中華民國八十四年全國計算機會議論文集
Proceedings of National Computer Symposium 1995

圖形理論的敘述模型
A Specification Model for Graph Theory

李聰　　賴啓超
Tsung Lee and Chi-Chia Lai

中山大學電機工程研究所
Department of Electrical Engineering
National Sun Yat-Sen University
Kao-Hsiung, Taiwan, ROC

摘　要

在此篇論文中 我們描述圖形的基本性質以建立圖形模型. 我們將圖形理論之基本性質分類為彼此獨立的陳述, 可以逐漸地因應各種圖形性質上的需求, 而加入對應的陳述, 例如, 原本無向圖之模型加入方向性的陳述, 即成為一有向圖的模型. 因此, 無向圖, 有向圖, 及超圖, 皆可使用同一基本模型產生各自的描述模型. 根據這些建立的模型, 我們發展了基本圖形理論的知識庫, 並驗証這些知識的完整性及正確性, 這些經驗証的知識已被使用於圖形理論的自動証明器, 証明基本的圖形定理.
關鍵詞: 圖形敘述模型, 圖形理論的自動推理.

Abstract

In this research, we construct graph models by describing the basic properties of graphs for specifying various types of graph objects. By characterizing basic graph properties into orthogonal descriptions, graph models for undirected graphs, digraphs, and hypergraphs can be unified in a generic graph model. With these graph models, a knowledge base of basic definitions of graph theory was constructed to form the basis of a graph specification system. The specification system was proved to be a terminating, confluent, and complete algebraic specification system and can be used in automated reasoning of graph theory.
Keyword: graph specification model, automated reasoning of graph theory.

1 Introduction

Over the decades, graph theory was widely applied in computer system modeling. A unified graph model for all kinds of graphs will be useful in the modeling of complex computer systems. In the past, the specified graph knowledge are restricted in limited classes of graphs. The unification of graph models will bring us the advantages of knowledge compatibility and information exchangability among graph-modeled systems. In this paper, we present an approach to construct a unified graph model for all kinds of graphs and specify graph knowledge in an algebraic specification. With an embedded theorem prover, we can perform reasoning of graph theory in computers.

Previously, Wong [2] constructed a model for digraphs. He applied the model in modeling railway signaling problem. Wong's model is illustrated in Fig. 1.1. In the model,

- A vertex is specified as an abstract type *.
- An edge is specified as a record of two vertices with the abstract type ** representing a weight.
- A graph is specified as the composition of a vertex set and an associated edge set.

```
Vertex = ":*"
Edge = ":(* # * # **)"
Graph = ":(*)set # (* # * # **)set"
```

Fig. 1.1 Wong's model of digraphs

The description is suitable to describe the traversal of digraphs. However, it cannot be directly used to model undirected graphs and hypergraphs.

In [3], Chou constructed a model for undirected graphs. He applied the model in distributed algorithm research. Chou's model is illustrated in Fig. 1.2. In the model,

- Vertex and edge are specified as two abstract types.
- A link is specified as a record for modeling the incidence relations and graph traversals such as walks and paths in undirected graphs. The record consists of two vertices and an edge incident to the vertices. When the vertex ordering

the record is meaningful, it specifies a digraph, otherwise an undirected graph.

- A graph is specified as the composition of a vertex set, an edge set, and a link set.

```
Vertex = ":*"
Edge = ":**"
Link = ":* # ** # *"
Graph = ":(*)set # (**)set # (* # ** # *)set"
```

Fig. 1.2 Chou's model of indirected graph

Chou's model tailors to directed and undirected graphs by specifying two incident vertices of an edge. However, the model cannot be directly used for modeling hypergraphs.

Both models are used to specify the knowledge of limited classes of graphs. In order to develop a universal theorem proving tool for all kinds of graphs, we need to construct a unified graph model. In this research, we developed a generic graph model and a set of derived graph models for this purpose.

In a graph-based modeling system, the specification of various types of graph objects is a fundamental issue. Based on the analysis of graph theory [?], we identify three basic components of a graph which should be specified: the vertex set, the edge set, and the relationships that exist between vertices, between edges, and between vertices and edges. By elaborating these components, we need to design a model to specify the following graph objects : the vertex set, the edge set, the incidence of a vertex and an edge, the adjacency of vertices, the adjacency of edges, and the traversal in a graph, such as walks. A graph model satisfying above specification requirements can provide us a complete view of basic graph properties of the modeled graphs. Therefore, such a graph model can be used in specifying the knowledge in graph theory as the basis for proving graph theorems.

