State Isomorphism in Model Programs with Abstract Data Structures[⋆]

Margus Veanes[1], Juhan Ernits[2], and Colin Campbell[1]

[1] Microsoft Research, Redmond, WA, USA
{margus,colin}@microsoft.com,
[2] Inst. of Cybernetics / Dept. of Comp. Sci.,
Tallinn University of Technology, Tallinn, Estonia
juhan@cc.ioc.ee[⋆⋆⋆]

Abstract. Modeling software features with model programs in C# is a way of formalizing software requirements that lends itself to automated analysis such as model-based testing. Unordered structures like sets and maps provide a useful abstract view of system state within a model program and greatly reduce the number of states that must be considered during analysis. Similarly, a technique called linearization reduces the number of states that must be considered by identifying isomorphic states, or states that are identical except for reserve element choice (such as the choice of object IDs for instances of classes). Unfortunately, linearization does not work on unordered structures such as sets. The problem turns into graph isomorphism, which is known to have no polynomial time solution. In this paper we discuss the issue of state isomorphism in the presence of unordered structures and give a practical approach that overcomes some of the algorithmic limitations.

1 Introduction

Model programs are a useful formalism for software modeling and design analysis and are used as the foundation of industrial tools such as Spec Explorer [24]. The expressive power of model programs is due largely to two characteristics. First, one can use complex data structures, such as sequences, sets, maps and bags, which is sometimes referred to as having a *rich background universe*. Second, one can use instances of classes or elements from user-defined abstract types; we use the word *object* to mean either case.

Taking into account practical experience with Spec Explorer and user feedback, we can characterize a typical usage scenario of model programs as a three step process: *describe*, *analyze* and *test*.

Describe: A *contract model program* is written to capture the intended behavior of a system or subsystem under consideration. Complex data structures and abstract elements are utilized to produce a contract, or trace oracle, at the desired level of abstraction.

[⋆⋆⋆] This work was done during an internship at Microsoft Research, Redmond, WA, USA.
[⋆] Submission to FORTE'07

Analyze: Zero or more *scenario model programs* are written to restrict the contract to relevant or interesting cases. The scenarios are composed with the contract and the resulting model program is *explored* to validate the contract. The possible traces of a composition of model programs is the intersection of possible traces of the constituent model programs.

Test: The model program, that is, the contract possibly composed with additional scenarios, is used to generate test cases or used as a test oracle.

The expressive power of combining abstract, unordered data types with objects is useful when describing a model but complicates analysis. The core problem is to efficiently identify "relevant" states during exploration. It is often the case that two states that are isomorphic should be treated as being equivalent. Isomorphism between states with a rich background universe is well defined. It exists when there is a one-to-one mapping of objects (within each abstract type) that induces a structure-preserving mapping between the states [1].[3] Informally, two states are isomorphic if they differ in choice of object IDs (or elements of the reserve) but are otherwise structurally identical.

Consider for example a state signature containing two state variables V and E. (States are introduced in the next section.) The type of V is a set of vertices (distinct values of an abstract type *Vertex*) and the type of E is a set of vertex sets. Let v_1, v_2, v_3, v_4 be vertices and let S_1 be a state where,

$$\begin{aligned}
V &= \{v_1, v_2, v_3, v_4\},\\
E &= \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_1\}\}.
\end{aligned}$$

Intuitively, the state S_1 is an undirected graph that is a circle of four vertices. Let S_2 be a state where V has the same value as in S_1 and,

$$E = \{\{v_1, v_3\}, \{v_3, v_2\}, \{v_2, v_4\}, \{v_4, v_1\}\}.$$

States S_1 and S_2 are isomorphic because structure is preserved if the reserve element v_2 is swapped with v_3. This is an isomorphism that maps v_3 to v_2, v_2 to v_3 and every other vertex to itself. Let S_3 be a state where V has the same value as in S_1 and,

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_4, v_1\}\}.$$

State S_3 is not isomorphic to S_1, because all vertices in S_1 are connected to two vertices but v_4 is only connected to one vertex in S_3, i.e., there exists no structure-preserving mapping from S_1 to S_3.

The example illustrates the point that state isomorphism is as hard as graph isomorphism, when objects and unordered data structures are combined. A customer survey of Spec Explorer users within Microsoft has shown that this combination occurs often in practice. It occurs in the standard Spec Explorer example included in the distribution [21] known as the chat model [24, 23], where chat clients are objects and the state has a state variable that maps receiving clients to sets of sending clients with pending messages. The state isomorphism problem for reserve elements in unordered structures was

[3] In ASM theory, what we call *objects* are called *reserve elements*.

not solved in Spec Explorer and to the best of our knowledge has not been addressed in other tools used for model based testing or model checking that support unordered data structures. There are model checkers that support *scalar sets* [13], which are basically ranges of integers, but we do not know of instances where such sets can contain objects with abstract object IDs.

