY -> 10;
          maxLength -> 14.

:- include 'obstacles2.t'.
:- include 'routing.t'.

:- plan
facts::
0: (occupied(N,XC,YC) ->> at(N,XC,YC)),
0: at(0,1,4),
0: at(1,2,2);
goal::
10: (at(0,5,6) && at(1,7,5)).
```

Fig. 10. Input file for the problem from Fig. 9

```
never (\/L0: \/L1: (length(0,L0) && length(1,L1) && L is L0+L1
                                         && L >= maxTotalLength)).
```

The symbol \/ represents the existential quantifier (over a finite domain) and is expanded by CCALC into a finite disjunction.

6 Spacing Constraints

We say that two wires in a solution to a routing problem are *adjacent* if a segment of one of them and a segment of the other form two opposite sides of a unit square. In Fig. 2, for instance, Wires 0 and 1 are adjacent, and Wires 2 and 3 are adjacent. In this section we consider the problem of finding a wire routing without adjacent wires. This is a simple spacing constraint, interesting in view of its relation to the problem of avoiding signal interferences.

To describe adjacency, we introduce auxiliary fluents that represent the positions of vertical and horizontal unit segments in the trajectory of every robot. Fluent `in_v(N,XC,YC)` holds if the part of the trajectory of Robot `N` constructed so far includes the segment connecting points `(XC,YC)` and `(XC,YC+1)`. Initially, these fluents are identically false. They are affected by actions `move(N,up)` and `move(N,down)` as follows:

```
move(N,up) causes in_v(N,XC,YC) if at(N,XC,YC).
move(N,down) causes in_v(N,XC,Y)
             if at(N,XC,YC) && Y is YC-1 && YC>0.
```

Once such a fluent becomes true, it remains true:

```
caused in_v(N,XC,YC) after in_v(N,XC,YC).
```

Fluents `in_h(N,XC,YC)` describe the positions of horizontal segments in a similar way.

Using these fluents, we can eliminate adjacent wires by postulating

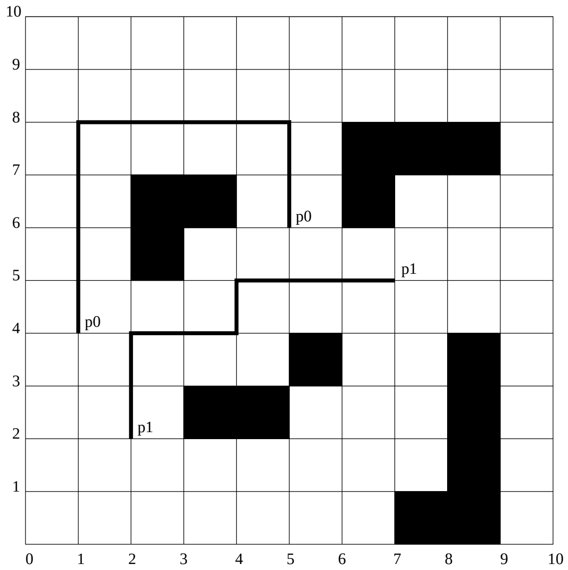

Fig. 11. A solution to the problem from Fig. 9 with the length of Wire 1 limited by 8.

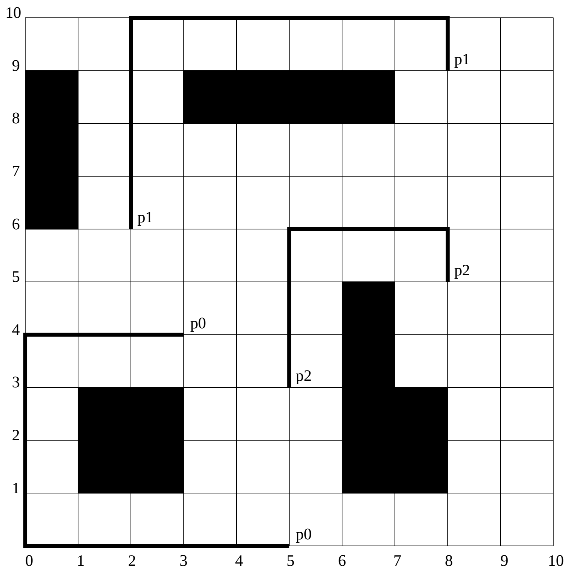

Fig. 12. A solution to a routing problem without adjacent wires.