In order to represent various types of graphs in a unified graph model, from an analysis on a graph theory, the graph properties of different types of graphs should be decoupled into orthogonal descriptions for incrementally specifying the properties specific to the corresponding type of graphs. The analysis results in the requirements of a unified graph model listed in the following:

1. The set of vertices should be independently modeled.
2. The set of edges should be independently modeled.
3. The incidences of vertices and edges should be independently modeled.
4. The direction of edges should be independently modeled in order to apply the model for both undirected graphs and digraphs.
5. The number of incidences of an edge should not be fixedly specified in the unified model.
6. The traversal in a graph structure should be independently represented.

These requirements form the basis for designing incrementally specifiable graph models. In the following, we introduce a generic graph model and the associated derived graph models as a solution satisfying these requirements incrementally for specifying various types of graphs.

2 Graph Models

In this section, we describe the graph models by specifying basic graph components, integrity constraints, and graph properties in abstract data types. By characterizing basic graph properties, graph models for undirected graphs, digraphs, and hypergraphs can be unified in a generic graph model and the associated derived graph models.

Approach

A graph model can be specified by describing the basic components and integrity constraints of the represented graph objects. The construction of several graph models is illustrated in Fig. 2.1. A generic graph model is used to specify the graph components, the graph properties, and integrity constraints common to all kinds of graphs. By decomposing these graph descriptions into orthogonal dimensions as described in introduction, each type of graphs can be modeled with additional graph descriptions to form derived graph models.

In this approach, we capture all the required orthogonal component descriptions in a basic model called the generic graph model referred by other derived graph model. In addition, the common validity properties for all kinds of graphs in this description method form the basic integrity constraints which are also recorded in the generic graph model. In order to represent each specific type of graphs, we can refer to the component description and basic integrity constraints, and add the additional integrity constraints for the validity testing of the type of graphs. Therefore, we obtain the following categories of graph models.

1. The generic graph model is specified with
   - basic graph components and their access function for vertices, edges, terminals, vertex sets, edge sets, terminal sets, and graph objects.
   - basic integrity constraints common to all kinds of graphs:
     The integrity constraints enforce the relationships between abstract objects that all types of graphs should possess. Thus,

the illegal combinations of abstract objects in specifying graphs are excluded. The types of integrity constraints include naming identity for graph objects, existence consistency between graph objects, and attribute range restrictions.

2. The model of undirected graphs is specified with
   - the specification of generic graph model
   - an integrity constraint: The number of incidences of an edge is two.
3. The model of digraphs is specified with
   - the specification of generic graph model.
   - an additional access function direction for terminal
   - two integrity constraints : (1)The number of incidences of an edge is two. (2)One incident terminal of an edge is with the incident_from direction attribute, and the other is with incident_to direction.
4. The model of hypergraphs is specified by identity of the generic graph model.
   - the specification of the generic graph model.

In the following, we describe the construction of generic graph model.

```
Generic Graph Model
    objects: graph,vertex,edge,terminal,traversal edge
    access function: VertexSet(g),terminal_vertex(t),....
    derived function: IncidentVertexSet(g,e),IncidentEdgeSet(g,v),...
    integrity constraints:
      naming identity: v1 != v2 <-> vertex_name(v1) !=vertex_name(v2)
      existence relationships: ALL t:TerminalSet(g). terminal_edge(t):EdgeSet(g).
      attributes range restriction: ALL t:TerminalSet(g).
          terminal_direction(t):{incident_from,incident_to,undefined}
Graph Models:
    model of generic graph (MGG)
    model of hypergraph (MHG)
    model of directed hypergraph (MDHG)
    model of simple hypergraph (MSHG)
    model of simple directed hypergraph (MSDHG)
    model of simple edge-directed hypergraph (MSEDHG)
    model of pseudograph (MPG)
    model of multigraph (MMG)
    model of undirected graph (MUG)
    model of simple undirected graph (MSUG)
    model of digraph (MDG)
    model of simple digraph (MSDG)
(1)two_incident_edge constraint:
    ALLe:EdgeSet(g). two_incident_edge(g,e)
(2)loopless constraint:
    ALL e:EdgeSet(g). Loopless(g,e)
(3)unique_edge constraint:
    EX e1:EdgeSet(g). EX e2:EdgeSet(g). IncidentVertexSet(g,e1)=IncidentVertexSet(g,e2) --> e1=e2
(4)direction constraint:
    ALL e:EdgeSet(g). EX t1:edge_terminal_set(g,e). EX t2:edge_terminal_set(g,e).
        ~(t1=t2) & terminal_direction(t1)=incident_from & terminal_direction(t2)=incident_to))
(5) directed constraint:
    ALL t:TerminalSet(g). terminal_direction(t):{incident_from,incident_to}
where (1) and (4) imply (5)
```

