Declarative Symbolic Pure-Logic Model Checking

by

Ilya A Shlyakhter

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

Author . . . .
Department of Electrical Engineering and Computer Science
January 12, 2005

Certified by . . . .
Daniel N Jackson
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Declarative Symbolic Pure-Logic Model Checking

by

Ilya A Shlyakhter

Submitted to the Department of Electrical Engineering and Computer Science
on January 12, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Model checking, a technique for findings errors in systems, involves building a formal model that describes possible system behaviors and correctness conditions, and using a tool to search for model behaviors violating correctness properties. Existing model checkers are well-suited for analyzing control-intensive algorithms (e.g. network protocols with simple node state). Many important analyses, however, fall outside the capabilities of existing model checkers. Examples include checking algorithms with complex state, distributed algorithms over all network topologies, and highly declarative models.

This thesis addresses the problem of building an efficient model checker that overcomes these limitations. The work builds on Alloy, a relational modeling language. Previous work has defined the language and shown that it can be analyzed by translation to SAT. The primary contributions of this thesis include: a modeling paradigm for describing complex structures in Alloy; significant improvements in scalability of the analyzer; and improvements in usability of the analyzer via addition of a debugger for overconstraints. Together, these changes make model-checking practical for important new classes of analyses. While the work was done in the context of Alloy, some techniques generalize to other verification tools.

Thesis Supervisor: Daniel N Jackson
Title: Associate Professor

Acknowledgments

Many people helped make this thesis happen. First of all, I thank my advisor Daniel Jackson for getting me interested in formal verification, for giving patient guidance and useful advice, and for showing me that I can be wrong even when I'm sure I'm right. I thank my thesis readers, Srini Devadas and David Karger, for their good comments. I thank my colleagues at CSAIL – Manu Sridharan, Sarfraz Khurshid, Rob Seater, Mana Taghdiri, Mandana Vaziri and others – for the many good discussions. I thank my family for their constant support. And I thank Bet' – for teaching me many things, and for being.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis contributes a set of techniques for finding errors in systems and algorithms. The techniques apply in the context of model checking: the user builds a formal model of the algorithm and its correctness conditions; the space of possible algorithm executions is then automatically searched for executions violating the correctness conditions. The techniques enable declarative modeling and analysis of algorithms that manipulate complex, graph-like data structures. The techniques enable the analysis to scale to realistic examples. The techniques also enable flexible modeling in which new types of analyses can be realized by adopting new modeling patterns, rather than by changing the modeling language or the analysis tool. A prototype implementation of the techniques in the Alloy Analyzer, a model checker for the Alloy modeling language, is described. The result of the implementation is a model checker with a combination of features unavailable in other tools.

1.1 Model checking

Model checking [12], a framework for finding errors in algorithms, has gained popularity in recent years. In model checking, the user creates a formal model of an algorithm and of its correctness properties. An automatic tool (the model checker) then answers the question "is there an execution of the algorithm violating a given correctness property?" Because this question is in general undecidable, the model checker analyzes only instantiations of the algorithms below some bound (e.g. all executions of an n-process distributed algorithm with up to 10 processes). Within the bound, all or most executions of the algorithm can be checked.

Model checking has advantages relative to other methods of assuring correctness, such as testing and theorem proving. Unlike testing, model checking does not require manually constructing many test cases; the user only needs to construct one model. Model checking can also provide much better coverage of algorithm executions than testing. Relative to theorem proving, model checking provides less of a guarantee of correctness since it only checks bounded instantiations of an algorithm. The advantage of model checking over theorem proving is that model checking requires much less human effort and mathematical sophistication. Also, when an algorithm does not satisfy a correctness property, model checking provides a trace of the algorithm illustrating the violation. Model checking can be used in conjunction with theorem proving, to help formalize the statements to be proved and to ensure that they are indeed correct at least on examples of bounded size.

1.2 Current model checkers

In existing model checkers [33, 16, 11], the algorithm to be checked is specified as a finite state machine (FSM). The user describes the FSM by providing the following elements: the structure of the FSM's state vector, a specification of the initial state, and a specification of the transition relation. The FSM state vector is defined as a collection of variables of primitive type (e.g. integer or enumeration). The initial state is given as an assignment to the state vector variables. The transition relation is given by specifying formulas for computing the next-state value of each variable from the present-state variable values. Several FSMs specified in this way can be composed in parallel. Communication between FSMs is modeled using shared variables [11] or message queues [33].

An example of an SMV model is shown in Figure 1-1. The model represents Peterson's mutual exclusion algorithm for two processes [66]. It's not necessary to understand the details of the algorithm; we use it only to illustrate the main elements of SMV modeling.

The state of each of two processes consists of a single enumerated-type variable `state`. In addition, there is one global binary variable `turn`. The state of the entire two-process system thus consists of two enumerated `state` variables and one binary `turn` variable. In the definition of `MODULE proc`, `turn` denotes the global variable `turn`; `myturn` denotes the constant identifier of the process (0 or 1); and `other.state` denotes the `state` variable of the other process.

The initial value of `state` is set to be `noncritical` by the line

```
init(state) := noncritical
```

The following line specifies the transition relation for the `state` variable, giving the formulas to compute the next-state value of this variable from the present-state values of the state variables. For instance, if the present-state value of `state` is `noncritical`, then the next-state value of this variable may be either `noncritical` or `request`. The module `proc` specifies the behavior of a single process as an FSM; the module `main` instantiates two such processes. The transition relation of the two-process system is obtained as a parallel composition of the transition relations of the two individual processes. A correctness property, specified by the line

```
AG !(p1.state = critical && p2.state = critical)
```

states that the two processes are never in their critical sections simultaneously. The SMV tool can automatically search for system traces violating this property.

1.3 Alloy Alpha

Traditional model checkers are well-suited for analyzing hardware protocols with simple node state. However, they're hard to apply to model checking problems arising from analysis of software. In these problems, complexity can arise not from the large number of interleavings of parallel processes but from the complex structure of the state space of a single process. Moreover, the correctness conditions can be complex topological conditions such as "a heap manipulation preserves acyclicity of the heap", rather than simple state predicates such as "a process reaches an error state" or "two processes reach critical section simultaneously". Furthermore, in these problems, the ability to specify operations

```
-- SMV model of Peterson's mutual exclusion algorithm
MODULE proc(turn, myturn, other)

VAR
  state : {noncritical, request, enter, critical};

ASSIGN
  init(state) := noncritical;
  next(state) :=
    case
      state = noncritical : {noncritical, request};
      state = request : enter;
      state = enter & (other.state = noncritical | turn = myturn) : critical;
      state = critical : {critical, noncritical};
      1 : state;
    esac;

  next(turn) :=
    case
      state = request : !myturn;
      1 : turn;
    esac;

MODULE main

VAR
  turn : boolean;
  p1   : process proc(turn, 0, p2);
  p2   : process proc(turn, 1, p1);

SPEC
  -- The two processes are never both critical
  AG !(p1.state = critical && p2.state = critical)
```

Figure 1-1: Sample SMV model

declaratively rather than imperatively – by describing the conditions that are true when an operation occurs, rather than giving an executable recipe for the operation – can be important.

To meet these needs, the Alloy language and analyzer were developed [37, 38, 39, 36]. The language is based on first-order relational logic with transitive closure. The state of the system under analysis is represented as a collection of relations. All analysis questions are reduced to satisfiability of a first-order logical predicate over these relations. Several types of analysis are possible: simulation (showing examples of system state, or of operation execution); checking of invariant preservation (checking whether an execution of an operation always preserves a given invariant); refinement checking (checking that a concrete implementation correctly simulates a given abstract operation under a specified abstraction function). The use of relational logic with transitive closure allows expression of constraints on graph-like data structures that occur in software systems. The use of logic also allows declarative specification of operations. Analysis is done by reduction to Boolean satisfiability.

Figure 1-1 shows an Alloy Alpha model of a simple file system. An Alloy model consists of three primary parts: basic types, relations, and constraints.

Each basic type defines a set of uninterpreted atoms, used to represent system components or primitive values. Basic types are defined in the `domain` paragraph of the model. The Finder model has only one basic type (`Obj`); its atoms represent files and directories. The number of atoms in each basic type (called the *scope*) is not built into the model, but is specified during analysis. The scope determines the space of system instances that can be represented by the model; for `|Obj|=5`, file system instances containing a total of up to five files and directories can be represented.

Relations – sets of tuples of basic type atoms – represent the state of the system. Relations are defined in the `state` paragraph. The Finder model has six unary relations (`drive`, `trash`, `File`, `Folder`, `Alias` and `Trashed`) and two binary relations (`dir` and `alias`). `File` and `Folder` partition all file system objects (`Obj` atoms) into those representing files and those representing folders. Some files are aliases pointing to other file system objects; `Obj` atoms representing aliases are members of `Alias`. `dir` relates each

```
model Finder {
  domain {Obj}                             // Obj models the set of all file system objects
  state {
    disjoint drive, trash : fixed Obj !    // drive and trash are distinct individual objects
    partition File, Folder: static Obj     // objects are partitioned into files and folders
    dir: Folder ? -> Obj                   // d.dir is the set of objects inside directory d
    Alias : File                           // aliases are treated as a subset of files
    alias : Alias ->! Obj                  // a.alias is the object the alias a points to
    Trashed : Obj                          // set of objects in the trash
    }

  inv Standard {                          // some basic invariants
    Trashed = trash.*dir                  // Trashed is the set of objects contained in the trash
    drive not in Trashed                  // can't trash the drive
    drive + trash in Folder               // drive and trash are folders
    no (drive + trash).~dir               // drive and trash are both top-level
    no o: Obj | o in o.^alias             // no cyclic aliasing
    no o: Obj | o in o.^dir               // no cycles in directory structure
    }

  cond TwoLevelFS {some Folder.dir}       // make a file system with at least two levels

  op Move (x, to : Obj!) {                // Move x to the new folder to
     to not in x.*dir                     // to cannot be a descendant of x
     all o | o.alias = o.alias'           // aliases are unchanged
     all o | o != x -> o.~dir = o.~dir'   // objects distinct from x stay in same place
     x.~dir' = to.*alias - Alias          // x's new parent found by following aliases
     Obj' = Obj                           // no objects created or destroyed
     }

  assert TrashingWorks { all x, to | Move (x, to) and to in Trashed -> x in Trashed' }
}
```

Figure 1-2: Sample Alloy Alpha model: the Macintosh Finder

directory to its contents, and `alias` relates each alias to its target. Singleton sets (unary relations) `drive` and `trash` represent two special folders. `Trashed` represents the contents of `trash`, including anything reachable through subfolders. Eached relation has a primed version; this lets model instances represent operation executions, by representing system state before and after the operation.

Constraints are used to describe well-formedness requirements, specify operations, and specify correctness conditions to be checked. Constraints are defined in `def`, `inv`, `cond`, `op` and `assert` paragraphs. The analyzer searches the space of models within the specified basic type scopes for instances satisfying the given constraints. There are two analysis modes: simulation and checking. In simulation mode, the tool finds sample instances of system state satisfying the basic well-formedness conditions (`inv`) and any additional conditions (`cond`); this is used to check consistency of constraints. An operation can also be simulated; in that case, the analyzer finds a pair of states satisfying the basic well-formedness conditions (`inv`) and the operation constraints (`op`) relating pre-state to post-state. In checking mode, the analyzer searches for a state or state pair satisfying the well-formedness conditions (`inv`) and operation constraints (`op`), but violating an assertion (`assert`).

The constraint language is essentially first-order logic with transitive closure. The expression `*dir` denotes reflexible transitive closure of `dir`, `trash.*dir` denotes the relational image of the singleton set `trash` under that closure – i.e. the set of objects reachable from `trash` by following arcs of `dir`. `drive + trash` denotes relational union (as sets of tuples) of `drive` and `trash`; `drive + trash in Folder` makes that union a subset of the unary relation (set) `Folder`. `~dir` denotes the transpose of the binary relation `dir`; `(drive + trash).~dir` thus denotes the set of parents of `drive` and `trash` under `dir`; `no (drive + trash).~dir` makes that set empty. `+alias` denotes transitive closure of `alias`, `o.+alias` denotes the set of objects reachable from `o` in at least one step via arcs of `alias`; `no o | o in o.+alias` makes that set empty, stating that `alias` is acyclic.

This model illustrates two key characteristics of Alloy. The first is the ability to represent complex system state, and to express predicates on complex state. The state of a file system includes graphs representing the directory and aliasing structure, and complex predicates involving transitive closures can be expressed. The second is the ability to specify operations declaratively. The `Move` operation is specified by listing constraints between the pre-state and the post-state, rather than by giving explicit formulas for computing post-state values of relations from the pre-state.

1.4 Limitations of Alloy Alpha

Alloy Alpha had a number of limitations that limited the range of problems to which it could be applied. In this section, we give a concise summary of these limitations. In the next section, we will describe techniques contributed by this thesis for overcoming these limitations.

1. Alloy Alpha's support for handling complex data structures was limited. It could

model system state containing several graphs and relations; but there was no systematic way to model algorithms manipulating multiple, distinct instances of complex data structures. You couldn't, for instance, model a distributed algorithm in which the state of each node included complex data structures and in which nodes exchanged messages containing complex data structures. There was no systematic support for modeling sets of complex structures, or tables with complex structures as keys and values.

2. In Alloy Alpha, analyses were limited to those that could be expressed in the form "find a state or a pair of states satisfying a given condition". A number of commonly needed analyses do not fit this framework. For example, Bounded Model Checking (BMC) analyses ask whether there exists a sequence of k states, starting with an initial state and obeying the transition relation between adjacent states, that illustrates an error [9]. Adding BMC analyses, or other analyses for which the analyzer was not originally designed, would require changing the Alloy language and analyzer. Even if support for specific analyses such as BMC was added, supporting variations of these analyses would require further changes to the language and the analyzer.

It was possible to "abuse" the original Alloy language to conduct analyses not initially intended by language designers. For instance, all relations were declared inside a `state` paragraph, implying that they represent a single state of the system; and explicit language support was provided for creating a second copy of state using these relations. All Alloy Alpha documentation and examples assumed that the relations inside the `state` paragraph represent system state. However, it was syntactically possible to "cheat" and use these relations to represent other things – for example, entire traces (i.e. sequences of states). While this was syntactically possible, as a practical matter this ability was not described anywhere and thus wasn't useful to Alloy Alpha users.