The lack of isomorphism checking when states include both unordered structures and objects is a serious practical concern for users of tools like Spec Explorer. If isomorphic states are not pruned, the number of states that must be considered during exploration can become infeasibly large. This problem is known as *state space explosion*.

In practical terms this means that users must either use various pruning techniques that only partially address the problem or extend the model program with custom scenario control that tries to work around the problem by restricting the scope of exploration. The results are not always satisfactory.

The pruning techniques that have been partially helpful in this context are state grouping [9] and multiple state grouping [4, 24]. But the grouping techniques have an orthogonal usage that is similar to abstraction in model checking, whereas state isomorphism is a generalization of symmetry checking in model checking. In general, it is not possible to write a grouping expression that maps two states into the same value if and only if the states are isomorphic; the “only if” part is the problem.

If objects are not used, then state isomorphism reduces to state equality. State equality can be checked in linear time. This is possible because the internal representation of all (unordered) data structures can then be ordered in a canonical way. The same argument is true if objects are used but no unordered data structures are used. Then state isomorphism reduces to what is called heap canonicalization in the context of model checking and can also be implemented in linear time [12, 19].

In this paper we describe a solution for the state isomorphism problem for model programs with states that have both unordered structures and objects. We do so by providing a mapping from model program states to rooted labeled directed graphs and use a graph isomorphism algorithm to solve the state isomorphism problem. The graph construction and the labeling scheme use techniques from graph partitioning algorithms and strong hashing algorithms to reduce the need to check isomorphism for states that are known not to be isomorphic. We also outline a graph isomorphism algorithm that is customized to the particularities of state graphs. Our algorithm extends a symmetry-checking algorithm with backtracking and is, arguably, better suited for this application than existing graph isomorphism algorithms.

Before we continue with the main body of the paper, we illustrate why state isomorphism checking is useful on a small example, shown in Figure 1, that we use also in the later sections. The example is small but typical for similar situations that arise for example in the chat model [23] or when modeling multithreaded applications where threads are treated as objects [26]. The example is written in C# and uses a modeling library and a toolkit called *NModel*. The formal definition of a model program is given in Section 2, where it is also explained how the C# code maps to a model program. NModel is going to be an open source project that supports the forthcoming text book [14] that

```
namespace Triangle
{
  [Abstract]
  enum Side { S1, S2, S3 }

  [Abstract]
  enum Color { RED, BLUE }

  static class Contract
  {
    static Map<Side, Color> colorAssignments = Map<Side, Color>.EmptyMap;

    static bool AssignColorEnabled(Side s)
    {   return !colorAssignments.ContainsKey(s); }

    [Action]
    static void AssignColor(Side s, Color c)
    {   colorAssignments = colorAssignments.Add(s, c); }
  }
}
```

Fig. 1. A model program where a color, either `RED` or `BLUE`, is assigned to the sides of a triangle.

discusses the use of model programs as a practical modeling technique. All algorithms described in this paper have been implemented in NModel.

Example

Let us look at a simple model program that describes ways to assign colors to the sides of a triangle. The model program is given in Figure 1. The triangle in the program has three sides, `S1`, `S2`, and `S3` and each side can be associated with the color `RED` or `BLUE`. The model program has a single action that assigns a color to one side at a time. There are $(|\texttt{Color}| + 1)^{|\texttt{Side}|} = 27$ possible combinations of such assignments, including intermediate steps where some sides have not been colored yet. There are three sides; each side has three possible values if you count “no color” as a value.

The state transition graph visualizing all possible transitions and all distinct states of the triangle program is given in Figure 2. In this case the `[Abstract]` attributes of `Side` and `Color` have not been taken into account and each combination of $\texttt{Side} \mapsto \texttt{Color}$ is considered distinct.

2 Definitions

A formal treatment of model programs builds on the ASM theory [11] and can for example be found in [25]. Here we provide some basic terminology and intuition and illustrate the main concepts with examples. A *state* here is a full first-order state, that is intuitively a mapping from a fixed set of *state variables* to a fixed universe of *values*. States also have a rich *background* [1] that contains sequences, sets, maps, sets of sets,

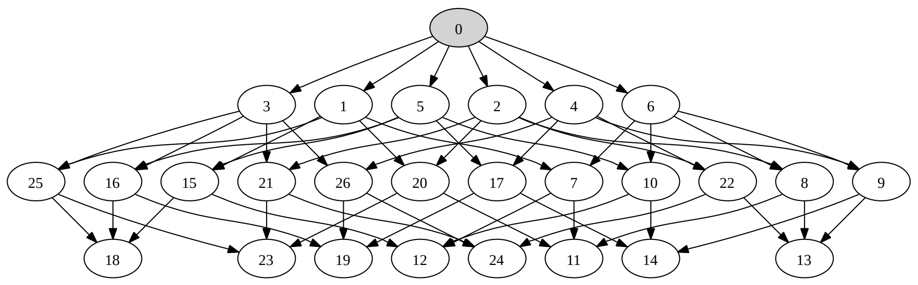

Fig. 2. Full state space of the Triangle example in Figure 1. Each combination of `Side` $\mapsto$ `Color` is considered distinct and thus the blowup of the state space.

maps of sets, etc. We assume here that all state variables are nullary.[4] For example, the model program in Figure 1 has one state variable `colorAssignments`.