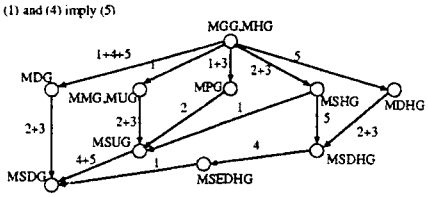

Fig 2.1 construction of graph models

Generic Graph Model

Our view of graphs for designing the generic graph model is illustrated in Fig. 2.2. The basic graph components and access functions of the generic model are designed for specifying graphs and satisfying the requirements in both section one. These graph model components are summarized in the following:

```
basic objects: graph,vertex,edge,terminal,traversal edge
Attributes:(access functions)

graph: VertexSet
       EdgeSet
       TerminalSet

vertex: vertex_name
        vertex_label

edge:edge_name
     edge_label

terminal:terminal_name
         terminal_vertex
         terminal_edge
         terminal_direction

traversal-edge:traversal_source
               traversed_edge
               traversal_destination
```

Fig.2.2 The generic graph model

- basic objects: graphs, vertices, and edges.
- object for modeling the incidence relation: terminals. This description specifies the relationships between vertices and edges.
- object for modeling graph traversal: traversal-edges. This description is used to specify walks, and paths.

In the generic graph model, we obtain some distinct descriptive features which unify the models for different types of graphs with the generic graph model. The distinct descriptive features of the generic graph model are as followed:

1. Terminals for the incidence and the directions of edges

   In order to unify the descriptions of the incidence of vertices and edges in simple graphs and hypergraphs and the associated edge direction, we introduce the **terminal** concept to directly describe the incidence relationship between vertices and edges.

   - A terminal represents a link between a vertex and an edge.
   - A terminal has access functions terminal_vertex(t) and terminal_edge(t) to record the incident vertex and the incident edge. Two derived functions, edge_terminal_set(g,e) for edge and vertex_terminal_set(g,v) for vertices, can be used to capture the incidence relation from another abstract view. The incidence function between vertices and edges, Incident(g,v,e), is specified also

with the terminal concept. These derived specifications will be given in section three.

- terminal_direction(t) records the incidence direction on an edge : It is either **incident_from** or **incident_to** in model of digraphs. For an edge in a digraph, its edge_terminal_set consists of two terminals that one is a starting (incident_from), and the other is a leaving (incident_to) terminal. The two terminals thus connect their two incident vertices via the edge with the direction specified in terminals. It would be specified to undefined in model of undirected graphs. The situation in hypergraph is similar.

2. Flexible value of edge incidences

   Since the terminal is independently specified, the incidences of an edge is not fixed to two. Thus, we can extend the generic graph model to a model of hypergraphs.

3. Traversal graph for describing graph traversal

   A traversal graph is invented to record the traversal in a graph. A traversal graph is a directed graph(hypergraph). The directed edges in the traversal graph record the traversal which enters the vertex traversal_source(te), passes through the edge traversed_edge(te) and leaves at vertices traversal_destination(te) in the source graph. With these attributes, the specification of structures, walks, paths and various graph traversals are viable.

4. Unique representation for graphs

   In Chou's model, when undirected graphs are specified, the two vertices in an edge record form an ordered combination. Each edge thus has two possible representations. For a graph, the redundancy results in $2^{|E|}$ different representations. Chou uses a predicate almost-equal to test for edge equivalence which is an external checking for edge equivalence. In this model, edge equivalence checking is contained in the underlying set theory and hence is transparent in the formulation.

5. Integrity constraint checking for graph validity verification

   Since the orthogonal descriptions of graphs span data space larger than the space of any specific kind of graphs, integrity constraints can be used to specify the required properties among orthogonal descriptions, and hence define the valid space of the the type of graphs. Integrity constraints can thus be checked to verify the validity of specific type of graphs given graph specifications or instances. Once integrity constraints are verified, all graph operations and knowledge inferences can be applied safely.