3. Declarative modeling in Alloy Alpha, while possible, was limited in practice by the lack of a debugger for overconstraints. Overconstraint is a modeling error that can prevent the analyzer from finding an error in the modeled algorithm. It occurs when some behaviors possible in the algorithm are inadvertently ruled out by the model. Declarative modeling carries a special risk of overconstraint, because an operation is typically described as a conjunction in which each conjunct rules out some impossible executions. Thus the set of

possible executions is specified implicitly, and whether all algorithm behaviors are included isn't obvious from the specification. In an extreme case, a single conjunct may erroneously rule out all possible executions; then, the analyzer won't find any error traces because the model has no traces at all. The possibility of overconstraint reduces confidence in the analyzer's reports of correctness: are there no counterexamples because the algorithm is correct, or because the model is overconstrained?

Even if the user suspects overconstraint (e.g. analysis finishes suspiciously quickly on what should be a large search space), identifying the culprit conjunct is difficult. The lack of support for debugging overconstraints has been the biggest complaint of Alloy Alpha users. Being able to debug overconstraints is critical to enabling declarative modeling. For example, the NuSMV model checker allows declarative specification of transition relations; but the NuSMV manual explicitly discourages the practice because of the risk of overconstraint, and almost none of the sample models that come with NuSMV use declarative specification [11].

4. Alloy Alpha analyzer didn't scale well. Its translation of Alloy formulas to Boolean formulas was inefficient, producing large formulas that were hard for SAT solvers.

5. Alloy Alpha didn't take advantage of inherent symmetries in Alloy models.

1.5 Contributions of this thesis

In this section, we describe a set of techniques for overcoming the limitations described in the previous section.

1.5.1 Objectification of complex data structures

While Alloy Alpha could model systems where the overall system state is described by graphs and relations, it had no systematic way of modeling multiple, distinct instances of complex data structures. The need for such modeling occurs frequently in practice. For instance, in a distributed algorithm for updating network routing tables, the state of each node and the contents of each message might contain a routing table. A counterexample showing a run of this algorithm on 3 nodes for 2 steps might contain 6 or more distinct routing tables.

We describe a simple technique, *objectification*, for modeling multiple complex structures in Alloy. The technique involves using basic types in a new role. In Alloy Alpha, basic type atoms are used to represent two things: system elements (files, network nodes) and primitive values (time points, enumerated type members). We add a third use of basic type atoms: to represent instances of complex structures used in the solution. [1]

As Alloy Alpha has shown, a single complex structure can be represented by a group of relations. Suppose we want to represent a network routing table. We could use the following definitions:

```
domain { Node }
state {
  nextHop: Node -> Node,  // for each msg destination, the next hop
  definedFor: set Node    // nodes for which we have routing info
}
```

`nextHop` represents routing information: it contains the tuple $\langle n_i, n_j \rangle$ iff packets destined for node n_i should be routed through node n_j. `definedFor` represents the set of nodes for which we have routing information; it contains the tuple $\langle n_i \rangle$ iff we have routing information for node n_i.

Alloy Alpha also showed that informal predicates on the complex structure can be formalized naturally and concisely using first-order logic with transitive closure. It's possible to express properties such as the following:

```
// definedFor contains exactly the nodes in the domain of nextHop
all n: Node | n in definedFor iff some n.nextHop
// there are no routing cycles
all n: Node | n !in n.^nextHop
```

To represent multiple instances of a complex structure, we define a new basic type. Each atom of this basic type represents a distinct instance of the complex structure. For each relation representing a component of the structure, we define a new relation by adding the new basic type as first column. The new relation maps each atom of the new basic type to the component's value in the corresponding complex structure instance. In the routing table example, the new definitions would look as follows:

[1]These "roles" given to basic types are purely in the mind of the modeler; neither the language nor the analyzer distinguish basic type roles.

```
domain { Node, RTab }
state {
  rtNextHop: RTab -> Node -> Node,
  rtDefinedFor: Rtab -> Node
}
```

Each atom of `RTab` represents a distinct instance of a routing table. If `t` is a singleton set containing an `RTab` atom, then `t.rtNextHop` (a relation of type `Node -> Node`) denotes the routing information of the corresponding routing table. `t.rtDefinedFor` (a unary relation of type `Node`) denotes the set of nodes for which the corresponding routing table has routing information.

Constraints on an Alloy instance representing a single complex structure can be systematically rewritten into constraints on the structure instance represented by a given atom. In our example, assuming `t` is a singleton set containing an `RTab` atom, the constraints can be rewritten as follows:

```
// t.rtDefinedFor contains exactly the nodes in the domain of t.rtNextHop
fun ConsistentRTab(t: RTab) { all n: Node | n in (t.rtDefinedFor) iff some n.(t.rtNextHop) }
// there are no routing cycles
fun AcyclicRouting(t: RTab) { all n: Node | n !in n.^(t.rtNextHop) }
```

Thus, a singleton set containing an atom representing a complex structure instance is a full-featured representative of that instance: we can write constraints on the value of the complex structure in terms of the singleton set. [2] The relational image operator lets us denote the value of the components of the complex structure associated with the given atom. At the same time, the atom remains a primitive entity without internal structure. The atom can be part of tuples contained in relational values of other complex structures instances. This lets the value of one complex structure instance contain references to other complex structure instances. The full generality of a heap can therefore be represented; in particular, recursive data structures such a lists and trees – with complex structures as nodes – can be represented. By contrast, the compound structures representable in SPIN [32] and SMV [11] can't contain references to instances of other compound structures. Analysis is therefore limited to structures that can be represented on a C stack.

The total number of possible values of a complex structure can be very high: each relation-valued component of size $m \times n$ contributes a factor of $2^{m \times n}$. In any given Alloy

[2]The function notation used here is simply a way to define parameterized macros in Alloy; after writing the above definitions, *ConsistentRTab(expr)* denotes the body of *ConsistentRTab* with *t* replaced by *expr*.

instance, only a small number of these complex structure values will be represented by atoms. However, *which* values are represented is a choice made by the analyzer. The chosen complex structure values act as a palette from which the instance is constructed. Analysis can be thought of as searching through all possible palettes of the specified size, and for each palette, searching through all instances that can be constructed from that palette. In the routing table example, for `|RTab|=k`, the analyzer considers all palettes of `k` routing table values, and for each palette considers all Alloy instances that can be constructed using the given `k` routing table values.

A key feature of the objectification scheme is that existential quantifiers over complex structures can be expressed using first-order constraints. [3] The constraint "there exists a complex structure value satisfying constraint X" can be expressed as "there exists an atom such that its associated complex structure satisfies X". Any instance satisfying the latter also satisfies the former. This, and the ability to reference complex structure instances from complex structure values, make it possible to ask "is there a group of mutually referencing complex structures, satisfying the given constraints?" – where the constraints on a structure can include constraints on the structures it references.

Signatures

To formalize the view of basic type atoms as representatives of complex structure instances, the notion of *signatures* was added to the Alloy language [36, 44]. [4] A signature is a basic type whose atoms represent complex structure instances, together with a group of relations with that basic type as first column. The relations define, for each basic type atom, the value of the complex structure represented by that atom. In the routing table example, the definitions rewritten using signatures would look as follows:

```
sig Node { }
sig RTab {
   rtNextHop: Node -> Node,
   rtDefinedFor: set Node
}
```

[3] This doesn't hold for universal quantifiers over complex structures, but such quantifiers are rarely needed in modeling.

[4] Signatures as a language feature are not a contribution of this thesis, but are explained here so we can use the signature notation in subsequent examples.

The signature `Node` has no fields; this reflects the fact that the atoms of `Node` represent primitive entities (node identifiers), rather than complex structures. The signature `RTab` has two fields. The field `rtNextHop` is defined to have relation type `Node -> Node`; because it is a field of `RTab`, `RTab` is added as the first column of the relation. Thus, the declaration of the field `rtNextHop` here defines the same relation, of type `RTab -> Node -> Node`, as the earlier declaration of relation `rtNextHop`. The signature-based declaration used in Alloy can always be desugared to flat relation declarations used in Alloy Alpha. However, signatures play an important role in structuring the model and organizing the modeler's thinking. They make explicit the objectification of complex structures, and bear a useful resemblance to `struct` declarations in languages such as C. They also support an inheritance mechanism, which is not covered here but is explained in Section 2.3.3.

1.5.2 Pure-logic modeling

One important consequence of being able to objectify complex structures is that the entire state of a system (or a system component, e.g. a process) can be objectified. This means that an Alloy instance can represent a collection of states – for instance, a k-step trace of an algorithm – rather than just one or two states as in Alloy Alpha. Moreover, standard Alloy constraints can be used to express relationships between state instances – for example, to require that adjacent states in a sequence of states satisfy a transition relation, or that the sequence start with a valid initial state. As a result, explicit support for modeling finite state machines can be removed from the Alloy language and analyzer. The model then simply describes a group of relations and constraints on the values of these relations; the relations are not necessarily interpreted as being elements of system state. No primed (next-state) versions of relations are created. While the modeler may mentally designate some relations to represent system state and some constraints to represent transition relations, such designations are not formally specified in the Alloy language and are not used by the Alloy analyzer. We call this approach *pure-logic modeling*.

Encoding standard analyses with pure-logic modeling

While pure-logic modeling requires the user to write some constraints previously generated automatically, in practice this isn't a burden and standard patterns can usually be followed. On the other hand, pure-logic modeling gives the user great flexibility in defining new analyses and variations of existing analyses. New analyses can be defined by adopting new modeling patterns, rather than by changing the Alloy language or its analyzer.

Let'ss consider some modeling patterns that can be used in pure-logic modeling.

Simulating a system state or an operation; checking invariant preservation. Suppose we have a system the state of which is described by a group of relations. We can objectify the state by declaring a signature `SystemState`, and making these relations fields of `SystemState`:

```
sig SystemState {
   // relations describing system state: f, g
}
```

The Alloy instance can now represent a collection of states, the exact number being controlled by the scope of `SystemState`. We can now express a variety of analyses. First, we can express the analyses possible in Alloy Alpha – finding a state or state pair satisfying the given constraints:

```
fun ValidState(s: SystemState) { ... s.f ... s.g ... }
// find instance representing one state,  satisfying ``some s: SystemState | ValidState(s)''
// set scopes of all basic types except SystemState to 3; set scope of SystemState to 1.
run ValidState for 3 but 1 SystemState

fun Op(s, s': SystemState) { ... s.f ... s'.f ... s.g ... s'.g ... }
// find instance representing a (pre,post)-state pair, satisfying ``some s, s': Op(s, s')''
run Op for 3  but 2 SystemState

fun FindInvarViolation(s, s': SystemState) { ValidState(s) && Op(s,s') && !ValidState(s') }
// find instance representing a (pre,post)-state pair, showing invariant violation
run FindInvarViolation for 3 but 2 SystemState
```

In the definitions of functions `ValidState`, `Op` and `FindInvarViolation`, `s.f` and `s.g` denote the present-state value of state components `f` and `g`, while `s'.f` and `s'.g` denote the next-state value of the respective state components. However, the prime in `s'` is simply part of the variable name; it has no special meaning in the language or for the analyzer. As far as the language and analyzer are concerned, we're simply specifying and solving a constraint problem. Note also that, unlike in Alloy Alpha, the relation declarations aren't wrapped inside a `state { ... }` schema. In pure-logic modeling, the

Alloy instance is not restricted to representing a state; it can represent a state, a state pair, or any other complex relational object intended by the modeler.

Bounded Model Checking in Alloy. One useful idiom that can be expressed with pure-logic modeling is *bounded model checking* (BMC) [9]. In BMC, the Alloy instance represents a *bounded trace* of an algorithm: a sequence of states where the first state is a valid initial state and each adjacent pair of states satisfies the transition relation. Additional constraints on the trace require the trace to illustrate an error – for example, requiring one of the reached states to exhibit an error condition. A search for such an instance answers the question "does there exist a k-step trace of the algorithm illustrating an error", for some bound k. The general pattern for expressing BMC problems in Alloy is as follows:[5]

```
open std/ord  // use total ordering module
fun InitialState(s: SystemState)  { ... s.f ... s.g ... }
fun Transition(s, s': SystemState) { ... s.f ... s.g ... s'.f ... s'.g ... }
fun ValidTrace() {
   ValidInitialState(OrdFirst(SystemState))
   all s: SystemState - OrdFirst(SystemState) | Transition(OrdPrev(s), s)
}
fun GoodState(s: SystemState)  { ... s.f ... s.g ... }
fun FindError() { ValidTrace() &&  some s: SystemState | !GoodState(s) }
run FindError for 3 but 5 SystemState
```

The function `InitialState` constrains the state `s` (i.e. the state value associated with the `SystemState` atom in the singleton set `s`) to be a valid initial state of the algorithm. The function `Transition` constrains `s,s'` to represent a valid (pre-state,post-state) pair according to the transition relation. The function `ValidTrace` constrains the Alloy instance to represent a valid bounded trace, beginning in an initial state and obeying the transition relation. Finally, the function `FindError` requires the trace to reach an error state. An instance satisfying these constraints illustrates an error in the modeled algorithm. The absence of such an instance means that an error may still be present, but finding it requires larger basic type scopes.

Checking a system under all possible environments. Now consider a variation of BMC analysis: suppose that each run of the modeled algorithm occurs within some en-

[5]The example uses the total ordering module `std/ord`, which defines some predefined relations expressing a total order on a given basic type. For the purposes of this example, it's enough to know that `OrdFirst(SystemState)` denotes the singleton set containing the first atom of `SystemState` in the total order on `SystemState` atoms, and `OrdPrev(s)` denotes the singleton set containing predecessor of the atom denoted by the singleton set `s` (or the empty set if `s` is the first atom in the order).

vironment. For example, a run of a network algorithm occurs on a particular network topology; a run of a railway system occurs on a particular rail net. We'd like to check, for all possible environments, all possible runs of the algorithm on each environment. We can make the environment a variable of the model; an instance would then represent the value of the environment, together with a bounded trace of the algorithm on that environment. The modified definitions would look as follows:

```
static sig Env {  // static means |Env|=1
   // relations describing the environment: e1, e2, ...
}
fun InitialState(s: SystemState)  { ... s.f ... s.g ... Env.e1 ... Env.e2 ... }
fun Transition(s, s': SystemState)
{ ... s.f ... s.g ... s'.f ... s'.g ... Env.e1 ... Env.e2 ...  }
// ...
run FindError for 3 but 5 SystemState, 1 Env
```

An instance represents several `SystemState` values (a bounded trace), but only one `Env` value. The environment is not part of the changing state, but it's still a variable of the model: when the analyzer searches for an Alloy instance illustrating an error, it searches through all possible environment values and through all possible traces on each environment (within the specified scopes). In the definitions of `InitialState` and `Transition`, `Env.e1` and `Env.e2` denote the value of the relations representing the environment.