```
never in_h(N,XC,YC) && in_h(N1,XC,Y)
      && -(N=N1) && Y is YC+1 && YC < maxY.
never in_v(N,XC,YC) && in_v(N1,X,YC)
      && -(N=N1) && X is XC+1 && XC < maxX.
```

Fig. 12 shows a solution to a routing problem, with adjacent wires prohibited, that was generated by CCALC on the basis of such a formalization. In this example, the run time of RELSAT was 156 seconds.

7 Discussion

We showed how satisfiability planning can be applied to wire routing problems of several kinds. Action language $\mathcal{C}$ that we use to describe the effects of actions in the routing domain is much more expressive than older action description languages STRIPS and $\mathcal{A}$ (see [3, Sections 3–6] for references and comparisons). Its expressivity is essential for our purposes.

CCALC transforms planning problems in the routing domains into propositional satisfiability problems, and RELSAT serves as the search engine. In some of our experiments, propositional solver SATO [16] was used instead of RELSAT. On some routing problems it performed much worse than RELSAT, and never much better. In our first example, for instance, RELSAT found a solution after about 1 minute of computation, and SATO did not terminate after 2 hours.

The new approach to wire routing always correctly determines whether a given problem is solvable, and it always produces a solution if it exists. Its other attractive feature is that some enhancements of the basic problem—in which lengths of wires and distances between them come into play—can be easily represented by modifying goals or by adding auxiliary fluents. The CCALC input files for all examples discussed in this paper `include` the same file `routing.t` describing the effects and executability of actions in the routing domain. In this sense, our representation method is similar to the work on elaboration tolerance [10] described in [8].

On the negative side, the size of the grid used in our examples is much too small for serious applications. Investigating the applicability of the new routing method to larger problems is a topic for future work.

There are several other possible future directions that we can take. One is to extend the current approach to perform routing on multiple wiring layers. In this paper we only addressed planar routing (i.e., only one layer is available for wiring). Another direction is to consider more complex spacing constraints where adjacent wires are allowed but the total amount of adjacencies between each pair of wires should be bounded. This more general formulation captures the fact that small amount of adjacencies may not produce enough signal interferences to affect circuit performance. Finally, we plan to investigate how to extend our new routing approach to solve the "global routing" problem [7], which is the problem of determining the connections between adjacent regions after an VLSI chip is decomposed into an array of smaller rectangular regions. The global routing problem resembles the routing problem we studied in this paper except that we would allow more than one wire to be placed on a grid edge.

Acknowledgments

We are grateful to Emilio Remolina and Hudson Turner for comments on a draft of this paper. The work of the first two authors was partially supported by the National Science Foundation under Grant No. IIS-9732744. The work of the third author was partially supported by a grant from the Intel Corporation and by the Texas Advanced Research Program under Grant No. 003658288.

References

1. Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. IJCAI-97*, pages 203–208, 1997.
2. Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proc. AAAI-90*, pages 524–530. AAAI Press, 1990.
3. Michael Gelfond and Vladimir Lifschitz. Action languages.[5] *Electronic Transactions on AI*, 3:195–210, 1998.
4. Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, pages 623–630. AAAI Press, 1998.
5. Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363, 1992.
6. C. Y. Lee. An algorithm for path connections and its application. *IRE Transactions on Electronic Computers*, EC–10:346–365, 1961.
7. T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Design*. John Wiley & Sons, 1990.
8. Vladimir Lifschitz. Missionaries and cannibals in the Causal Calculator. In *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int'l Conf.*, pages 85–96, 2000.
9. Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.
10. John McCarthy. Elaboration tolerance.[6] In progress, 1999.
11. T. Ohtsuki. Maze-running and line-search algorithms. In T. Ohtsuki, editor, *Layout Design and Verification*, chapter 3. Elsevier Science Publishers, 1986.
12. Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
13. Semiconductor Industry Association. The national roadmap for semiconductors, 1997.
14. T. G. Szymanski. Dogleg channel routing is NP-complete. *IEEE Transactions on Computer-Aided Design*, 4(1):31–41, 1985.
15. Hudson Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.
16. Hantao Zhang. SATO: An efficient propositional prover. In *Proc. CADE-97*, 1997.

[5] `http://www.ep.liu.se/ea/cis/1998/016/` .
[6] `http://www-formal.stanford.edu/jmc/elaboration.html` .