The generic graph model should capture the common structure, properties, and definitive integrity constraints for all types of graphs. In the following, we describe the orthogonal composition of structures and properties, and the definitive integrity constraints for the generic graph model.

- orthogonal composition for graph description

  The specifications for graph theory of various types of graphs can be constructed by giving the orthogonal basic functions. These basic functions are :

  - access functions of basic objects
    * graph: VertexSet(g), EdgeSet(g), and TerminalSet(g).
    * vertex: vertex_name(v) and vertex_label(v).
    * edge: edge_name(e) and edge_label(e).
    * terminal: terminal_name(t), terminal_vertex(t), terminal_edge(t), and terminal_direction(t)
    * traversal: traversal_source(te), traversed_edge(te), and traversal_destination(te).

  - derived functions
    * incident relation: Incident(g,v,e)
    * incident-vertex set concept: IncidentVertexSet(g,v)
    * incident-edge set concept: IncidentEdgeSet(g,v)
    * adjacent vertex relation: Adj_v(g,v)
    * adjacent edge relation: Adj_e(g,e)
    * adjacent-vertex set concept: AdjacentVertexSet(g,v)
    * adjacent-edge set concept: AdjacentEdgeSet(g,e)

- integrity constraints

  Integrity constraints are used to specify the conditions that a modeled data represents a valid graph. The constraints contains the correlations between basic objects, special constraints of an object(direction limitation), constraints to prevent ambiguity, and the identity of objects.

  - naming identity of vertices, edges, and terminals: For example, the constraint

    ALL v1 : VertexSet(g).
    ALL v2 : VertexSet(g).
    v1 ≠ v2 →
    vertex_name(v1) ≠ vertex_name(v2)
    (1)

    represents that the vertex object is identified by vertex_name(v).

- existence
  consistency between VertexSet(g), EdgeSet(g), and TerminalSet(g).
  * existence relationships between vertices, edges, and terminals: For example, the constraint

    ALL t : TerminalSet(g).

    (2) **terminal_edge(t) : EdgeSet(g)**

    represents that the edges incident with any terminals in terminal set must be an edge in the edge set.
  * existence uniqueness: For example,

    ALL t : TerminalSet(g).
    (EX v1 : VertexSet(g).
    EX v2 : VertexSet(g).
    t : vertex_terminal_set(g, v1) &
    t : vertex_terminal_set(g, v2))
    (3) **→ v1 = v2**

    represents the same terminal should be mapped to the same vertex.
- attribute range restriction: For example, the constraint

  ALL t : TerminalSet(g).
  terminal_direction(t) :
  {incident_from, incident_to,
  (4) **undefined}**

  represents the possible incidence relation represented by terminals is one of the three types.

When all constraints are satisfied, the modeled data is a valid graph in defined space. We will illustrate the extension of our generic model to several kinds of graphs in next section.

3 Graph Specification

The specification of graph knowledge is organized at two levels: graph model level(GML) and graph specification level(GSL). The specifications are specified at the two levels hierarchically to define the graph models and the high-level graph knowledge. We use the generic theorem prover Isabelle [?] in building the graph models. The HOL theory in Isabelle implements higher-order logic [?]. It's based on Gordon's HOL system, which itself is based on Church's original paper [?]. We choose higher-order logic as the specification language.

The higher-order logic inference rules are supported in Isabelle. The natural number theory, set theory and list definition are separately specified in higher order logic. These theoretical and specification foundations form the basis for our specifications of the graph models.

Graph Knowledge Hierarchy

From an analysis of graph theory, the graph knowledge can be specified hierarchically. We designed a two-level organization which consisting of graph model level and graph specification level. At the graph model level, the basic level of graph concepts, graph construction operations, and definitions of models are specified. At the graph specification level, some high level graph properties, graph construction operations, and graph theorems are specified. The specified graph knowledge hierarchy is illustrated in Fig. 3.1. In the following, we describe the model in details.