Pure-logic modeling lets us encode this variation of BMC, in which there is a state that changes with time and an environment that doesn't. We can check, in one run, all runs on all environments within the specified scopes. Doing this with a traditional BMC model checker would require one of the following: modifying the checker to support this variation of BMC analysis; manually hard-coding each possible environment value into the model and running a separate analysis for each value; or making the environment part of the state, unnecessarily increasing analysis complexity. With pure-logic modeling, we can express exactly what we want without changing the language or the analyzer.

Language design considerations. While pure-logic modeling lets us encode common modeling patterns such as BMC relatively easily, one may still ask: wouldn't it be better to have special language support for at least some of the common modeling patterns? After all, objectification is also a modeling pattern, and having special language support for it (signatures) has proven very useful. One answer is that it's possible to use Alloy as an intermediate language for more specialized languages or "veneers", which are analyzed by translation to Alloy but make certain tasks more convenient. Such veneers have been created for modeling virtual functions [59], annotating Java code [59, 80], and specifying the structure of test cases [50]. While Alloy can act as an intermediate language to which veneers are translated, it remains very usable as a modeling language in which users write models directly. Objectification is a basic building block on top of which a variety of modeling patterns (such as bounded model checking) can be implemented. It is therefore sufficient, in the base Alloy language, to provide direct support for objectification [36, 44] but not for higher-level modeling patterns. Whenever the need to write some common pattern by hand becomes a problem, an appropriate veneer can be created. Also, it's possible to define standard Alloy libraries for common tasks, that can then be reused for a variety of models. Such generic libraries have been created, for example, for modeling groups of processes communicating via messages.

1.5.3 Debugging of overconstraints

One of the difficulties inherent in declarative modeling is the risk of unintended overconstraint. An overconstrained model does not allow all algorithm executions that can occur in practice, and may therefore be unable to illustrate some errors. In an extreme case, a model may have no error traces because it has no traces at all. Such a scenario is unlikely when the transition relation is given imperatively: there is an explicit formula for computing the post-state from the pre-state, and by iterating the formula from initial states at least some traces can be generated. A declaratively specified transition relation, on the other hand, typically consists of a conjunction of conditions specifying what must be true of a valid (pre,post)-state pair: $T(S, S') = \bigwedge_{i=0}^{k} T_i(S, S')$. If any one of the T_i disallows all transitions, so does the entire transition relation. The danger of overconstraint undermines confidence in analysis results: if the analyzer finds no counterexamples, is this because the algorithm is correct or because the model is overconstrained? Even if the user suspects overconstraint (e.g. because of unusually fast analysis times), localizing the problem to a particular T_i is non-trivial.

One way to guard against overconstraint is to use simulation: to use the analyzer to find sample executions, to make sure they're allowed by the model. Symmetry-breaking, described in Chapter 4, can aid this process by removing isomorphic executions; this lets the user quickly verify that all the "essentially different" (non-isomorphic) scenarios are allowed by the model. This process, however, is not a satisfactory solution to the overconstraint problem. It requires the user to come up manually with specific execution scenarios, as when constructing a test suite; this defeates the purpose of doing model checking, which was to avoid such tedious and error-prone manual work. The number of scenarios to check for can be very large, and it's hard to know when enough scenarios have been checked.

This thesis contributes an *overconstraint debugger* for Alloy Analyzer. When the analyzer fails to find a counterexample, the debugger identifies parts of the model that are irrelevant to showing absence of counterexamples; if these parts are changed, there would still be no counterexamples to the checked property. Irrelevance of large parts of the model can indicate an overconstraint. More generally, the modeler writes certain parts of the model to guarantee certain properties. If the modeler's intuition about which parts are required to guarantee which properties turns out to be wrong, this can point to an error in the model. Conversely, if the correspondence of model parts to guaranteed properties matches the modeler's intuition, this increases confidence in correctness reports from the analyzer.

For example, consider a model of a system in which there are processes and locks, and at each step a process either grabs or releases a lock. If the locks are numbered and the processes only allowed to grab locks numbered higher than the locks already held by the process, deadlock will be avoided. This can be checked by an assertion such as:

```
assert DijkstraPreventsDeadlocks {
   some Process && GrabOrReleaseAtEachStep() && LocksGrabbedInOrder() => ! Deadlock()
}
```

When an Alloy model of this system was analyzed, no counterexamples were found. However, the overconstraint debugger showed something surprising: that the condition `LocksGrabbedInOrder()` was irrelevant to proving the absence of counterexamples. That is, with that condition omitted, there would still be no deadlock. This contradicted the modeler's intuition, and in fact indicated an error in the model.

As noted earlier, all analysis questions in Alloy are reduced to the satisfiability of a single Alloy formula. That formula is typically a large conjunction, e.g.

```
F && G && H && P && Q
```

If the formula is unsatisfiable, the debugger can identify an *unsatisfiable core* – a subset of the conjuncts that by themselves rule out all solutions. For instance, it could determine that `G && P` is unsatisfiable, making the contents of the remaining conjuncts irrelevant.

The debugger works by taking advantage of the ability of SAT solvers to determine the unsatisfiable core of a CNF formula. A SAT solver takes a Boolean formula in conjunctive normal form (conjunction of clauses which are disjunctions of literals). If the formula is unsatisfiable, it can identify an unsatisfiable core of the CNF: a subset of CNF clauses that by itself is unsatisfiable. SAT solvers' ability to extract smaller unsatisfiable cores from CNF formulas is being constantly improved [86, 56, 85]; as that ability improves, the precision of Alloy's overconstraint debugger improves correspondingly.

We test satisfiability of an Alloy formula by translating it to an equisatisfiable CNF formula. However, the unsatisfiable core results must be given in terms of the original Alloy model in order to be meaningful to the user. To map the unsatisfiable core of that CNF formula back to the original Alloy formula, we keep track of which CNF clauses were generated from which parts of the Alloy formula. More precisely, the original Alloy formula is represented as an Abstract Syntax Tree (AST). During translation to CNF, some CNF clauses are generated from each AST node. To map CNF unsatisfiable core to Alloy unsatisfiable core, we include an AST node in the Alloy unsatisfiable core if at least one CNF clause generated from that AST node was part of the CNF unsatisfiable core.

A naïve mapping of CNF clauses to the original Alloy formula might produce a malformed formula that is not a valid Alloy formula. For instance, suppose the original Alloy formula included a negation – a subformula of the form `!F`. If the CNF clauses generated from AST nodes of the subformula F were in the unsatisfiable CNF core, but CNF clauses generated from the AST node expressing the negation operator weren't, then the unsatisfiable core of the Alloy formula includes a negation node with no child. That's not a valid Alloy formula, and we cannot call it "unsatisfiable" because it cannot be evaluated (to true or false) on various Alloy instances.

Even if the Alloy formula obtained by naïvely mapping back the CNF unsatisfiable core is well-formed, it's hard to prove that it is unsatisfiable. The CNF translation of an Alloy formula is compositional and depends on the entire structure of the Alloy formula. Altering the formula's structure by removing formula parts which didn't yield CNF clauses in the unsatisfiable core breaks the relationship between the original Alloy formula and the CNF translation containing the unsatisfiable CNF core – making it hard to prove that the resulting Alloy formula is unsatisfiable.

To solve these problems, we use the following scheme to map CNF unsatisfiable cores to Alloy unsatisfiable cores. An Alloy formula can be viewed as a tree, with nodes representing subformulas. Each node computes a particular function of its children. We define a notion of *relaxing* a node: allowing it compute an arbitrary function of its children. So, if a node of the form `F && G` is relaxed, it may compute either Boolean value regardless of the values of `F` and `G`. If a node of the form `P.Q` is relaxed, it may compute any relational value of the correct relation type, regardless of the values of `P` and `Q`. Thus, an Alloy formula with some nodes relaxed (a *relaxed formula*) is always well-formed and meaningful to the user. A relaxed Alloy formula represents a class of normal Alloy formulas, which agree with the relaxed formula on non-relaxed nodes. The notion of unsatisfiability extends naturally to relaxed formulas: a relaxed formula is unsatisfiable iff none of the concrete formulas it represents is satisfiable.

We express unsatisfiable cores of Alloy formulas by marking a subset of Alloy formula nodes as relaxed. To map a CNF unsatisfiable core to the Alloy formula, we relax a formula node if none of the clauses generated from that node are in the CNF core. A correct CNF translation of a relaxed Alloy formula can be obtained by taking the CNF translation of the normal Alloy formula and removing clauses generated from the relaxed nodes. Our mapping scheme ensures that the CNF translation of the relaxed Alloy formula includes all clauses of the unsatisfiable CNF core; since the CNF translation of the relaxed formula is unsatisfiable, the relaxed formula itself is unsatisfiable.

1.5.4 Scalability features

The modeler reduces the question "does the system have an error?" to the question "does the given Alloy formula have a solution?" The Alloy Analyzer reduces the question "does the given Alloy formula have a solution?" to the question "does the given Boolean formula in conjunctive normal form (CNF) have a satisfying assignment?", which is answered by external satisfiability solvers. How the Alloy formula is encoded in CNF can significantly affect solver performance. Two methods are used by Alloy Analyzer for improving solver performance: symmetry-breaking and subformula sharing.

Symmetry-breaking

Many systems exhibit symmetry, for example in the form of interchangeable system components or indistinguishable primitive values. Symmetry partitions the space of executions into equivalence classes. For any given property, the executions in a single equivalence class either all satisfy or all violate the property. It is therefore sufficient to consider only one execution per isomorphism class. This can lead to an exponential reduction in the size of the search space.

To take advantage of model symmetries, Alloy Analyzer conjoins *symmetry-breaking predicates* [15] to the Boolean formula given to the SAT solver. A symmetry-breaking predicate is constructed to be true of at least one solution in each isomorphism class; thus, adding a symmetry-breaking predicate preserves satisfiability of the formula. An effective symmetry-breaking predicate is true of the smallest possible number of solutions in each isomorphism class. The predicate speeds up backtracking search by causing a backtrack whenever all extensions of the backtracking algorithm's current partial variable assignment violate the predicate.

For unsatisfiable formulas – which typically take longest to analyze, since the entire search space must be considered – the addition of symmetry-breaking predicates clearly provides a benefit. For satisfiable formulas, addtion of symmetry-breaking predicates – which results in the removal of perfectly good solutions – may seem like a bad idea, since it reduces the chance of stumbling upon a solution early in the search. While this may be true, even for satisfiable formulas symmetry-breaking predicates can help by summarily excluding solutionless regions of searchspace during backtrack search. If a region contains no solutions AND no isomorphism class representatives, it will be quickly excluded where without symmetry-breaking predicates it would have had to be searched.

Moreover, there are situations where we're interested not just in finding a satisfying assignment but in enumerating non-isomorphic satisfying assignments. For instance, one way to check sanity of an Alloy model is to simulate some instances – both to check that the allowed instances are well-formed and to make sure that the instances corresponding to specific scenarios are allowed. Simulation can be much more useful if each simulated instance is "essentially distinct" from the others – that is, not isomorphic to them. Another situation where isomorph elimination during solution enumeration helps is when Alloy is used for test case generation [50, 53]. Eliminating isomorphic test cases results in smaller test suites and reduces the testing time.

The major problem in constructing symmetry-breaking predicate is finding a predicate that is both effective (true of few solutions per isomorphism class) and compact (expressible as a small Boolean formula). In this thesis, we describe methods for generating compact and effective symmetry-breaking predicates, and for measuring predicate effectiveness.

Exploiting subformula sharing

Quantified formulas – statements such as $\forall x P(x)$ – are frequently used in formal specifications. They allow concise and natural formalization of system properties. The user can use quantified formulas to specify a parameterized family of systems, such as a distributed algorithm that works on n-node networks.

For these and other reasons quantified formulas are present in many constraint languages. Languages that permit some form of quantifiers include first-order logic, Alloy [44] and Murphi [16]. The recently developed Bounded Model Checking techniques express Linear Temporal Logic formulas as quantified boolean formulas [9].

While quantified formulas are convenient for writing specifications, they have proved significantly less tractable for automatic analysis than quantifier-free (propositional) formulas. This is not surprising given that the quantified formulas are known to be PSPACEcomplete, unlike propositional formulas which are known to be in NP. Some algorithms (QBF solvers) have been developed for directly determining the consistency of quantified formulas [70, 25]. However, despite several years of advances, these methods are still not competitive with the best propositional solvers such as Chaff [63]. That is, converting a quantified formula to propositional form (by grounding out the quantifiers) and applying a satisfiability solver is typically more efficient than applying a Quantified Boolean Formula solver directly to the quantified formula – as long as the conversion can be done in reasonable time [26]. If the conversion (grounding-out) can't be done in reasonable time, then using a QBF solvers is the only option.

Before a quantified formula can be solved with a propositional solver, the quantifiers must be expanded (grounded out). The ground formula can in general be much larger than the quantified (lifted) formula. The grounding-out of a quantified formula into ground form can become a bottleneck of analysis. For quantified formulas for which grounding-out is infeasible, the only option is to use procedures that work directly with quantified formulas.

One way to mitigate the costs of grounding-out is to represent the ground formula as a directed acyclic graph in which identical subformulas are shared. This can result in significant memory savings. However, grounding out first and determining identical subformulas afterwards often isn't a feasible approach, due to the size of the ground formula.

In this thesis, we develop techniques for converting a quantified (lifted) formula directly into a quantifier-free (ground) formula in the form of a directed acyclic graph, in which identical subtrees are shared. The conversion is performed directly, without first creating a ground form in which identical subtrees are not shared. The technique does not depend on details of the constraint language, and should be applicable whenever there is a need to convert quantified formulas into ground form.

1.5.5 Practical uses of our contributions

This section describes some examples of practical uses of our techniques.

Zave [83] used Alloy to model addressing in interoperating heterogeneous networks.

The model makes extensive use of objectification of complex logical structures. For example, the model includes the following definitions:

```
sig Domain { space: set Address, map: space -> Agent }
sig Agent { attachments: set Domain }
```

Both `Domain` and `Agent` are complex structures: each `Domain` contains a set and a relation, while each `Agent` contains a set. Objectification allows them to be treated as atomic entities; this in term permits them to act as elements of sets and relations. Thus, a `Domain` can contain a relation mapping `Addresses` to `Agents`, while an `Agent` can contain a set of `Domains`.

Khurshid and Jackson [51] have used Alloy to check a published protocol for name resolution in networks, finding several serious bugs. Symmetry-breaking and subformula sharing improve Alloy performance on that model, as experiments in Sections 4.8 and 5.4 illustrate.

Gassend and van Dijk [24] have used Alloy to model security protocols for Controlled Physical Random Functions [23]. In the course of modeling, several overconstraints were identified which were debugged using Alloy's unsatisfiable core extraction. A description of how one such overconstraint was debugged appears in Section 6.2.