Since states have a rich background universe they are infinite. However, for representation, we are only interested in the *foreground* part of a state that is the interpretation of the state variables. All values have a *term* representation. Terms that do not include state variables are called *value terms* and are defined inductively over a signature of function symbols. This signature includes constructors for the background elements.[5] We identify a state with a conjunction of equalities of the form $x = t$, where x is a state variable and t a value term.

The interpretation of a value term is the same in all states. Value terms are not unique representations of the corresponding values, i.e., value terms that are syntactically distinct may have the same interpretation. We say value for a value term when it is clear from the context that the particular term representation is irrelevant.

For example, a set of integers, containing the values `1`, `2`, and `3` is represented by the term `Set<int>(1,2,3)`. The term `Set<int>(2,1,3)` has the same interpretation. We use a relaxed notation where the arity of function symbols is omitted but is implicitly part of the symbol. For example `Set<int>(1,2)` represents a set containing 1 and 2, so the constructor `Set<int>` is binary here and ternary in the previous case. Function symbols are typed. For example, a set containing two sets of strings `Set<string>("a")` and `Set<string>("b")` is represented by the term

```
Set<Set<string>>(Set<string>("a"),Set<string>("b"))
```

Model programs typically also have user-defined types that are part of the background. For example the model program in Figure 1 has the user-defined type `Color`. This type has two elements `Color.RED` and `Color.BLUE`, respectively. The initial state of this model program is (represented by the equality)

[4] In ASM theory, state variables are called *dynamic functions* and may have arbitrary arities. Dynamic functions with positive arities can be encoded as state variables whose values are maps.

[5] In ASM theory, these function symbols are called *static*, their interpretation is the same in all states.

```
colorAssignments = Map<Side,Color>.EmptyMap.
```

A user-defined type may be annotated as being *abstract*, e.g., `Color` in Figure 1 is abstract. Elements of an abstract type are treated as typed *reserve* elements in the sense of [11]. Intuitively this means that they are interchangeable elements so that a particular choice must not affect the behavior of the model program. A valid model program must not explicitly reference any elements of an abstract type. For example, even though the `Color` enumeration type provides an operation to return the string name of a color value, the model program must not use that operation if color is to be considered abstract. Abstract types are similar to objects[6] that are treated the same way.

An *update rule* is a collection of (possibly conditional) assignments to state variables. An update rule p that has formal input parameters $\bar{x}$ is denoted by $p[\bar{x}]$. The instantiation of $p[\bar{x}]$ with concrete input values $\bar{v}$ of appropriate type, is denoted by $p[\bar{v}]$. An update rule p denotes a function $[\![p]\!] : \mathit{States} \times \mathit{Values}^n \rightarrow \mathit{States}$

A *guard* φ is a state dependent formula that may contain free logic variables $\bar{x} = x_1, \ldots, x_n$, denoted by $\varphi[\bar{x}]$; φ is *closed* if it contains no free variables. Given values $\bar{v} = v_1 \ldots, v_n$ we write $\varphi[\bar{v}]$ for the replacement of x_i in φ by v_i for $1 \leq i \leq n$. A closed formula φ has the standard truth interpretation $s \models \varphi$ in a state s. A *guarded update rule* is a pair (φ, p) containing a guard $\varphi[\bar{x}]$ and an update rule $p[\bar{x}]$; intuitively (φ, p) limits the execution of p to those states and arguments $\bar{v}$ where $\varphi[\bar{v}]$ holds.

We use a simplified definition a model program here, by omitting control modes. The state isomorphism problem is independent of the presence of explicit control modes. Thus, this simplification does not affect the main topic of this paper.

Definition 1. A *model program* P has the following components:

– A finite vocabulary $X_<$ of *state variables*
– A finite vocabulary Σ of *action symbols*
– An *initial state* s_0 given by a conjunction $\bigwedge_{x \in X} x = t_x$ where t_x is a value term.
– A *reset* action symbol *Reset* $\in \Sigma$.
– A family $(\varphi_f, p_f)_{f \in \Sigma}$ of guarded update rules.
  • The *arity* of f is the number of input parameters of p_f.
  • The arity of *Reset* is 0 and $[\![p_{\mathit{Reset}}]\!](s) = s_0$ for all $s \models \varphi_{\mathit{Reset}}$.