1. graph model level(GML): At this level, the specified knowledge includes
   - model description: primitive functions and derived functions are described at this stage.
     - primitive functions: VertexSet(g), EdgeSet(g), TerminalSet(g), vertex_name(v), etc. for access functions of basic objects in the generic graph model.
     - derived functions: vertex_terminal_set(g,v), IncidentVertexSet(g,v), Incident(g,v,e), etc. for derived functions for describing the relations between vertices and edges.
   - model operation: basic construction and destruction operations for graph are described at this stage, such as addvertex(g,v).
   - model definition: graph models are described at this stage, such as Generic_Graph(g), Hyper_Graph(g), etc.
2. graph specification level (GSL): At this level, the specified knowledge includes
   - object specification: Some graph structures and graph properties are described at this stage, such as Is_walk(g,w), Is_path(g,w), subgraph(g1,g2),etc.
   - object operation: Some more complex construction and destruction operations for graph are listed at this stage. Such as graph contraction operation.
   - high-level knowledge: Further descriptions of graph knowledge will be included at this stage. Such as Even_total_degree theorem.

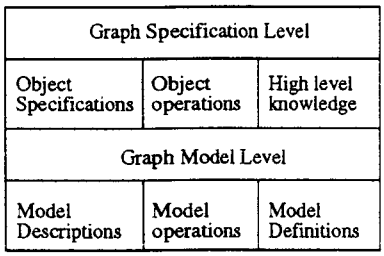

Fig. 3.1 Graph knowledge hierarchy

In the following, we describe the specifications of the derived functions in the generic graph model, and those of the derived graph models.

Derived Functions in the Generic Graph Model

Based on the basic functions of graph models, we further specify the relationships between vertices and edges. In the following, we list these relationships, and the corresponding specifications in HOL description.

- The derived function *vertex_terminal_set* is a derived access function for the collection of terminals incident to the vertex.

```
vertex_terminal_set(g, v) ==
    {t. t : TerminalSet(g) &
(5)     v = terminal_vertex(t)}
```

- The derived function *edge_terterminal_set* represents the collection of terminals that are incident to the edge.

```
edge_terminal_set(g, e) ==
    {t. t : TerminalSet(g) & e =
(6)     terminal_edge(t)}
```

- The predicate *Incident* represents the incidence of a vertex and an edge in the graph g.

```
Incident(g, v, e) ==
  (EX t : TerminalSet(g). v : VertexSet(g)&
        e : EdgeSet(g) &
      v = terminal_vertex(t)) &
(7)   e = terminal_edge(t))
```

- The derived function *IncidentVertexSet* represents the incident vertex set of an edge.

```
IncidentVertexSet(g, e) ==
(8)     {y. Incident(g, y, e)}
```

- The derived function *IncidentEdgeSet* represents the incident edge set of a vertex.

```
IncidentEdgeSet(g, v) ==
(9)     {y. Incident(g, v, y)}
```

- The predicate *Adj_v* represents two vertices are incident to a common edge.

```
Adj_v(g, v1, v2) ==
      (EX e : EdgeSet(g).
    v1 : IncidentVertexSet(g, e) &
(10)  v2 : IncidentVertexSet(g, e))
```

- The predicate *Adj_e* represents two edges are incident to a common vertex.

```
Adj_e(g, e1, e2) ==
      (EX v : VertexSet(g).
    e1 : IncidentEdgeSet(g, v) &
(11)  e2 : IncidentEdgeSet(g, v))
```

- The derived function *AdjacentVertexSet* represents the adjacent vertices of a vertex.

```
AdjacentVertexSet(g, v) ==
(12)    {y. Adj_v(g, v, y)}
```

- The derived function *AdjacentEdgeSet* represents adjacent edges of an edge.

```
AdjacentEdgeSet(g, e) ==
(13)    {y. Adj_e(g, e, y)}
```

The above derived formulas enrich the description of the specified basic components. They represent the basic relations between vertices and edges of graph theory.

Derived Graph Models

With the descriptive structure, properties, and integrity constraints for the generic graph model, additional conditions for various types of graphs can be introduced to enhance the generic graph model descriptions into models of other types of graphs. In the following, we include the work for undirected graphs, multigraphs, pseudographs, hypergraphs, and digraphs.

1. Model of (simple) undirected graphs

   In graph theory, the undirected graph is usually taken as a (simple) undirected graph. There are three constraints for simple undirected graphs. These constraints are two-incidence edge constraint, unique edge constraint, and loopless constraint. The relax of constraints will construct some relaxed graphs. The relax of unique edge constraint and loopless constraint construct the multi-graphs. The relax of loopless constraint construct pseudographs. The relax of the three constraints construct hypergraphs.