Taghdiri and Jackson have used Alloy to model protocols for secure multicast, finding several bugs in a published protocol [77, 78]. The model uses Alloy's support for declarative specification, and expresses analysis problems in the style of Bounded Model Checking. Even though Alloy has no direct support for modeling messaging, the model easily describes protocols involving message exchanges by objectifying messages. The model uses pure-logic modeling to achieve a modular description of the protocols; for example, the portion of the trace relating to each protocol participant is described as part of the specification of the participant, rather than as part of a large global state structure.

Hashii has used Alloy to model a Trusted Security Architecture for Polymorphous Computing [31, 30], and has found security violations in the design. The model was quite large (with 63 signatures, 64 relations and 114 constraint paragraphs), so the analyzer scalability features described here became important. The model included exchange of messages with encryption and authentication; the lack of explicit support for messaging in Alloy was not a hindrance, as all messaging was modeled using objectification. The model included both “dynamic relations” (changing with time) and “static” relations (not changing with time); pure-logic modeling easily accomodated the need to model these different kinds of relations.

Alloy has also been used as a back-end for other tools. TestEra [50, 58], a novel framework for testing Java programs, uses Alloy Analyzer as a back-end. The ability to represent complex structures is used to allow specification of structurally complex tests and accurate modeling of the Java heap. Also, Alloy’s symmetry-breaking ability is used to reduce the size of generated test suites by eliminating isomorphic test cases. Vaziri and Jackson [45, 80] used Alloy as a back-end of their JAlloy tool which checks Java programs. The tool uses the flexibility of Alloy’s pure-logic design in producing Alloy models that represent not a single system state but a bounded trace. The tool also relies on Alloy’s symmetry-breaking ability for efficient analysis.

1.5.6 Summary

This thesis describes a number of model checking techniques that were implemented in the Alloy Analyzer. The result is a model checker with a combination of characteristics unavailable in existing tools. It has extensive support for declarative modeling, including debugging of overconstraints. Its modeling language supports flexible pure-logic modeling, allowing for expression of a variety of analyses including bounded model checking. Algorithms that manipulate complex recursive data structures can be checked. Scalability features, including symmetry-breaking and exploitation of subformula sharing, allow the analyzer to scale to practical models. Table 1.1 compares the key features of existing model checkers with the Alloy Analyzer.

Table 1.1: Feature comparison of Alloy with other model checkers.

feature	SPIN[32]	SMV[11]	AlloyAlpha [42]	Alloy[44]
scalable	y	y	n	y
declarative	n	n	y	y
complex structures[6]	n	n	n	y
trace-based models	y	y	n	y
pluggable backend	n	n	y	y
pure-logic modeling	n	n	n	y
direct support for finite state machines and temporal logic	y	y	n	n

[6] Multiple instances of heterogeneous graph-like data structures, treated as first-class objects; each instance of a complex structure can contain sets and relations on instances of other complex structures.

Chapter 2

Pure-Logic Modeling with Alloy

In this chapter we'll describe the Alloy language, illustrate some common usage patterns with examples, and point out how Alloy's key features facilitate modeling. In particular, the following features of Alloy will be highlighted: pure-logic design; declarative modeling; and support for analyzing systems that manipulate complex structures.

This chapter is organized as follows. First, we describe the core constructs of the Alloy language. Then, we introduce the model of a railway system, which will be our running example. We begin by modeling only the topology of the railway tracks, without the trains. On the example of track topology, we show how to represent instances of a real-world system with a collection of relations, and how to translate informal predicates about the real-world system into formal Alloy constraints on the relations. Then we show how to use relations to model complex data structures, on the example of modeling routes through the tracks. We point out the generality of structures that can be represented; in particular, we can represent self-referential and mutually-referential structures such as lists, trees and other graphs.

Next, we show how dynamic aspects of the system can be modeled by using our technique for representing complex structures to represent instances of the system state. We show how Alloy's pure-logic design allows a variety of analyses to be expressed in a uniform manner. Specifically, we will look at the following analyses: checking that an operation preserves an invariant; checking that two specifications of an operation are equivalent; and Bounded Model Checking (BMC) [9].

2.1 Core elements of Alloy models

The core of an Alloy model consists of the following elements:

- *basic types* – disjoint finite sets of uninterpreted, indistinguishable atoms.
- *relations* – relation-valued variables. Each relation has a *relation type* which is a tuple of basic types. The value of a relation is a subset of the direct product of the basic types in its relation type; in other words, a set of tuples of atoms, with each atom in each tuple drawn from the corresponding basic type of the relation. An assignment of relational values to all relations is called an *instance*.
- *predicate* – a predicate on instances, expressed as a formula in first-order logic.

The actual sizes of the basic types (called *scopes*[1]) are not part of the model, but are specified for analysis. The Alloy Analyzer answers the question "for the given scope, does there exist an instance satisfying the predicate?" The universe of instances for a particular analysis is determined by the basic type scopes.

2.2 Railway example

Let us illustrate Alloy's core concepts on a model of a railway system, which we'll use as a running example throughout this thesis. The model used here takes a number of modeling ideas from prior railway modeling efforts [10, 82, 46]. The full model appears in Appendix A; in this chapter we will develop the most important parts.

The model will be developed incrementally; one of the strengths of Alloy is good support for incremental development. We'll first model the railway track topology, without the trains. This will be used to illustrate the main techniques of relational modeling. We'll then extend the model to include trains and to describe safe rules for train movement.

[1] Not to be confused with lexical scopes of variables. When there is an ambiguity, we will refer to basic type scopes as "analysis scopes".

2.2.1 The railway domain

First, let us describe the railway domain in informal language. A railway track consists of a collection of connected rectangular *units*. A unit has two opposite sides, left and right. On two opposite sides of each unit there are *connector*s, at least one on each side. Units are attached to each other at connectors; at most two units share any given connector. Besides being the basic building blocks of the railway track, the units are also used to define safe separation of trains. For safe train operation, at most one train can be in a unit at a given time. Train cars in different units cannot collide.

With each unit is associated a set of possible *paths* through the unit. (Here, a *path* spans exactly one unit; concatenations of paths will be modeled later and will be called *routes*.) Each path represents a possible movement of a train through the unit, and joins two connectors on opposite sides of the unit. At any given time, some subset of paths through the unit may be *open* (available to trains). Paths are undirected.

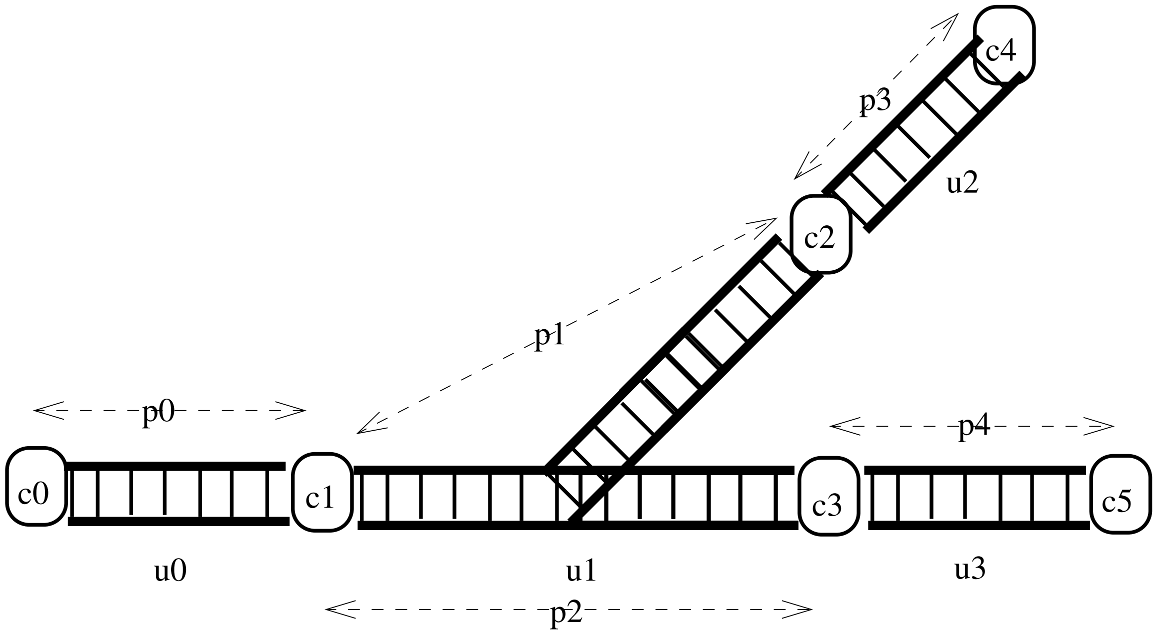


Figure 2-1: Sample instance of the railway model.

A sample railway track is shown in Figure 2-1. It consists of four units u_0, u_1 and u_2 and u_3. All units are simple linear units consisting of one path, except for u_1 which is a junction with two possible paths (p_1 and p_2).

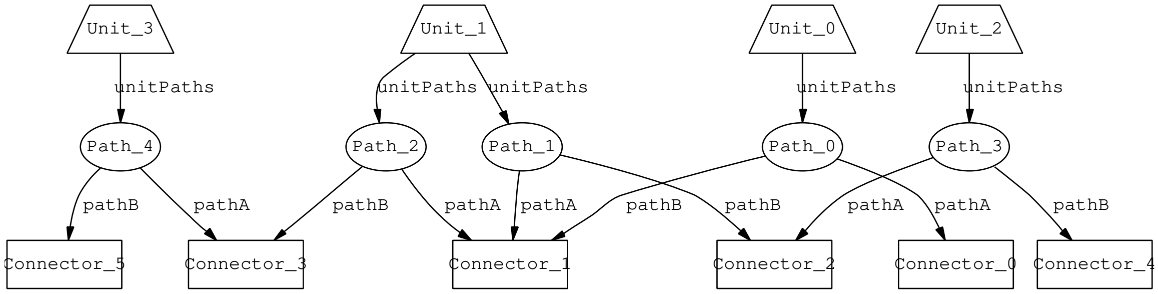


Figure 2-2: Relational view of the railway instance in Figure 2-1.

2.2.2 Alloy representation of the railway domain

We can formalize our informal description of railway topology by constructing an Alloy model, as follows. We can declare basic types `Unit`, `Connector` and `Path` to represent the basic physical elements of the railway track. We can then declare several relations among these basic types, as follows:

Relation(s)	Relation type	Meaning/interpretation
`unitConnsA`, `unitConnsB`	`Unit -> Connector`	$\langle u_i, c_j \rangle \in$ `unitConnsA(unitConnsB)` iff connector c_j is on the left (right) side of unit u_i
`unitPaths`	`Unit -> Path`	$\langle u_i, p_j \rangle \in$ `unitPaths` iff path p_j connects two connectors on opposite sides of unit u_i
`pathA`, `pathB`:	`Path -> Connector`	$\langle p_i, c_j \rangle \in$ `pathA` and $\langle p_i, c_k \rangle \in$ `pathB` iff path p_i runs between connectors c_j and c_k (which are on opposite sides of some unit)

Under this formalization, the railway track in Figure 2-1 corresponds to the following instance (illustrated in Figure 2-2 using Alloy Analyzer's visualization facility):

`unitConnsA`=$\{\langle u_0, c_0\rangle, \langle u_1, c_1\rangle, \langle u_2, c_2\rangle, \langle u_3, c_3\rangle\}$

`unitConnsB`=$\{\langle u_0, c_1\rangle, \langle u_1, c_2\rangle, \langle u_1, c_3\rangle, \langle u_2, c_4\rangle, \langle u_3, c_5\rangle\}$

`unitPaths`=$\{\langle u_0, p_0\rangle, \langle u_1, p_1\rangle, \langle u_1, p_2\rangle, \langle u_2, p_3\rangle, \langle u_3, p_4\rangle\}$

`pathA`=$\{\langle p_0, c_0\rangle, \langle p_1, c_1\rangle, \langle p_2, c_1\rangle, \langle p_3, c_2\rangle, \langle p_4, c_3\rangle\}$

`pathB`=$\{\langle p_0, c_1\rangle, \langle p_1, c_2\rangle, \langle p_2, c_3\rangle, \langle p_3, c_4\rangle, \langle p_4, c_5\rangle\}$

2.2.3 Alloy constraints

While every real railway track configuration maps to an instance of our Alloy model, not every instance represents a reasonable track configuration. For example, in a plausible configuration, at most two units can share a connector. We can restrict our instances to those representing valid track configurations, by writing Alloy constraints. Some basic constraints that ensure well-formedness of modeled tracks appear below.

```
fact BasicUnitConstraints {
  all u: Unit | {
     // each side of the unit has at least one connector
     some u.unitConnsA  &&  some u.unitConnsB
     // the two sets of connectors (left and right) are disjoint
     no u.unitConnsA & u.unitConnsB
     // each path in a unit connects a left connector
     // to a right connector
     all p: u.unitPaths |
        p.pathA in u.unitConnsA && p.pathB in u.unitConnsB
     // units are rectangular, with connectors on opposite
     // sides of the rectangle, so when this unit
     // connects to another unit,  only one side of
     // this unit is used.
     // in other words, no other unit can touch both
     // sides of this unit.
     all otherUnit: Unit - u | {
        let sharedConns =
           u.(unitConnsA + unitConnsB) &
           otherUnit.(unitConnsA + unitConnsB) |
           sharedConns in u.unitConnsA ||
           sharedConns in u.unitConnsB
  }
}

fact BasicPathConstraints {
  all p: Path | {
    // each path belongs to exactly one unit
    one unitPaths.p
    // each path has exactly one connector at each end
    one p.pathA  &&  one p.pathB
    // path atoms are canonicalized: only one path
    // atom per connector pair
    all otherPath: Path - p |
      (otherPath.pathA = p.pathA &&
       otherPath.pathB = p.pathB) => otherPath = p
  }
}

fact BasicConnectorConstraints {
  // At most two units share a connector
  all c: Connector | (# (unitConnsA + unitConnsB).c) < 3
}
```

Note that we describe the possible railway topologies declaratively – by writing predicates that are true of correct topologies. This allows the description to be simple and intuitive. In an imperative model checker [32, 11, 16], we would have to specify, in the model checker's language, an effective procedure for *generating* all possible topologies (as opposed to simply testing the validity of a particular topology). Ensuring that the code correctly generates *all* possible valid topologies could be non-trivial, especially if we seek to eliminate isomorphic topologies. Alloy's built-in symmetry-breaking mechanism described in Chapter 4 will automatically eliminate most isomorphs during search.