An *action* has the form $f(v_1, \ldots, v_n)$ where f is an n-ary action symbol and each v_i is a value term that matches the required type of the corresponding input parameter of p_f. We say that an action $f(\bar{v})$ is *enabled* in a state s if $s \models \varphi_f[\bar{v}]$. An action $f(\bar{v})$ that is enabled in a state s can be *executed* or *invoked* in s and yields the state $[\![p_f]\!](s, \bar{v})$.

The model program in Figure 1 has a single action symbol `AssignColor`. The guard of `AssignColor` is given by the Boolean function `AssignColorEnabled`. The action $a =$ `AssignColor(Side.S1, Color.RED)` is enabled in the initial state s_0 because `AssignColorEnabled(Side.S1)` returns true in s_0. The execution of a is s_0 yields the state

[6] By "objects" we mean object IDs. Instance fields associated with objects are considered to be state variables in their own right and not part of any nested structure. In this way, we can consider only global variables without loss of generality.

```
colorAssignments = Map<Side,Color>(Side.S1 ↦ Color.RED).
```

The unwinding of a model program from its initial state gives rise to a labeled transition system (LTS). The LTS has the states generated by the unwinding of the model program as its states and the actions as its labels.

Definition 2. A *rooted directed labeled graph*, G, is a graph that has a fixed root, has directed edges, and contains labels of vertices and edges. Such graph can be formally represented as a triple $G = (v_r, V, E)$ where $v_r \in V$ is the root vertex, V is a set of vertices v that are pairs $v = (id, l_v)$, where id is an identifier uniquely determining a vertex in a graph and l_v is the label of the vertex. E is the set of triples $(v_1.id, l_e, v_2.id)$ where v_1 is the start vertex, v_2 is the end vertex and l_e is the edge label.

3 States as Graphs

In this section we present a graph representation of the state of a model program.

The states of a model program can contain object instances and other complex data structures, thus we do not deal only with primitive types, such as integers and Boolean values, but also with instances of objects that can be dynamically instantiated and refer to other instances of objects. The state space of a model program may be infinite, but concrete states are finite first order structures. We look at the configuration of values and object instances that have been assigned to the fields of objects and data structures contained in the program. We do not consider the the program stack to be part of the program state as states are only compared between performing actions.

A state is defined by an assignment of term representations of values to fields, $s = \bigwedge_{x \in X_<} x = t_x$. There are two kinds of fields in a model program: global fields, like `colorAssignments` in the program on Figure 1 and fields of dynamically instantiated objects.[7] For the sake of brevity, we will look only at states containing global fields. Assignments to global fields are simple equations $x = t$. It is important to note that it is possible to establish a binary relation of total ordering, $<$, of field names. This can be achieved by, for example, ordering the field names alphabetically.

Figure 3 outlines the procedure of creating a graph from a state. In general the procedure is straightforward: the function `CreateGraph` creates the graph by analyzing the terms corresponding to each state variable x. The analysis of a term, `TermToGraph`, adds a field index to value mapping to the label of the parent node, if t denotes a value and adds a new node to the graph, if t is an object. A specialized procedure is used for creating nodes corresponding to built-in abstract data types. In fact, each ADT is handled in a slightly different way.

A `Set` becomes a node that has the count of its elements in the label of the incoming edge. All outgoing edges of a set are given a label starting with a 0, denoting membership in a set. It is possible that the label is extended with more arguments as the set may contain other sets.

[7] An instance field can be represented as a global field whose value is a map of objects to their corresponding field values. In other words, objects don't "contain" structure; instead, they are distinct identities.

```
class State {
  Sequence<Pair> X;
}

class G {
  Vertex v_r = new Vertex();
  Set<Vertex> vertices = Set<Vertex>.EmptySet.Add(v_r);
  Set<Edge> edges = Set<Edge>.EmptySet ;
}

G CreateGraph(State s) {
  g=new G();
  foreach (x in s.X) {
    g = TermToGraph(t,g.v_r, g.SequenceNumber(x), g);
  }
  return g;
}

G TermToGraph(Term t, Vertex parent, int fieldIndex, G g) {
  if (!isObject(t))
   parent.label.Add(new Label(fieldIndex,t));
  switch(t.functionSymbol) {
    case Set:
     Vertex setv=g.NewVertex();
     g.edges.Add(new Edge(parent,new Label(fieldIndex,t.argCount),setv));
     forall (Term elem in t.arguments)
      TermToGraph(elem,setv,0,g);
     break;
    case Bag
     Vertex bagv=g.NewVertex();
     Bag<Pair> bagCounts = Bag<Pair>.EmptyBag;
     forall (Pair<Term,Term> (elem,count) in t.GetArgumentsByPair()) {
      TermToGraph(elem,setv,count,g);
      bagCounts.Add(count);
     }
     g.edges.Add(new Edge(parent,new Label(fieldIndex,bagCounts.Sort()),setv));
      break;
    case Map:
     forall (Pair<Term,Term> (key,val) in t.GetArgumentsByPair()) {
      Vertex maplet=g.NewVertex();
      TermToGraph(key,maplet,0,g);
      TermToGraph(val,maplet,1,g);
       g.edges.Add(new Edge(parent,new Label(fieldIndex,t.GetPairCount()),maplet));
      }
     break;
    default:
     if (isObject(t)) {
       Vertex newVertex=new Vertex();
       g.edges.Add(new Edge(parent, new Label(fieldIndex),newVertex));
       if (arity(t)>0)
        forall (Term arg in t.arguments)
         TermToGraph(arg,newVertex,sequenceNumber(arg),g);
     }
  }
  return g;
}
```