   - two-incidence constraint: The number of incidences of any edge is equal to two. Thus, the two-incidence constraint is specified as:

```
(ALL e : EdgeSet(g).
(14)    two_incident_edge(g, e))
```

   A predicate two_incident_edge is introduced for specifying two-incidence property of an edge e useful in modeling of undirected graphs.

```
two_incident_edge(g, e) ==
      (e : EdgeSet(g)&
  card(edge_terminal_set(g, e)) =
(15)    Suc(Suc(0)))
```

- unique edge constraint: There exists a unique edge between two vertices. Thus, the unique edge constraint is specified as:

```
EXe1 : EdgeSet(g).EXe2 : EdgeSet(g).
    IncidentVertexSet(g, e1) =
    IncidentVertexSet(g, e2)
(16)        → e1 = e2
```

- loopless constraint: No selfloops exist in the graph. Thus, the loopless constraint is specified as:

```
(ALL e : EdgeSet(g).
(17)    Loopless(g, e))
```

A loopless predicate is introduced for specifying the property that any edge in the graph is not a selfloops.

```
Loopless(g, e) ==
  (ALL t1 : edge_terminal_set(g, e).
   ALL t2 : edge_terminal_set(g, e).
      terminal_vertex(t1) =
      terminal_vertex(t2)
(18)        → (t1 = t2))
```

As mentioned above, there are three constraints for simple graphs. Thus, the simple graph model is specified as:

```
simplegraph(g) ==
        (Generic_Graph(g)&:
        (ALL e : EdgeSet(g).
      two_incident_edge(g, e))&:
  (EX e1 : EdgeSet(g).EXe2 : EdgeSet(g).
    (IncidentVertexSet(g, e1) =
     IncidentVertexSet(g, e2))
          → (e1 = e2))&:
(19)  (ALLe : EdgeSet(g).Loopless(g, e)))
```

The undirected graphs commonly used in graph theory refer to simple graphs. Therefore, the definition of undirected graphs equivalent to that of simple graphs.

```
(20)        Graph(g) == simplegraph(g)
```

2. Models of (relaxed) undirected graphs

By relaxing the simple graph constraints, other graph models can be derived. Some of them are multi-graphs, pseudographs, and hypergraphs.

A multi-graph model is a kind of graphs in which there can be multiple-edges between two vertices. Thus, the multi-graph model is specified by relaxing unique edge constraint and loopless constraint.

```
Multi_Graph(g) ==
    (Generic_Graph(g) &:
    (ALL e : EdgeSet(g).
(21)    two_incident_edge(g, e)))
```

A pseudograph model is a kind of graphs in which there can be with selfloops but without multiple-edges. Thus, the pseudograph model is specified by relaxing loopless constraint.

```
Pseudo_Graph(g) == (Generic_Graph(g) &:
  (ALL e : EdgeSet(g). two_incident_edge(g, e))&:
    (EXe1 : EdgeSet(g).EXe2 : EdgeSet(g).
      (IncidentVertexSet(g, e1) =
       IncidentVertexSet(g, e2))
(22)        → (e1 = e2)))
```

A hypergraph model is specified by the generic graph model. The simple hypergraph model is specified by the generic graph model with the unique edge constraint and loopless constraint.

```
Hyper_Graph(g) ==
(23)        Generic_Graph(g)
Simple_Hyper_Graph(g) ==
        Generic_Graph(g)&:
  (EXe1 : EdgeSet(g).EXe2 : EdgeSet(g).
    (IncidentVertexSet(g, e1) =
     IncidentVertexSet(g, e2))
          → (e1 = e2))&:
(24)  (ALLe : EdgeSet(g).Loopless(g, e))
```

3. Model of directed graphs

In the general undirected graph model, the multi-graph model, when each edge is required to be directed, we obtain the general directed graph model. Hence, there are two constraints in the general directed graph model. They are the two-incidence constraint and the direction constraint. When we add the loopless constraint and the unique edge constraint in the general directed graphs, we get the simple directed graph model for commonly used directed graphs in graph theory.

- direction constraint: In directed graphs, the direction of edge is from one vertex to the other. That is, the incidence relationships of the one terminal of the edge must be incident_from, that of the other must be incident_to. Thus the direction constraint is specified as:

```
ALL e : EdgeSet(g).
      EX t1 : edge_terminal_set(g, e).
      EX t2 : edge_terminal_set(g, e).
            ~ (t1 = t2)&:
  terminal_direction(t1) = incident_from &:
    terminal_direction(t2) = incident_to
(25)
```

A digraph model is a kind of graph in which the edge is directed. The incidence of one end vertex with the edge is incident_from and another

one is incident_to.