The syntax and semantics of Alloy constraints are given in Figure 2.2.3. In our example we have used Alloy constraints to impose basic validity constraints; further on we will describe other uses of constraints.

$problem ::= decl^{*}\ formula$	$M : formula \rightarrow env \rightarrow boolean$
$decl ::= var : rtype$	$X : expr \rightarrow env \rightarrow value$
$rtype ::= btype^{+}$	$env = var \rightarrow value$
	$value =$ set of tuples of atoms
$formula\ ::=$	
$\mid expr\ in\ expr$	$M[a\ in\ b]\ e = X[a]\ e \subseteq X[b]\ e$
$\mid !formula$	$M[!F]\ e = \neg M[F]\ e$
$\mid formula\ \&\&\ formula$	$M[F\&\&G]\ e = M[F]\ e \wedge M[G]\ e$
$\mid formula \Vert formula$	$M[F \Vert G]\ e = M[F]\ e \vee M[G]\ e$
$\mid all\ v : rtype \mid formula$	$M[all\ v : t \mid F]\ e = \bigwedge\{M[F](e \oplus (v \mapsto \{x\})) \mid x \in t\}$
$\mid some\ v : rtype \mid formula$	$M[some\ v : t \mid F]\ e = \bigvee\{M[F](e \oplus (v \mapsto \{x\})) \mid x \in t\}$
$expr\ ::=$	$X[a+b]\ e = X[a]\ e \cup X[b]\ e$
$\mid expr\ +\ expr$	$X[a\&b]\ e = X[a]\ e \cap X[b]\ e$
$\mid expr\ \&\ expr$	$X[a-b]\ e = X[a]\ e \setminus X[b]\ e$
$\mid expr\ -\ expr$	$X[a.b]\ e = \{(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m) \mid$
$\mid expr\ .\ expr$	$\exists x_k.(x1, \ldots, x_k) \in X[a]\ e \wedge (x_k, \ldots, x_m) \in X[b]\ e\}$
$\mid \tilde{\ }expr$	$X[\tilde{\ }a]\ e = \{(x, y) \mid (y, x) \in X[a]\ e\}$
$\mid \hat{\ }expr$	$X[\hat{\ }a]\ e =$ the smallest r such that $r.r \subseteq r \wedge X[a]\ e \subseteq r$
	$X[\{v : t \mid F\}]\ e = \{x \in e(t) \mid M[F](e \oplus v \mapsto \{x\})\}$
	$X[v]\ e = e(v)$

Figure 2-3: Alloy core constructs: syntax, type rules and semantics.

2.3 Modeling complex structures

2.3.1 Representing complex structure instances with atoms

In our railway example, paths and units are not atomic entities; they have internal structure. A path has two endpoints; a unit has two disjoint sets of connectors and a set of paths joining connectors from opposite sides. Yet in the underlying Alloy model, paths and units are treated as atoms – just like connectors, which lack internal structure. The internal structure of units is represented by the relations `unitConnsA: Unit -> Connector`, `unitConnsB: Unit -> Connector` and `unitPaths: Unit -> Connector`.

For a given unit represented by a particular atom of `Unit`, the pieces of that unit's structure can be referenced by taking the relational image, under one of these relations, of a singleton set containing the atom. This lets us write constraints on the complex structure represented by each atom. For example, we can require that the left and right connector sets of each unit be disjoint by writing `no u.unitConnsA & u.unitConnsB`, where `u` is a singleton set containing the `Unit` atom representing the unit in question. But since each unit is still represented by an atom, we can write existential and universal quantifiers over units, and the formula will remain first-order. [2] Also, since each unit is represented by an atom, relations involving units remain first-order structures – sets of tuples of atoms, rather than sets of tuples of complex structures. This in turn allows the components of units to reference other complex structures (such as paths). Thus, we can define mutually referencing or self-referencing complex structures, and yet manipulate them as if they were simple atoms.

2.3.2 Signatures

To emphasize this view of Alloy models, a new language construct called a *signature* was introduced [36, 44]. In its simplest form, a signature is a basic type together with a group of relations which have that basic type as their first column. With signatures, the railway declarations above look as follows:

```
sig Unit {
  unitConnsA, unitConnsB: set Connector,
  unitPaths: set Path
}

sig Connector { }
```

[2]The quantification is over the atoms in the instance, rather than over all possible values of the complex structure. That is, each unit includes a pair of sets of connectors; but any given instance will have Unit atoms corresponding only to some of the possible pairs of sets of connectors. The quantification is over the unit atoms, rather than over all possible pairs of sets of connectors.

```
sig Path {
   pathA, pathB: Connector
}
```

These declarations define the same basic types and relations as before, but now the view of paths and units as complex structures – and the role of relations as "fields" of these structures – becomes clearer. Note that the declaration of a binary relation appears in a signature as a declaration of a unary relation (a set): for every atom of `Unit` the relation `unitPaths` gives a set of `Path` atoms.

The field declarations, in addition to declaring new relations, can specify some common well-formedness constraints on these relations. The default declaration of a binary relation constrains that relation to be a total function from the signature's basic type. For example, the declarations of binary relations `pathA` and `pathB` implicitly specify that each path is mapped to exactly one connector by each of these relations. The declaration of `unitPaths` includes the keyword `set` to disable this constraint.

2.3.3 Inheritance

.

In addition to emphasizing the view of basic type atoms as instances of complex structures, signatures provide an inheritance mechanism. Suppose we wanted to say that some `Units` are linear units, with one connector on each side. We could do this by writing

```
sig LinearUnit extends Unit { }
fact LinearUnitStructure {
  all lu: LinearUnit | one lu.unitConnsA && one lu.unitConnsB
}
```

The signature `LinearUnit` is a *subsignature* of `Unit`, and does not define a new basic type. Instead, it defines a subset of `Unit` (i.e. a unary relation with relation type `Unit`), called `LinearUnit`. In a given instance, `LinearUnit` will hold a set of atoms of basic type `Unit` that is a subset of the set of atoms in the set `Unit`. The fact `LinearUnitStructure` says that each linear unit has one left connector and one right connector [3]. Note that each

[3]Facts are given names for clarity purposes, and also to make it easier to convert a fact into a function.

atom of `LinearUnit` is also an atom of `Unit`, so we can use the fields of `Unit` to reference components of structure represented by `LinearUnit` atoms. [4] Subsignatures can also define their own fields. For example, we could define another kind of unit representing junctions:

```
sig JunctionUnit extends Unit {
   mainLine, sideLine: Connector
}

fact JunctionStructure {
   all ju: JunctionUnit | {
     one ju.unitConnsA
     ju.unitConnsB = ju.mainLine + ju.sideLine
     ju.mainLine != ju.sideLine
   }
}
```

The relations `mainLine` and `sideLine`, of relation type `Unit -> Connector`, map `Unit` atoms that are in the set `JunctionUnit` to the two right-side connectors of a junction, and map other `Unit` atoms to empty set. The one left-side connector of a junction can still be referenced using the relation `unitConnsA`.

2.3.4 Modeling "logical" complex structures

So far we have used basic types for representing physical elements such as units or connectors. We can also use basic types to represent "logical" structures. Continuing the railway example, let's introduce a basic type `Route` to represent a sequence of adjacent `Paths`:

```
open std/ord
open std/seq

sig Route {
   routePaths: SeqIdx -> Path,
   firstConn, lastConn: Connector
}
```

Here we make use of two standard Alloy modules: one for describing total orders (`std/ord`), and one for describing sequences (`std/seq`). We give a brief description of these modules sufficient to understand their use in the railway example.

[4] We can also use fields of a subsignature (LinearUnit) without casts on members of its supersignature (Unit), e.g. u.mainLine; for atoms of Unit that are not atoms of LinearUnit, this will denote the empty relation.

The total ordering module `std/ord` defines, for each basic type on which one of the module's functions (described below) is invoked, a total ordering on the atoms of that basic type. For a basic type `T`, `OrdFirst(T)` denotes the first atom in the order; `OrdLast(T)` denotes the last atom; `OrdNext(i)` denotes the successor atom of the single `T` atom contained in the set `i`, or the empty set if `i` contains `OrdLast(T)`; `OrdFirst(i)` denotes the predecessor atom of the single `T` atom contained in the set `i`, or the empty set if `i` contains `OrdFirst(T)`. While use of the `std/ord` module adds several relations for each totally ordered basic type to represent the total order on that type, special-case symmetry-breaking described in Section 4.5.5 ensures that the addition of these relations does not adversely affect analysis efficiency.

`SeqIdx` is a basic type defined in the module `std/seq`. On the atoms of this basic type, a total order relation is defined using the module `std/ord`. A sequence of paths is represented as a functional binary relation of type `SeqIdx -> Path`. For instance, the sequence `[Path_0,Path_2,Path_5]` would be represented as `{<SeqIdx_0,Path_0>,<SeqIdx_1,Path_2>,<SeqIdx_2,Path_5>}`.

The relevant functions from the `ord/seq` module are given below.

```
// does s represent a valid sequence?
fun SeqFunValid(s: SeqIdx ->? Path)
// the set of indices of items in the sequence
fun SeqFunInds(s: SeqIdx ->? Path): set SeqIdx
// the item at the given index; {} if none
fun SeqFunAt(s: SeqIdx ->? Path, i: SeqIdx): option Path
// index of the last item in a sequence; {} if s is empty
fun SeqFunLastIdx(s: SeqIdx ->? Path): option SeqIdx
// last element of a sequence; {} if s is empty
fun SeqFunLast(s: SeqIdx ->? Path): option Path
// set of elements in the sequence
fun SeqFunElems(s: SeqIdx ->? Path): set Path
// sequence without its first element
fun SeqFunRest(s: SeqIdx ->? Path): SeqIdx ->? Path
// a given sequence with a new element at the end
fun SeqFunAdd(s: SeqIdx ->? Path, e: Path): SeqIdx ->? Path
// concatenation of two sequences, truncated to |SeqIdx|
fun SeqFunConcat(s1, s2: SeqIdx ->? Path): SeqIdx ->? Path
```

In the above declarations, `{}` denotes the empty relation. Functions in Alloy are simply parameterized macros; function invocations are replaced by the function body with formal arguments replaced by actual ones. Functions can denote either Boolean predicates (in which case no return type is given) or relations (in which case the return type is given at the end of the function declaration.) Since Alloy is a declarative language in which functions are inlined rather than invoked, it is more accurate to say that a function "denotes" a Boolean or relational value rather than "returns" it; but we'll say "returns" by analogy with ordinary programming-language functions. The declarations of function arguments and function return types include relation type and multiplicity. The declaration `i: SeqIdx` means that `i` is a unary relation containing exactly one tuple; the declaration `s: SeqIdx ->? Path` means that `s` is a binary relation mapping each atom of `SeqIdx` to at most one atom of `Path`; In the function declarations, `SeqIdx ->? Path` denotes a binary relation which maps each atom of `SeqIdx` to at most one atom of `Path`. A function return type of `option Path` means the function returns a unary relation containing at most one tuple.

The relation `routePaths` represents the sequence of paths comprising the route. `firstConn` and `lastConn` give the first and last connectors of the route. Constraints requiring the route to be well-formed can be written as follows:

```
fact RoutesWellFormed {
  all r: Route | {
    // routePaths represents a valid sequence of Paths
    SeqFunValid(r.routePaths)

    // adjacent Path's in the sequence must share
    // an endpoint, and be from different units
    all i: SeqFunInds(routePaths) - OrdFirst(SeqIdx) |
      let p = SeqFunAt(routePaths, i),
          p_prev = SeqFunAt(routePaths, OrdPrev(i)) | {
         some p.pathConns & p_prev.pathConns
         unitPaths.p != unitPaths.p_prev
      }

    // the first connector is the connector of the
    // first path that is not a connector of the second path
    firstConn in
      SeqFunAt(routePaths, OrdFirst(SeqIdx)).pathConns -
      SeqFunAt(routePaths, OrdNext(OrdFirst(SeqIdx))).pathConns
    // the last connector is the connector of the
    // last path that is not a connector of the
    // next-to-last path
    lastConn in
      SeqFunAt(routePaths,
               SeqFunLastIdx(routePaths)).pathConns -
      SeqFunAt(routePaths,
               OrdPrev(SeqFunLastIdx(routePaths)).pathConns

    // first and last connector are distinct;
    // must specify this explicitly because for routes consisting
    // of one path, the above two constraints  don't imply this
    firstConn != lastConn
  }
}
```

2.3.5 Generality of our complex-structure representation

We could define a hierarchy of routes, in which a route can consist of shorter sub-routes, which in turn might consist of yet shorter sub-routes. Such a representation might be useful in planning the movement of trains through the station. We could represent this situation by adding new fields to `Route`:

```
sig RouteWithSubroutes extends Route {
   leftHalf, rightHalf: Route
}

fact SubroutesCompriseRoute {
    all r: RouteWithSubroutes |
        // the full route is a concatenation of its halves
        r.routePaths =
          SeqFunConcat(r.leftHalf.routePaths,
                       r.rightHalf.routePaths)
}
```

Note that the complex logical structure `Route` now contains references to other instances of the structure (Figure 2-4). In other words, the generality of structures that can be represented includes what can be represented on a heap in an object-oriented language such as Java. This, together with the fact that Alloy does not limit its models to representing finite state machines, has enabled the use of Alloy for checking the code of heap-manipulation procedures [80], generating tests for heap-manipulating programs from specifications [58] and checking runtime conformance of object-oriented programs to specifications [13], in addition to checking Alloy models involving heap-like structures. By contrast, the complex structures that can be represented in most other model checkers [33, 11] are limited to structures typically represented on the stack: complex structures can contain smaller structures in their entirety, but cannot contain references to instances of other (possibly larger) complex structures. While some explicit-state model checkers can handle heap-allocated structures [34, 64], no declarative, symbolic or pure-logic model checker with that capability exists.