Fig. 3. Pseudo-code of generating a rooted labeled directed graph from a state of a model program

The representation of a bag (or multiset) has a sorted list of element multiplicities on the incoming label. The label of the edge pointing to each element of a `Bag` is labeled by the corresponding multiplicity. In fact, a `Bag` is a set of pairs, and using a specialized representation is an optimization that helps to reduce the number of nodes in the state graph. A `Map` is also a `Set` of pairs but can be converted to a reduced fragment of the graph.

The labelings of outgoing edges may be unique, as in the case of different field indices of a structure, or unordered, as in the case of a set.

Thus, it is possible to classify the outgoing edges of a node into *ordered* and *unordered* edges. The graph representations of the abstract data structures `Set`, `Bag`, and `Map` are summarized in Figure 4.

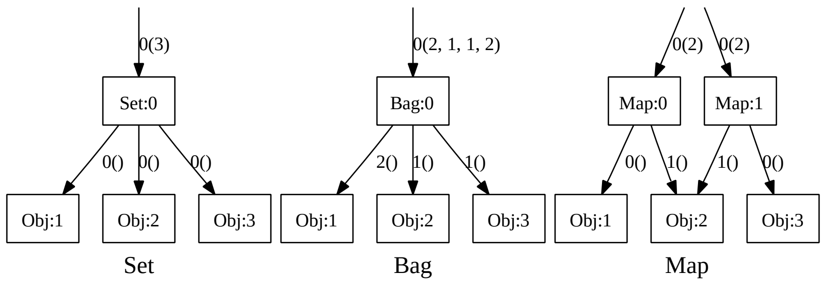

Fig. 4. Graph representations of abstract data types used by model programs.

There are some graphs representing the states of the Triangle example in Figure 5. The state graphs have been generated using the procedure outlined in Figure 3.

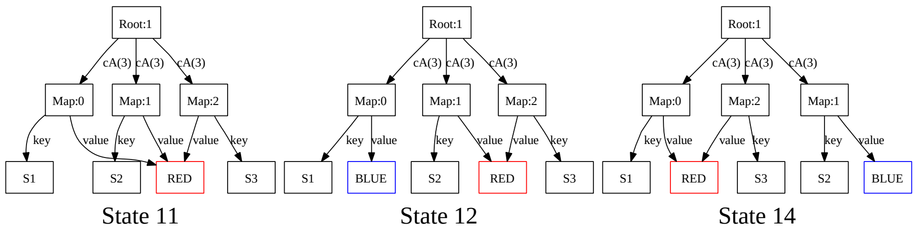

Fig. 5. State graphs of states 11, 12, and 14 of the triangle example on Fig. 2 and Fig. 6. State 14 is isomorphic to state 12 but neither 12 nor 14 is isomorphic to 11. The abbreviation `cA` stands for `colorAssignments` and `(3)` denotes that there are 3 key-value pairs in the map.

State graphs of states 11, 12, and 14 of the triangle example on Figure 2 and Figure 6. State 14 is isomorphic to state 12 but neither 12 nor 14 is isomorphic to 11. The abbreviation `cA` stands for `colorAssignments` and `(3)` denotes that there are 3 key-value pairs in the map.

Figure 6 illustrates the effects of isomorphism-based symmetry reduction applied to the triangle example studied previously. The state graph on the left shows at which stages of the search isomorphic states were encountered. The dashed arrows point to states that are isomorphic to the state the arrow starts from. The graph on the right is obtained by showing a representative example of a family of isomorphic states.