Digraph(g) ==
(Generic_Graph(g) &
(ALL e : EdgeSet(g).
two_incident_edge(g, e)) &:
(ALL e : EdgeSet(g).
EX t1 : edge_terminal_set(g, e).
EX t2 : edge_terminal_set(g, e).
~ (t1 = t2)&:
terminal_direction(t1) = incident_from &:
terminal_direction(t2) = incident_to))
(26)

The addition of loopless constraint and unique edge constraint on the specified directed graphs constructs simple directed graphs.

Simple_Digraph(g) == (Digraph(g)&:
(ALLe : EdgeSet(g).Loopless(g, e)&:
(EXe1 : EdgeSet(g).EXe2 : EdgeSet(g).
(IncidentVertexSet(g, e1) =
IncidentVertexSet(g, e2))
(27) **→ (e1 = e2))**

4 A Proving Example

Since the specification in HOL is supported in a generic theorem prover Isabelle, graph theorems can be specified and proved. For the consistency verification of the specified model and the corresponding mathematical model, several model assertions are mechanically proved in the system. The script of an example proof is as followed:

-val inc_nvs = goal gen.thy "Incident(g,v,e) = (v:IncidentVertexSet(g,e))":
-by (rewrite_goals_tac [IncidentVertexSet_def]):
-by (fast_tac set_cs 1):
No subgoals!
-val inc_nvs =result():

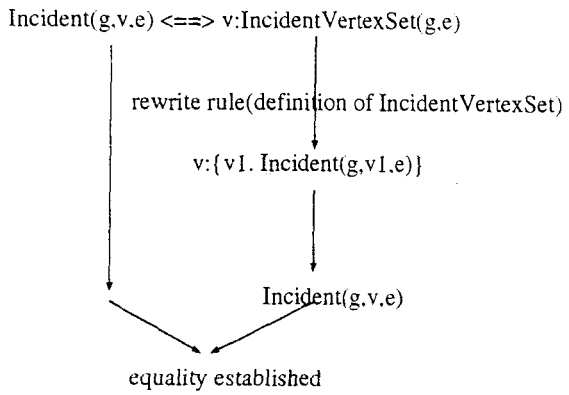

Fig. 4.1 equality testing by term rewriting

The goal is to prove that the incidence of a vertex and an edge is equivalent to the membership of a vertex in the incident-vertex set of an edge in a graph. By expanding the definition of incident-vertex set of an edge and applying depth-first search of inferences, the result "No subgoals" is obtained. It means that all subgoals are verified. Hence, the goal statement is proved.

5 Conclusion

In this research, we designed a set of graph models based on the orthogonal graph descriptions. A set graph knowledge is specified for the graph models and higher level graph properties in HOL algebraic specifications. In the development of the graph models, we introduced concepts of the generic and derived graph models which consist of orthogonal descriptions. In addition to vertices and edges, terminals and traversal-edges are introduced as new graph components for the orthogonal description of graphs. Different from other graph specification models, the orthogonal property of the developed graph model forms the basis of a unified graph model for various types of graphs. The unification of graph models can bring us the advantages of knowledge compatibility and information exchangability among graph-modeled systems.

References

[1] Ronald Gould, *Graph Theory*. The Benjamin/Commings publishing Company, 1988.

[2] Wai Wong, "A Simple Graph Theory and Its Application in Railway Signalling", *HOL Theorem Proving System and its Application*, IEEE Computer Society Press, 1992.

[3] Ching-Tsun Chou, "A Formal Theory of Undirected Graphs in Higher-Order Logic", *Higher Order Logic Theorem Proving and Its Applications. 7th International Workshop. LNCS 859*, Springer-Verlag, 1994.

[4] Lawrence C. Paulson, *Isabelle: A Generic Theorem Prover*, LNCS 828, Sprin ger-Verlag, 1994.

[5] M. J. C. Gordon, and T. F. Melham, *Introduction to HOL: A Theorem Proving E environment for Higher Order Logic*, Cambridge Univ. Press, 1993.

[6] A. Church, A formulation of the simple theory of types, *J. Symb. Logic, 5, pp56-68*, 1940.