2.4 Modeling system state

So far, we have only modeled the train topology and the route structure, without modeling the trains or the change of system state over time. (We have modeled the *possible* states of

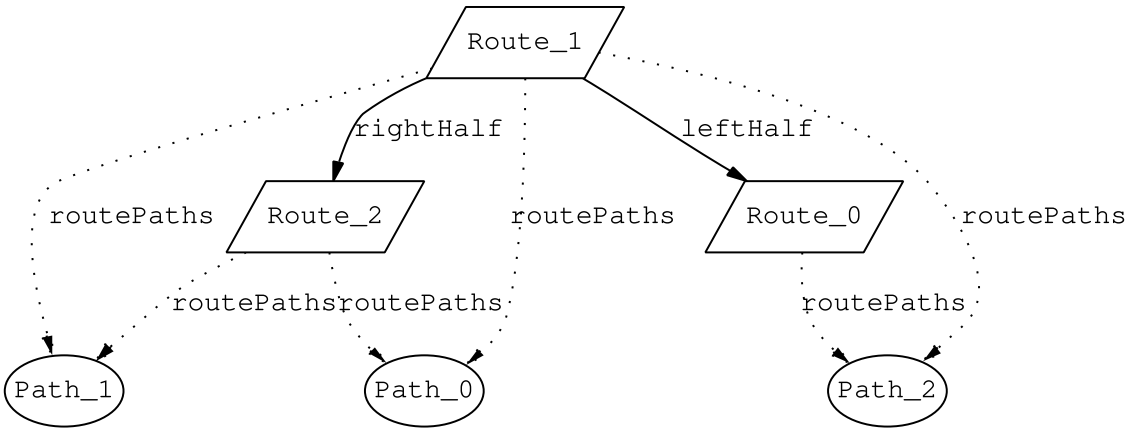


Figure 2-4: An Alloy instance with heap-like data structures.

each unit by modeling the paths that make up the unit, but we have not modeled the actual state of the units at various points in time. Recall that the state of a unit consists of the open paths in that unit.) Alloy's support for incremental modeling, and for describing instances that are not just finite state machines, enabled us to construct a non-trivial description of this part of the problem without worrying about the other parts. We will now turn to modeling some dynamic aspects of the railway.

2.4.1 Modeling a single copy of system state

Let us expand our railway model by adding trains, and modeling the locations of the trains and the status of the units. We will first model the state of the system at a single point in time; later we will see how pairs and sequences of states can be modeled. The additional definitions are as follows:

```
sig Train { trainLoc: Route }
sig OpenPaths extends Path { }
```

The relation `trainLoc` gives the position of each train; following [10], we model the location of a train as a `Route` covering the tracks occupied by the train. The set (i.e. unary relation) `OpenPaths` gives the paths that are open for trains. A path is open iff a train can occupy the entire path from one connector to the other, while standing correctly aligned with the physical tracks. For example, the sole path of a linear unit might be always open, while in a junction one of two paths might be open at a given time. The operation of linear and junction units can be specified with constraints such as the following:

```
// The path joining the two given connectors,
// or the empty set if no such path exists.
fun PathBetween(a, b: Connector): option Path {
   result = { p: Path | p.pathA = a && p.pathB = b }
}

fact HowUnitsOperate {
   // the one path of a simple linear unit is always open
   all lu: LinearUnit | lu.unitPaths in OpenPaths
   // in a junction, exactly one of two paths is open
   // at any time:
   all ju: JunctionUnit |
      // either the path joining the one left connector to the
      // ``main line'' connector on the right...
      (ju.unitPaths & OpenPaths) =
      PathBetween(ju.unitConnsA, ju.mainLine) ||
      // ...or the path joining the one left connector to the
      // ``side line'' connector on the right.
      (ju.unitPaths & OpenPaths) =
      PathBetween(ju.unitConnsA, ju.sideLine)
}
```

Upon defining relations describing the system state, we can write predicates describing particular kinds of states. For example, we could define a *safe* state: one where no two trains occupy the same unit, and all trains reside only on open paths.

```
fun RouteUnits(r: Route): set Unit {
   result = SeqFunElems(r.routePaths).unitPaths
}

fun SafeState() {
   // no two trains are in the same unit
   all t1: Train, t2: Train - t1 |
      no RouteUnits(t1.trainLoc) & RouteUnits(t2.trainLoc)
   // trains reside only on open paths
   let trainLocs = Train.trainLoc,
       occupiedPaths = SeqIdx.(trainLocs.routePaths) | {
       occupiedPaths in OpenPaths
   }
}
```

2.4.2 Modeling several copies of system state

So far we have modeled the state of the train station at a single point in time, describing the state by a group of relations. For model-checking purposes, we often want to model several instances of state. For example, to check whether a particular set of rules for train movement guarantees that the system will not pass from a safe state to an unsafe state, we would need to represent *two* states: the pre-state and the post-state. We would then write constraints saying that the pre-state is safe, the post-state unsafe, and the (pre,post)-state pair is a transition conforming to the rules being checked. Another scenario which requires representing multiple copies of state is Bounded Model Checking (BMC) [9], which will be discussed in greater detail later in this chapter.

To represent multiple copies of state in the model, we could "objectify" the state by declaring a new basic type whose atoms represent instances of state:

```
sig State {
   trainLoc: Train ->! Route,
   openPaths: set Path
}
```

Note that the relations `trainLoc: Train -> Route` and `OpenPaths: Path` have been replaced respectively by `trainLoc: State -> Train -> Route` and `openPaths: State -> Path`. (The `->!` notation adds the constraint that in each `State`, each `Train` is at exactly one `Route`.) Like atoms of `Route`, atoms of `State` represent instances of a complex logical structure. This structure represents the state of the train system – the location of each train, and the state of each unit – at one time point. (It's important to note that no notion of state is built into Alloy, and the `State` signature we have defined is just a regular Alloy signature without any special semantics.) By varying the scope of `State` we can ensure that the model can represent a sufficient number of distinct system states.

The constraints we have written for a single state can be rewritten for the new representation, by using relational image to access the state components which previously were accessed as global relations:

```
fun OccupiedPaths(s: State): set Path {
   result = SeqIdx.((Train.(s.trainLoc)).routePaths)
}


fun SafeState(s: State) {
   // no two trains are in the same unit
   all t1: Train, t2: Train - t1 |
      no RouteUnits(t1.(s.trainLoc)) & t2.(s.trainLoc)
   // trains reside only on open paths
   let trainLocs = Train.(s.trainLoc),
       occupiedPaths = SeqIdx.(trainLocs.routePaths) |
       occupiedPaths in s.openPaths
}
```

2.4.3 Objectifying complex structures

In general, it is possible to take any collection of relations representing a complex structure (such as system state here), and create a new signature representing instances of that complex structure. Besides making the number of instances of the structure an easily controlled parameter of the model, all the previously listed benefits of representing complex structures with atoms will be available: the ability to declare relations involving the complex structure, and to reference instances of this complex structure from other complex structures. Any predicates on the complex structure modeled with a collection of global relations can be rewritten for the "objectified" version of that complex structure, as was done for system states above.

2.4.4 Data abstraction through objectification

Note that objectification of a complex structure, together with the use of Alloy functions, can facilitate data abstraction. In the above example, `OccupiedPaths(s)` denotes the set of paths occupied by trains in a given state. We could change the representation of state, e.g. by adding an explicit field to represent the occupied paths:

```
sig State {
   // ...
   occPaths: set Path
}

fact DefineOccPaths {
   all s: State |
     s.occPaths = SeqIdx.((Train.(s.trainLoc)).routePaths)
}
```

We could then redefine the function `OccupiedPaths` as

```
fun OccupiedPaths(s: State): set Path { result = s.occPaths }
```

All the uses of `OccupiedPaths` throughout the model would then still retain their meaning and would not be changed, even though the definition of `State` has changed. In this way, objectification of complex structures lets us encapsulate the representation of the complex structure.

2.4.5 Fine-grained control over search space size

Representing logical structure instances by basic type atoms enables fine-grained control over the space of instances representable by the model. Basic control is achieved by changing the length of the traces (by changing |`State`|) or changing the number of physical elements (e.g. by changing |`Train`|). More fine-grained control can be achieved by changing the number of distinct instances of a complex logical structure that can appear in a solution. For example, for |`Route`|`=2` the railway model can represent traces with up to two distinct instances of the route structure. This includes the trace in which two trains start out on adjacent paths and then one train moves into the other's path while the other remains in place:

```
trainLoc={<s0,t0,r0>,<s0,t1,r1>,
          <s1,t0,r0>,<s1,t1,r0>}
routePaths={<r0,p0>,<r1,p1>}
```

However, this does not include the trace with the same initial state, but in which one train extends to occupy both tracks.

```
trainLoc={<s0,t0,r0>,<s0,t1,r1>,
          <s1,t0,r0>,<s1,t1,r2>}
routePaths={<r0,p0>,<r1,p1>,<r2,p0>,<r2,p1>}
```

Increasing the scope of a complex logical structure like `Route` may increase the space of representable instances in a more gradual way than increasing the scope of a physical element like `Train`. Fine-grained control over the space of representable instances is important, because the size of this space can make a large difference in analyzability during model-checking. The size of the space also matters when Alloy is used as a backend to a test-case generation tool [58], since the number of solutions directly affects the testing time. While the space of representable instances is not the same as the effective search space (since many instances may be quickly ruled out by constraints), the number of representable instances is strongly correlated with the actual search space size.

2.5 Specifying the transition relation

Now that we can model several instances of state, we can use this ability to specify the rules of train movement by writing predicates on pairs of states to test whether the pair represents a valid train movement. Train movement constraints fall into two broad groups: laws of physics (e.g. trains move only along existing tracks and only to adjacent tracks), and rules of the road (e.g. trains don't run red lights). The declarative nature of Alloy lets us address these groups in a modular and incremental way; physics constraints can be specified separately from "rules of the road" constraints.

2.5.1 Constraints arising from physics

Let us first specify a physics constraint. First, let's specify a constraint on tracks. Of all paths in a given unit emanating from a given connector, only one can be open at a time. This reflects a physical property of rails: a train entering a unit at a connector will deterministically end up on one particular path in that unit emanating from that connector.

```
fact TrackPhysics {
  all s: State, u: Unit, c: Connector |
     sole pathConns.c & u.unitPaths & openPaths
}
```

Next, let's specify the possible movement of trains. A train moves by adding a new `Path` at its front and dropping the `Path` at its tail.

```
// The set of paths occupied by the given train
// in the given state.
fun TrainPaths(s: State, t: Train): set Path {
   result = SeqFunElems((t.(s.trainLoc)).routePaths)
}

fun TrainPhysics(s, s': State) {
   let r = t.(s.trainLoc), r' = t.(s'.trainLoc) | {
     r'.lastConn != r.lastConn
     let openPathsAtTrainFront =
         (pathConns.(r.lastConn) & s.openPaths),
         newPath = openPathsAtTrainFront - TrainPaths(s, t) | {
         some newPath
         r'.routePaths =
           SeqFunAdd(SeqFunRest(r.routePaths), newPath)
     }
   }
}
```

2.5.2 Constraints arising from signalling rules

To specify rules-of-the-road constraints, let's first expand the state to include train signals:

```
sig State {
   // ...
   // may a new train enter this unit through this connector?
   mayEnter: Unit -> Connector
}
```

The relation `mayEnter` represents whether in a given state, a new train may enter a given unit through a given connector. We can constrain train movement to be consistent with the value of `mayEnter`, as follows:

```
fun TrainsObeySignals(s, s': State) {
   all t: Train |
    let r = t.(s.trainLoc), r' = t.(s'.trainLoc) |
       (
          // if the last path of the new
          // train location is different from what it was...
          SeqFunLast(r'.routePaths) !in
          SeqFunElems(r.routePaths) =>
          // then the train had a right to enter
          // the specified unit.
          r.lastConn in
           (unitPaths.SeqFunLast(r'.routePaths)).(s.mayEnter)
       )
}
```

2.5.3 Specifying a railway control policy to check

Finally, we need to specify how signals and units are controlled in response to train movement. This is the part of the railway system that we actually want to check for correctness. The logic that controls signal colors and unit states (e.g. junction positions) should ensure that trains do not collide and do not derail.

For the signal policy, let's use the following rule: if a path is occupied by a train, that path may not be entered via either of its connectors. The intent is to avoid train collisions.

```
fun SignalPolicy (s: State) {
   // if a path is occupied by a train, it may not be entered
   // via any connector
   all c: Connector, u: Unit |
      c in u.(s.mayEnter) =>
        no pathConns.c & u.unitPaths & OccupiedPaths(s)
}
```

For the unit policy, let us use the trivial policy that the set of open paths never changes. That is, no junctions are ever switched. This (together with `fact TrackPhysics` defined above) will ensure that no derailments happen because of changing unit state. Only train collisions caused by inadequate signalling will remain as possible problems with the railway.

```
fun UnitPolicy(s, s': State) { s'.openPaths = s.openPaths }
```

2.6 Invariant preservation testing

Having defined the transition relation, we can now specify and check some dynamic properties of our railway system. One common type of analysis involves testing whether an operation preserves an invariant. In our case, we can test whether our signalling policy and unit policy successfully prevent the railway system from passing from a safe state into an unsafe state:

```
fun SafetyInvariantViolation(s, s': State) {
   TrainPhysics(s, s')
   TrainsObeySignals(s, s')

   SignalPolicy(s)
   UnitPolicy(s, s')

   SafeState(s)
   !SafeState(s')
}
```

To search for counterexamples illustrating a safety invariant violation, we need to specify the basic type scopes within which to search. This is done with the `run` command, which specifies the function to be analyzed and the scope for each basic type:

```
run SafetyInvariantViolation for 2 Unit, 5 Connector, 3 Path,
    3 Route, 1 SeqIdx, 2 Train, 2 State
```

The tool will add two new unary relations `s,s': State` for the arguments of `SafetyInvariantViolation`, and will search for an instance satisfying `SafetyInvariantViolation(s,s')` (as well as all the facts and all constraints implicit in signature declarations). In this analysis, the tool finds no counterexamples.

2.6.1 Unsatisfiable core analysis: debugging overconstraints

Our last analysis found no counterexamples. This could mean one of several things. It could be that our rules for preventing train collisions are indeed correct. Another possibility is that a counterexample exists, but only in a larger scope. In this case, increasing the scope to `3 Unit, 5 Path` still finds no counterexample.

Yet another possibility is that the model contains an unintended overconstraint that masks a counterexample. When we created the model, we established a correspondence between real-world scenarios and model instances (assignments to all relations). We then wrote constraints restricting model instances to those that represent well-formed real-world scenarios in which something bad happens (e.g. trains collide). It's possible that in writing these constraints, we have unintentionally excluded some well-formed real-world scenarios that would illustrate errors. An error in our model would thus prevent us from discovering an error in the actual system.

To deal with this problem, Alloy provides an *overconstraint debugger*. When an Alloy analysis finds no instances, the debugger can identify a subset of the model (an *unsatisfiable core*) sufficient to rule out all satisfying instances. Constraints not in the core are not needed to establish the absence of counterexamples to the property; removing them won't make the model satisfiable. Since most constraints were written by the user for a particular purpose, the fact that some constraints are irrelevant may alert the user to errors in the model.