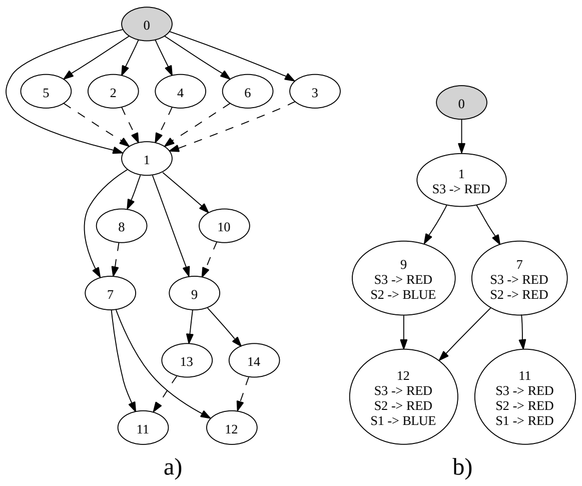

Fig. 6. State space of the Triangle example from Fig. 1, where exploration of isomorphic states has been pruned. The dashed lines on a) exhibit encounters of isomorphic states during exploration. b) exhibits the structure of the state graph when isomorphic states are collapsed.

Field maps

As mentioned earlier, *objects* are just abstract ids or reserve elements. So how do we deal with *fields* of objects? Fields of objects are represented by state variables, called *field maps*, whose values are finite maps from objects of the given type to values of the given field type.[8] From the point of view of this paper, field maps are handled in the same way as map-valued state variables. A difference compared to map-valued state variables is that field maps can not be referenced as values inside of a model program, which can sometimes be used to simplify the graph representation of a state.

In order to illustrate field maps, consider a version of the triangle example, shown in Figure 7, where sides are instances of a class *Side*. The fact that sides are reserve elements is indicated by the base class. This model program has two state variables,

[8] The name of a field map is uniquely determined from the fully qualified name of the class and the name of the field.

```
namespace Triangle
{
  [Abstract]
  enum Color { RED, BLUE }

  class Side : LabeledInstance<Side> { public Color color; }

  static class Contract
  {
    static Set<Side> sides = Set<Side>.EmptySet;

    [Action]
    static void AssignColor([New] Side s, Color c)
    {
       s.color = c;
       sides = sides.Add(s);
    }

    static void AssignColorEnabled(Side s)
    { return sides.Count < 3; }
  }
}
```

Fig. 7. A version of the triangle model where sides are objects. The `AssignColor` action is enabled if not all sides have been colored. The `New` keyword indicates that the side is a new object (reserve element).

`sides` and `color`, where color is a field map. In the initial state, both `color` and `sides` are empty. When a color `c` is assigned to a side `s`, the `color` map gets a new entry `s` $\mapsto$ `c`.

The presented approach is also extended to states resulting in the composition of model programs, as presented in [25]. The root of a state of a composition of model programs becomes a set of two rooted graphs that may share objects.

4 Isomorphism checking

Unlike arbitrary graphs, state graphs are rooted and encode state information in a way that partially reflects the underlying static structure of a program. For example, all objects of a given type have a fixed set of fields that are ordered alphabetically. Several built-in ordered data types, such as sequences and pairs, also have an order of the elements contained in them according to their position. Moreover, user-defined types, other than abstract types, have a fixed alphabetical order of fields. A typical model program uses both ordered and unordered data structures. As explained above, the resulting state graph includes both ordered and unordered edges.

Our intent was to devise an algorithm for graph isomorphism that takes advantage of the ordered edges as much as possible while handling the unordered cases as a last resort through backtracking. The starting point is that all vertices of the graph have been given strong labels through object ID-independent hashing[9] that already reduces the possible pairings of vertices dramatically. In the case when all edges are ordered

[9] The hashing part of the algorithm is outside the scope of this paper.

the algorithm should not do any backtracking at all. The basic idea of the algorithm is an extension of the linearization algorithm used in Symstra [27] with *backtracking*. The algorithm reduces to linearization when the graphs that are being compared are fully ordered, i.e. have no unordered edges. A small difference compared to Symstra is that the linearizations are computed and compared simultaneously for the two graphs as depth first walks, rather than independently and then compared.

Linearization with backtracking

The following is an abstract description of the algorithm. Given are two state graphs G_1 and G_2. The algorithm either fails to produce an isomorphism or returns an isomorphism from G_1 to G_2. The abstract description of the algorithm is non-deterministic. In the concrete realization of the algorithm the **choose** operation is implemented through backtracking to the previous backtrack point where more choices were possible. The details of the particular backtracking mechanism are omitted here.

We say that an edge with label l is an l-edge. The edge labels that originate from ordered background data structures are called *functional*. It is known that for all functional edge labels l and for all nodes x, there can be at most one outgoing l-edge from x. Other edge labels are called *relational*.

Bucketing: Compute a "bucket map" B_i for all nodes in G_i, for $i = 1, 2$. Each node n in G_i with label l is placed in the bucket $B_i(l)$. If either B_1 and B_2 do not have the same labels and the sames sizes of corresponding buckets for all labels then **fail**. Otherwise execute **Extend**$(\emptyset, r_1, r_2)$, where r_i is the root of G_i, for $i = 1, 2$.