In our example, invoking the overconstraint debugger produces a surprising result: the constraints `TrainsObeySignals(s,s')` and `SignalPolicy(s)` are irrelevant (see Figure 2-5). We can verify this by commenting them out and re-running the analysis; indeed, no counterexamples are produced. This suggests an error in the model: the signalling constraints were written to exclude some specific scenarios (train collisions), yet these scenarios are ruled out even without the help of these constraints. That means that the remaining constraints erroneously exclude more scenarios than intended. At the very least, they erroneously exclude the collision scenario we had thought of (and wrote `TrainsObeySignals` and `SignalPolicy` to prevent); it's possible that they also inadvertently exclude other bad scenarios that we hadn't thought of, and that our signalling rules would not prevent. Unless we identify and fix the overconstraint, we cannot be sure that our `SignalPolicy` is safe.

In addition to alerting the user to the presence of overconstraint, the debugger can help identify the erroneous constraints. In case of severe overconstraint where a few constraints contradict each other and rule out all instances, the debugger will usually identify the problem constraints and report that the bulk of the model is irrelevant. In our example however, the debugger says that most of the model is relevant to showing absence of counterexamples – only the function invocations `TrainsObeySignals` and `SignalPolicy` are irrelevant. Still, this information can help us find the overconstraint. We have written these functions for the purpose of excluding some specific error scenarios. These scenarios are being erroneously excluded by other constraints; the tool can tell us by which ones. The tool lets us manually enter a specific instance, and see which constraints (if any) the in-

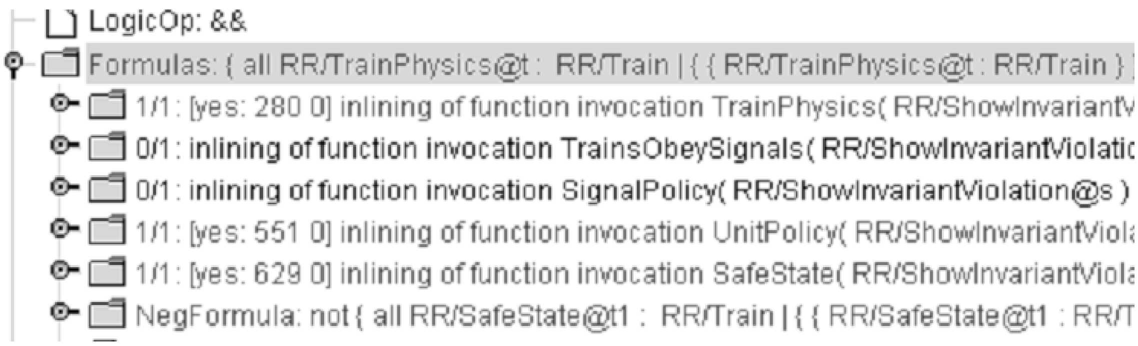


Figure 2-5: Debugging of overconstraints: identification of irrelevant constraints. The figure shows a fragment of the Abstract Syntax Tree of the Alloy model, with markings indicating which branches (subformulas) are relevant to showing absence of counterexamples and which are irrelevant. Branches beginning with "[yes:" are relevant while others are irrelevant. In this case, TrainsObeySignals and SignalPolicy are identified as irrelevant.

stance violates. When we enter an instance (using a GUI instance editor built into Alloy Analyzer) corresponding to a train collision, we see that it satisfies all constraints except for `TrainPhysics`. We then realize that we have required each train to move at each time step, but did not allow a train to remain in place. We have thus overconstrained the possible movement of trains, ruling out the collision scenario we had thought of and perhaps others we hadn't thought of.

We can fix the problem by modifying the `TrainPhysics` constraint to allow trains to remain in place:

```
fun TrainPhysics(s, s': State) {
   all t: Train |
       TrainStays(s, s', t) ||
       TrainMovesToNeighboringPath(s, s', t)
}

fun TrainStays(s, s': State, t: Train)
{ TrainLoc(s,t) = TrainLoc(s',t) }

fun TrainMovesToNeighboringPath(s, s': State, t: Train) {
   let r = t.(s.trainLoc), r' = t.(s'.trainLoc) | {
     let openPathsAtTrainEnd =
         (pathConns.(r.lastConn) & s.openPaths),
         newPath = openPathsAtTrainEnd - TrainPaths(s, t) | {
         some newPath &&
         r'.routePaths =
           SeqFunAdd(SeqFunRest(r.routePaths), newPath)
         r'.lastConn != r.lastConn
     }
   }
}

// The train location in the given state,
// or the empty set if this train is not on the track
// in the given state.
```

```
fun TrainLoc(s: State, t: Train): option Route {
{ result = t.(s.trainLoc) }
```

Now when we run the analysis, the tool does find a counterexample, While our `SignalPolicy` prevents new trains from entering an occupied *path*, it does not prevent new trains from entering an occupied *unit*. This can cause a collision when two paths in a unit intersect – for example, if the unit is a crossover unit. The scenario is illustrated in Figures 2-6 and 2-7. Note that this error would have been missed because of overconstraint; Alloy's overconstraint debugger has enabled us to detect and find the overconstraint, and eventually to find the error scenario formerly masked by overconstraint. Another example of using the overconstraint debugger is given in Section 6.2.

A corrected signalling policy can be expressed as follows:

```
fun SignalPolicy (s: State) {
   // if a unit is occupied by a train, it may not
   // be entered via any connector
   // (the actual property below specifies the contrapositive)
   all c: Connector, u: Unit |
      c in u.(s.mayEnter) => no u.unitPaths & OccupiedPaths(s)
}
```

With the corrected `SignalPolicy`, no examples of invariant violation are found in the scopes in which we have searched. However, analysis in higher scopes (with 3 units) reveals another problem: two trains may enter an empty unit on different paths, and end up in the same unit. This problem can be fixed by requiring that a unit may be entered through at most one connector. (Since only one train needs to enter an empty unit at any given time, this requirement should not impede normal train operations.) The revised Alloy definition of signal policy can be expressed as follows:

```
fun OccupiedUnits(s: State): set Unit
{ result = unitPaths.OccupiedPaths(s) }

fun SignalPolicy (s: State) {
   // if a unit is occupied by a train, the unit may not
   // be entered through any connector.
   // otherwise, there is only one connector through
   // which the unit may be entered.
   all u: Unit | let openConns = u.(s.mayEnter) |
      u in OccupiedUnits(s) => no openConns else sole openConns
}
```

With the revised definition, no counterexamples showing safety invariant violation are found even in higher scopes. Moreover, running the overconstraint debugger shows that all

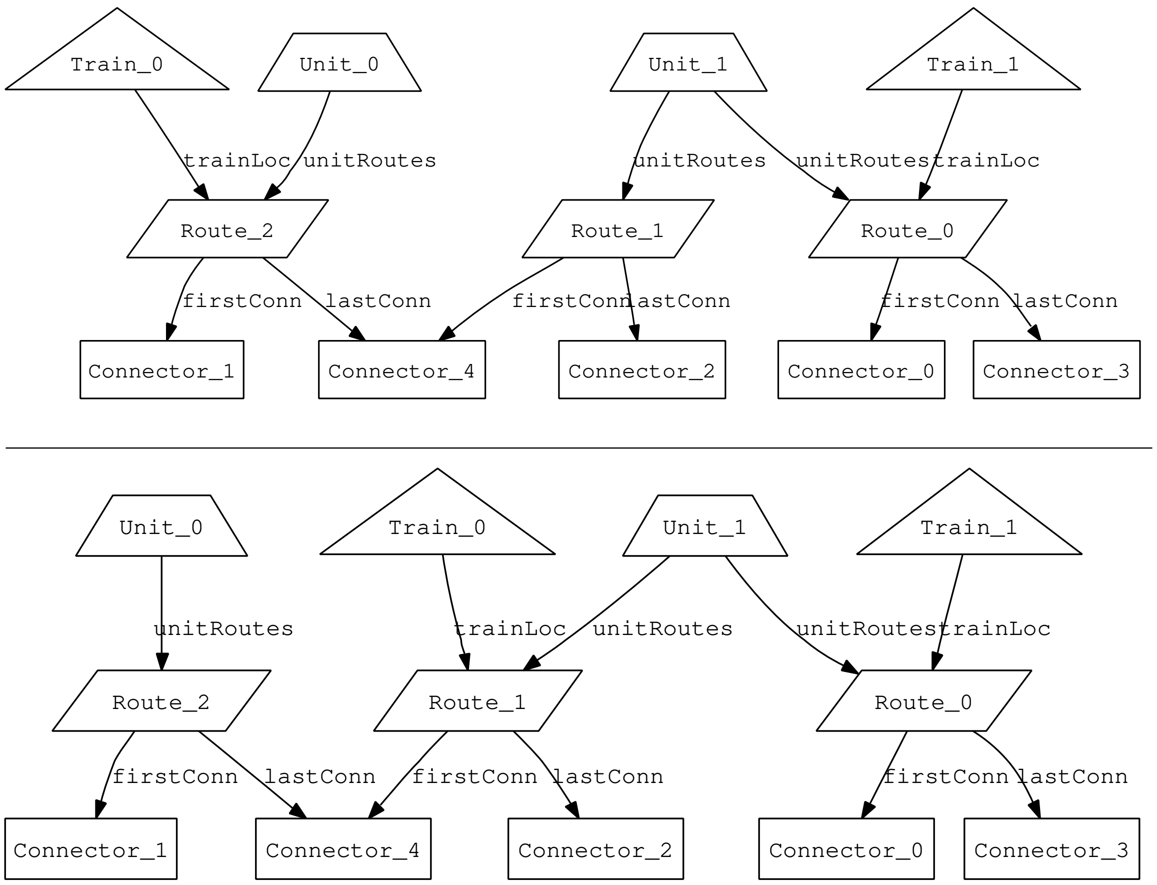
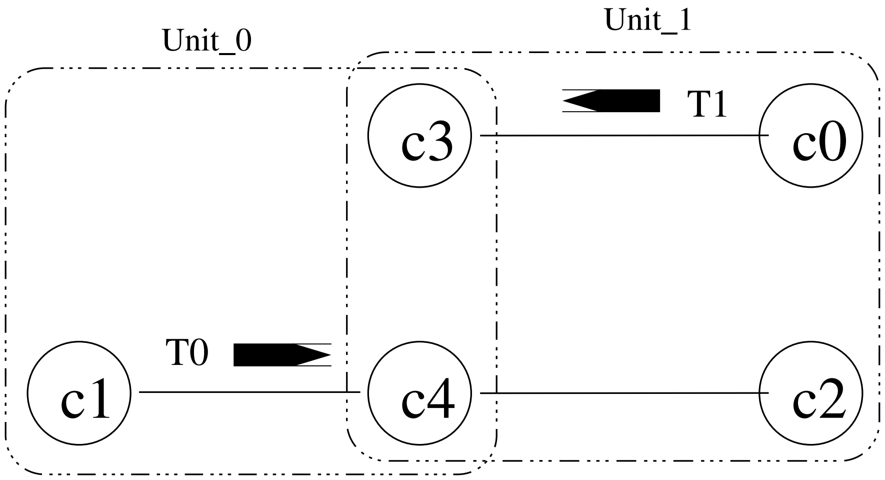


Figure 2-6: Safety violation: trains collide (relational view). The top figure shows the pre-state and the bottom figure the post-state. Train_0 moves from Unit_0 onto an unoccupied path in Unit_1, causing a conflict with Train_1 which resides on another path in Unit_1.

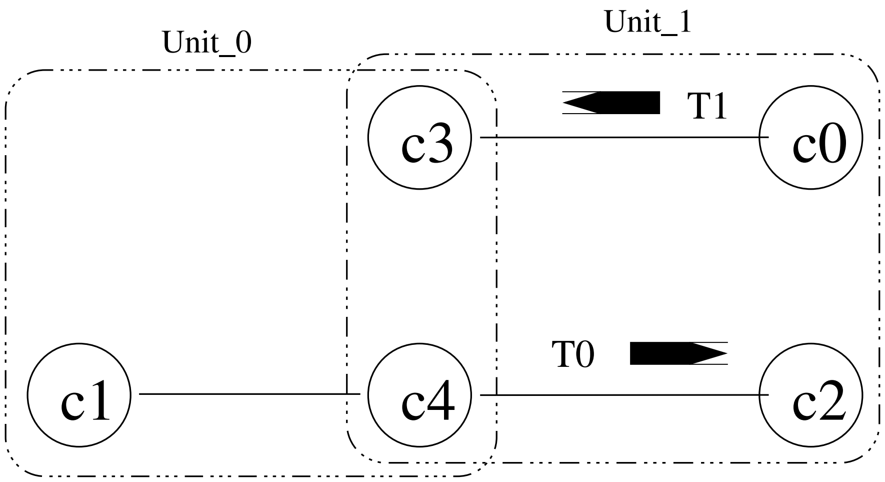
constraints are relevant to ruling out error traces. This gives us confidence that the model is correct, and that the absence of counterexamples reflects correctness of the modeled algorithm rather than the presence of an overconstraint.

2.6.2 Limitations of invariant preservation testing

While checking invariant preservation as shown above can uncover many errors, the technique has some serious limitations. To begin with, the technique does not compute the reachable states of the system, and is therefore prone to producing bogus counterexamples. For instance, supposed we want to ensure that our railway cannot get into a "stuck" state from which there are no valid transitions. We could test whether the system can pass from



State_0: Trains stand on adjacent units.



State_1: Train 0 moves into Unit 1, causing a conflict with Train 1 on another track segment in that unit.

Figure 2-7: Safety violation: trains collide (physical view). The top figure shows the pre-state and the bottom figure the post-state. Train 0 moves from Unit 0 onto an unoccupied path in Unit 1, causing a conflict with Train 1 which resides on another path in Unit 1.

an unstuck state into a stuck state, using the invariant preservation test. But there would be no guarantee that the unstuck state (from which the system gets into the stuck state) is itself reachable from an initial state of the system.

In some cases, the reachable states of a system can be easily described by a predicate. This was the case, for instance, in Alloy analysis of the Intentional Naming System (INS) [51]. In that work, the authors were interested in the effect of an update operation on a database. The original INS specification [1] included operations for building the database from scratch. However, it was determined (by manual analysis) that the databases that can be built using these operations can be described by simple constraints. It was therefore unnecessary to model the database construction operations; the database update operation could be checked on the valid databases by specifying Alloy constraints describing the valid databases.

However, in many transition systems, the reachable state space is not so easily described. The simplest example might be the chessboard. While it's easy to write predicates describing the initial chess position and the transition relation (whether two chess positions are related by a valid chess move), describing the *reachable* chess configurations with a predicate appears very hard.

Another drawback of invariant preservation analysis is that it cannot be used to check liveness properties. Unlike a safety property (which typically states that the system never reaches a bad state), a liveness property states that a system eventually does reach a "good" state.