Extend(ρ, x_1, x_2)**:** Given is a partial isomorphism ρ and isomorphism candidates x_1 and x_2. If x_1 and x_2 have distinct labels then **fail**, else if x_1 is already mapped to x_2 in ρ the return ρ, else if either x_1 is in the domain of ρ or x_2 is in the range of ρ then **fail**, else let $\rho_0 = \rho \cup \{x_1 \mapsto x_2\}$ and proceed as follows.
Let $l_1, \ldots, l_k$ be the outgoing edge labels from x_1 ordered according to a fixed label-order.[10] For $j = 1, \ldots, k$,
- For $i = 1, 2$, **choose** l_j-edges (x_i, y_i) in G_i for some y_i.
  If **Extend**(ρ_{j-1}, y_1, y_2) fails then **fail**, else let $\rho_j =$ **Extend**(ρ_{j-1}, y_1, y_2).

Return ρ_k.

Notice that the algorithm is deterministic and reduces to linearization when all choices are made from singleton sets. A sufficient (but not necessary) condition for this to be true is when all edge labels are functional. A heuristic we are using in the implementation of this algorithm is that all *functional* edge labels appear before all *relational* edge labels in the label-order that is used in the algorithm.

The implementation of the algorithm has also some optimizations when backtrack points can be skipped, that have been omitted in the above abstract description. One particular optimization is the following. When there are multiple l-edges outgoing from a node x for some fixed relational edge label l, but all of the target nodes of those edges have the same label and degree 1, then an arbitrary *but fixed* order of the edges can be

[10] At this point we know that x_2 must have the same outgoing edge labels in G_2 as x_1 has in G_1 or else x_2 would have a different label than x_1.

chosen that uses the order of the node labels and choice points can be cut. The algorithm bears certain similarities to the practical graph isomorphism algorithm in [16], by using a partitioning scheme of nodes that eliminates a lot of the backtracking. The algorithm has been implemented in NModel.[11]

5 Related work

Two program states, in the presence of pointers or objects, can be considered equivalent if the structure of the logical links between data objects is equivalent while the concrete physical addresses the pointers point to differ, i.e. when the actual arrangement of objects in memory is different due to the effects of memory allocation and garbage collection. This is known as one form of symmetry reduction and has been used in software model checking. The principles of such symmetry reductions have been outlined by Iosif in [12]. One of the key ideas in [12] is to canonicalize the representation of program heap by ordering the heap graph during a depth first walk. The order of outgoing edges (pointers) from a node (for example an object) is given by a deterministic ordering by edge labels (field name and order number, for example position in the array, in the parent data structure). Lack of such ordering would render state comparison to an instance of the graph isomorphism problem, which requires exponential time in the number of nodes in the general case [17]. In [19] Musuvathi and Dill elaborate on Iosif's algorithm to allow incremental heap canonicalization, i.e. take into account that state changes are often small and modify only a small part of the heap, thus it should not be necessary to traverse the whole heap after each state change.

In addition to dSpin [6] where the above mentioned principles were initially implemented, there are several analysis tools specifically targeted for object-oriented software that utilize the approach, for example, *XRT* [10] and *Bogor* [20].

XRT is a software checker for common intermediate language, CIL. It processes .Net managed assemblies and provides means for analyzing the processed programs.

Bogor is a customizable software model checking engine that supports constructs that are characteristic to object-oriented software. Although there is support for using abstract data types, like sets, the underlying state enumeration and comparison engine performs heap canonicalization based on an ordering of object IDs based on the previously mentioned work by Iosif [12].

Korat [3] is a tool for automated test generation based on Java specifications. It also uses the concept of heap isomorphism to generate heaps that are non-isomorphic.

We have layered ASM semantics on top of the underlying programming environment and thus the concrete memory locations have been abstracted by interpreting the program state in the ASM semantics. But in addition to using the concrete data structures, we can declare some types to represent instances of abstract objects and there are some data structures, such as the Set, Map and Bag, that are designed to accommodate such objects, among others.

Symstra [27] uses a technique that linearizes heaps into integer sequences to reduce checking heap isomorphism to just comparing the integer sequence equality. It starts

[11] The proof of the correctness of the algorithm as well some experimental data and comparison with the standard algorithm by Ullmann [22] is going to appear in a technical report.

from the root and traverses the heap depth first. It assigns a unique identifier to each object, keeps this mapping in memory and reuses it for objects that appear in cycles. It extends the previously mentioned approaches [12, 19] in that it also assigns a unique identifier to each symbolic variable, keeps this mapping in memory and reuses it for variables that appear several times in the heap.

In [5] a glass box approach of analyzing data structures is presented. The reductions described therein involve isomorphism-based reductions, but encoding the task requires manual attribution of the data structures to be analyzed. The approach does not present a general way how to handle object-oriented programs containing abstract data types.

Spec Explorer [24, 21] is a tool for the analysis of model programs written in AsmL and Spec#. It is possible in some cases to specify symmetry reductions in Spec Explorer using state groupings but the tool does not have a built-in isomorphic state checking mechanisms.

Graph isomorphism is a topic that has received scientific attention for decades. Ullmann's (sub)graph isomorphism algorithm [22] is a well known backtracking algorithm which combines a forward looking technique. As the algorithm is relatively straightforward to implement, we used it as an oracle for testing purposes.

The algorithm described in Section 4 builds on another well known approach also known as the *Nauty* algorithm, which uses node labelings and partitioning based on such labelings [16].

It is known that there exist certain classes of graphs for which there is a polynomial time algorithm for deciding graph isomorphism. In [15] a method for deciding isomorphism of graphs with bounded valence in polynomial time is presented. The reason why such algorithms are not directly usable in practice is that the polynomial complexity result contains large constants [8].

There are model checkers, such as for example *Mur*ϕ [7] and *Symmetric Spin* [2], that allow modeling using scalar sets [13]. These sets are similar to the sets described in the current paper but they do not have support for abstract object IDs. A survey of symmetry reductions in temporal logic model checking is given in [18].

6 Conclusion

In this paper we showed how state isomorphism for states with both unordered structures and objects may be understood in the context of model programs. We reviewed how the concept of background structures and reserve elements can formalize the meaning of isomorphism for program states. We then described how to represent state as a rooted directed labeled graph so that existing isomorphism algorithms could be applied. Finally, we showed an isomorphism-checking algorithm that takes advantage of the information contained in states with elements drawn from a rich background universe.

The techniques in this paper can be applied in a variety of industrially relevant modeling and testing contexts and are motivated by practical concerns that arose from the industrial use of the Spec Explorer tool in Microsoft.

While this current paper gives a solid notion how program states of object-oriented programs can be viewed as graphs, it also leads to a number of interesting open problems. For example, how can one speed up isomorphism checking for the particular graphs of program states? Would it be useful to describe graph isomorphism as a SAT problem? How could this be accomplished?

As future work, we plan on showing how hashing techniques can be used to improve the performance of isomorphism checks for larger numbers of states.

Acknowledgements

We thank Wolfram Schulte for referring us to the Symstra work and Jonathan Jacky for insightful discussions related to the topics of this paper.

References

1. A. Blass and Y. Gurevich. Background, reserve, and gandy machines. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 1–17, London, UK, 2000. Springer-Verlag.
2. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.
3. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. *SIGSOFT Softw. Eng. Notes*, 27(4):123–133, 2002.
4. C. Campbell and M. Veanes. State exploration with multiple state groupings. In D. Beauquier, E. Börger, and A. Slissenko, editors, *12th International Workshop on Abstract State Machines, ASM'05*, pages 119–130. Laboratory of Algorithms, Complexity and Logic, Créteil, France, March 2005.
5. P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 363–382, New York, NY, USA, 2006. ACM Press.
6. C. Demartini, R. Iosif, and R. Sisto. dspin: A dynamic extension of spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.
7. D. L. Dill. The murphi verification system. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996. Springer-Verlag.
8. S. Fortin. The graph isomorphism problem, 1996.
9. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA'02*, volume 27 of *Software Engineering Notes*, pages 112–122. ACM, 2002.
10. W. Grieskamp, N. Tillmann, and W. Schulte. XRT — exploring runtime for .Net architecture and applications. *Electr. Notes Theor. Comput. Sci.*, 144(3):3–26, 2006.
11. Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, 1995.
12. R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *STTT*, 6(4):302–319, 2004.
13. C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.
14. J. Jacky, C. Campbell, M. Veanes, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2007. Forthcoming.

15. E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982.
16. B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
17. B. T. Messmer. Efficient graph matching algorithms, 1995.
18. A. Miller, A. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3):8, 2006.
19. M. Musuvathi and D. L. Dill. An incremental heap canonicalization algorithm. In P. Godefroid, editor, *SPIN*, volume 3639 of *Lecture Notes in Computer Science*, pages 28–42. Springer, 2005.
20. Robby, M. B. Dwyer, and J. Hatcliff. Domain-specific model checking using the bogor framework. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 369–370, Washington, DC, USA, 2006. IEEE Computer Society.
21. SpecExplorer. http://research.microsoft.com/SpecExplorer, 2006.
22. J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
23. M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.
24. M. Veanes, C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, and N. Tillmann. Model-based testing of object-oriented reactive systems with Spec Explorer, 2005. Tech. Rep. MSR-TR-2005-59, Microsoft Research. Preliminary version of a book chapter in the forthcoming text book *Formal Methods and Testing*.
25. M. Veanes, C. Campbell, and W. Schulte. Parallel and serial composition of model programs. Technical Report MSR-TR-2007-22, Microsoft Research, February 2007.
26. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 273–282, New York, NY, USA, 2005. ACM Press.
27. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer, April 2005.