The limitations of invariant-preservation testing can be overcome by encoding Bounded Model Checking (BMC) [9] problems in Alloy. That is the subject of the following section.

2.7 Bounded Model Checking

In this section, we show how Bounded Model Checking (BMC) [9]problems can be expressed and analyzed in Alloy. In Bounded Model Checking, the model represents a k-step bounded trace of an algorithm, and we can search for traces satisfying specified conditions – typically, for traces illustrating some error. Our contribution here lies in showing that BMC problems can be expressed in Alloy within the pure-logic model checking paradigm, without adding special language support for BMC. Note that expressing BMC problems in Alloy lets us use all other Alloy features – such as the ability to model complex data structures and the flexibility of pure-logic model checking – on BMC problems.

2.7.1 The elements of Bounded Model Checking

Bounded Model Checking (BMC) [9] is a technique for model-checking finite state machines using SAT solvers. Consider an FSM with a k-bit state vector. With $k \times m$ bits, we can represent a *bounded trace* of the FSM: a sequence of m states such that the first state is a valid initial state, and any two adjacent states are related by the FSM's transition relation. Formally, let S_i denote the i'th k-bit vector (representing the state of the FSM after i steps). Let the state machine be specified by two predicates: $I(S)$, specifying the initial state(s), and $T(S, S')$, specifying the transition relation. ($I(S)$ is a predicate on k Boolean variables, and $T(S, S')$ is a predicate on $2k$ Boolean variables). Then the following formula constrains the Boolean variable vectors $S_0, \ldots, S_{m-1}$ to represent a valid m-step initial trace of the FSM:

$$I(S_0) \wedge T(S_0, S_1) \wedge T(S_1, S_2) \wedge \ldots \wedge T(S_{m-2}, S_{m-1}) (*)$$

Note that in any Boolean assignment satisfying the formula (*), the Boolean vectors S_i represent only reachable states of the FSM; specifically, the states reachable from an initial state in up to m transitions. The formula can be generated automatically from the definitions of $I(S)$ and $T(S, S')$, by unrolling the transition relation m times.

Once the Boolean formula describing the valid initial bounded traces has been constructed, we can conjoin to it additional conditions requiring the trace to illustrate undesirable FSM behaviors. The resulting Boolean formula will be satisfiable iff the FSM fails to satisfy correctness properties, and a satisfying assignment will correspond to a bounded trace illustrating the problem.

The original BMC paper [9] gives an algorithm for converting correctness properties expressed in a temporal logic, Linear Time Logic (LTL), into Boolean predicates on the Boolean variables representing the bounded trace. For example, violations of the typical safety property "in all states, the invariant $P(S)$ holds" can be detected with the following predicate on the bounded trace representation:

$$!P(S_0)\vee !P(S_1) \vee \ldots \vee !P(S_{m-1})$$

Violation of the typical liveness property "eventually, the condition $Q(S)$ becomes true" can be detected by constraining the bounded trace never to satisfy the condition $Q(S)$ and to end in a loop:

$$!Q(S_0)\wedge !Q(S_1) \wedge \ldots \wedge !Q(S_{m-1}) \wedge ((S_0 = S_{m-1}) \vee (S_1 = S_{m-1}) \vee \ldots \vee (S_{m-2} = S_{m-1}))$$

Clearly, if such a trace is found, then an infinite trace exists in which $Q(S)$ never becomes true; but the converse also holds. Since the number of states of the FSM is finite, any infinite trace must end with a repeating loop; therefore, to find violations of liveness properties it is sufficient to test for the existence of a finite looping trace in which $Q(S)$ never becomes true.

LTL supports description of more complex safety and liveness properties (e.g. "$Q(S)$ becomes true some time after $P(S)$ becomes true). Conversion from full LTL to Boolean predicates on bounded traces was described by Biere et al [9].

If a BMC analysis fails to find a counterexample for a particular trace length (the value of m), it is still possible that a counterexample exists for a larger value of m. However, since the FSM has a finite number of states, there exists a sufficiently large number M such that the absence of counterexamples for a $m = M$ guarantees the absence of counterexamples for all m. If we can determine M, and if we can do the BMC analysis with the required trace length, then we can verify the absence of counterexamples for all possible traces. Unfortunately, in many cases M is too large, and verifying correctness for arbitrary-length traces is not practical. Despite this limitation, BMC can still be very useful. First, may errors can be illustrated with short counterexamples [4]. Second, trace length is typically only one of several limits on a model-checking analysis. Others bounds may include the number of processes in a distributed algorithm, the maximum length of communication queues, and so on. So even if no counterexamples exist for any trace length, counterexamples may yet be found for larger scopes of other parameters of a parameterized algorithm. Nevertheless, the focus of model checking is on finding bugs rather than proving correctness, and for that purposes a bounded search often suffices.

2.8 Encoding BMC problems in Alloy

The Alloy language and analyzer have no explicit support for either Finite State Machines or Bounded Model Checking. However, as we have seen earlier, it is possible to model the state of a transition system, and to construct Alloy models whose instances include several copies of system state. These techniques can be used to model BMC problems in Alloy. We can construct Alloy models whose instances represent the valid bounded traces of a transition system, and use Alloy constraints to require the traces to show counterexamples to correctness properties.

It's important to note that not only is BMC support not built into Alloy, but the ability to encode BMC problems in Alloy was discovered after the Alloy Alpha language [42] had been finalized. Moreover, no BMC-specific changes or language features were needed when the current Alloy language was defined. The ability to integrate a new modeling paradigm without changing the language is due to Alloy's pure-logic nature. If other useful modeling patterns are developed in the future, it's likely that they can be integrated into Alloy with similar ease.

2.8.1 General form of BMC constraints in Alloy

The general form of Alloy constraints for representing a BMC instance can be described as follows. First, we constrain the Alloy instance to represent a valid bounded trace:

```
fun Initial(s: State) { ... }
fun ValidTransition(s, s': State) { ... }

fact WellFormedTrace {
   Initial(OrdFirst(State))
   all s: State - OrdLast(State) |
     let s' = OrdNext(s) | ValidTransition(s, s')
}
```

The function `Initial` tests whether `s` (i.e. the one `State` atom contained in the singleton set `s`) represents a valid initial system state. The function `ValidTransition` tests whether the given pair of states represents a possible system transition. In the railway model, `Initial` might test that the station is empty (no trains have entered the modeled tracks). `ValidTransition` might test that trains obey laws of physics and red lights. Note how the objectification of state and the use of the total ordering module help express BMC constraints in Alloy. Another factor that facilitates expression of BMC constraints, inherited from Alloy Alpha, is the presence of quantifiers in the language.

In addition to trace well-formedness constraints, we write constraints requiring the bounded trace to show violations of correctness properties. The general form of these constraints is:

```
fun SafetyPropertyViolation() {  some s: State | !Safe(s)  }
fun LivenessPropertyViolation() {
   (some s: State - OrdLast(State) |
      StatesAreEquivalent(s,OrdLast(State))) &&
   all s: State | !LivenessProperty(s)
}
```

The function `SafetyPropetyViolation` requires the instance to show the violation of a simple safety property: that some invariant (`Safe`) holds in all reachable states. The function requires that some atom of `State` represent an unsafe state. All `State` atoms were restricted to representing reachable states by the fact `WellFormedTrace` defined earlier.

The function `LivenessPropertyViolation` requires the instance to show the violation of a simple liveness property: that the predicate `LivenessProperty` eventually becomes true. The function requires that the trace consist of states in which `LivenessProperty` is false, and end in a loop. The latter requirement ensures that `LivenessProperty` will not become true later in the trace; the bounded trace exhibited as counterexample corresponds to an infinite looping trace in which `LivenessProperty` never becomes true. The predicate `StatesAreEquivalent`, omitted here, tests whether two `State` atoms represent equivalent states, and is used to constrain the trace represented by the Alloy instance to end with a loop [5].

[5] We cannot simply test for equality of State atoms, because we are using State atoms to represent positions in the trace as well as instances of state.

2.8.2 Extending the railway example

To illustrate Bounded Model Checking on the railway example, we will first make the example somewhat more sophisticated. Since we'd like to specify and check a liveness property, we need to be able to specify some positive goal for the system to reach (as opposed to modeling aimless motion of trains). Therefore, to the model we will add a routing plan, which specifies a planned route for each train. The liveness property can then state that each train completes its plan. Note that, like track topology, the routing plan will be a global variable of the model, to be solved for by the analyzer; it will not be hard-coded by the user.

The transition relation will be expanded to take account of the routing plan. Besides following the laws of physics and the road signals, trains will now have to move according to the plan. At each time point, each train will attempt to advance along its plan. If several trains want to enter the same unit in order to advance along their respective plans, one train is chosen to move into the unit, and the others must wait. The one train that is given permission to enter the unit will do so if the unit is unoccupied, or if we know that the train currently occupying the unit will move away under our instructions. In other words, the planned movement of a given train depends not just on current locations of the trains, but on our movement instructions to other trains.

We will model the reachable states of the train system by starting with a simple, clearly reachable state: empty tracks. Each train's plan will call for the train to enter the tracks from a "hanging connector" (one attached to only a single unit, as opposed to one joining two units), and eventually leave the tracks through another hanging connector. We will also simplify matters by requiring each train to occupy only one unit at a time. Thus, it should always be possible to execute each train's plan in turn, taking one train at a time through the tracks. When more than one train enters the station at the same time, however, error scenarios become possible.

In sum, we will expand our railway model in the following ways: 1) the transition relation will now allow trains to enter and leave the tracks via hanging connectors; 2) there will be a routing plan which specifies a route for each train to follow; 3) the transition relation will require each train to follow its plan, and will constrain unit states and train signals to be consistent with the planned train movements. Afterwards, we will use the BMC modeling pattern described earlier to model bounded traces of the railway system, and to constrain these traces to illustrate violations of safety and liveness properties.

First, let's write some auxiliary Alloy definitions, which will make subsequent Alloy text more readable and concise:

```
// The set of paths occupied by the given train
// in the given state.
fun TrainPaths(s: State, t: Train): set Path
{ result = SeqFunElems((t.(s.trainLoc)).routePaths) }

// The connector at the back of the train,
// in the given state.
fun TrainBack(s: State, t: Train): option Connector
{ result = (t.(s.trainLoc)).firstConn }

// The connector at the front of the train,
// in the given state.
fun TrainFront(s: State, t: Train): option Connector
{ result = (t.(s.trainLoc)).lastConn }

// The units (at most two) that come together
// at the given connector
fun ConnUnits(c: Connector): set Unit
{ result = (unitConnsA+unitConnsB).c }

// Does the connector belong to only one unit (as opposed to
// joining together two units)?
fun IsHangingConnector(c: Connector) { sole ConnUnits(c) }
```

Next, let's expand our transition relation to allow for entrance and exit of trains:

```
fun TrainAppears(s, s': State, t: Train) {
   no TrainLoc(s,t)
   one TrainPaths(s', t)
   IsHangingConnector(TrainBack(s', t))
}

fun TrainDisappears(s, s': State, t: Train) {
   one TrainPaths(s, t)
   IsHangingConnector(TrainFront(s, t))
   no TrainLoc(s,t)
}

fun TrainPhysics(s, s': State) {
   all t: Train |
       TrainAppears(s, s', t) ||
       TrainDisappears(s, s', t) ||
       TrainStays(s, s', t) ||
       TrainMovesToNeighboringPath(s, s', t)
}
```

Next, let's model the routing plan. To model the planned route of each train, we will reuse the same `Route` signature used earlier to model train locations.

```
static sig RoutingPlan {
   // for each train, the route to follow through the tracks
   trainPlan: Train ->! Route
}

// The sequence of paths in the given train's plan.
fun TrainPlan(t: Train): SeqIdx ->? Path
{ result = t.(RoutingPlan.trainPlan).routePaths }

fact WellFormedTrainPlans {
   all t: Train | let r = t.(RoutingPlan.trainPlan) | {
     // every train's plan starts and ends at track edges
     IsHangingConnector(r.firstConn)
     IsHangingConnector(r.lastConn)
   }
}
```

Note that, like track topology and unlike train locations, the routing plan is not part of the changing system state; there is only one copy of the routing plan for the entire instance, while there are |`State`| copies of system state. The declaration `static sig RoutingPlan` forces the scope of `RoutingPlan` to be 1. Thus, `trainPlan` is a degenerate ternary relation: the scope of its leftmost column is equal to 1, so it is isomorphic to a binary relation of type `Train -> Route`. (That binary relation can be referenced as `RoutingPlan.trainPlan`. Thus, we can objectify the routing plans and yet treat it as an unobjectified, global relation (and pay no analysis cost) as long as we keep the scope of `RoutingPlan` at 1. The current definition of Alloy does not permit declaring a free-standing global binary relation – every relation must be part of some signature; but as the `trainPaln` example shows, we can easily achieve the effect of declaring a global relation by wrapping a relation in a `static` signature. Moreover, if in the future we need the model to include several routing plans, it would be easy to adapt the model: we would simply need to remove `static` modifier from the declaration of `RoutingPlan`.

Next, we need to extend the system state to include the position of each train along its plan, and some auxiliary information used in planning the next move [6].

```
sig PlanState extends State {
   trainPlanPos: Train ->? SeqIdx,
   trainWish: Train -> Unit,
   trainMayMove: set Train,
   unitEmptiedTo: Unit -> Unit
}
```

[6]We could also directly modify the State signature, but using the extension mechanism makes the model clearer and more modular by separating the train-plan aspect of the model into a separate paragraph

The relation trainPlanPos represents the position of each train along its plan; if a train t has not yet entered the tracks in state s, then t.(s.trainPlanPos) is empty. The other relations are used in planning train movement. The relation trainWish represents, for each train, the unit that train needs to enter to progress along its plan. The relation trainMainMove specifies the trains we allow to move; of all trains wishing to enter a given unit, we'll allow one train to move. unitEmptiedTo is used to keep track of occupied units that may be entered immediately because their occupants will move; unitEmptiedTo contains $\langle s_i, u_j, u_k \rangle$+ iff in state s_i, the train in unit u_j will move to unit $\langle u_k \rangle$.

We will omit the detailed Alloy constraints on the above relations; the full commented model appears in Appendix A. The constraints ensure that at each state, each train moves along its plan if possible or remains in place if not. The constraints also ensure that unit states (openPaths) and signals (mayEnter) allow the planned train movements.

Here we will only look at one aspect of these constraints: specifying which trains get to move.

```
fun TrainsFollowPlans(s, s': PlanState) {
   ...
   // of all the trains wishing to enter a unit, one may move
   all u: Unit | {
     // if at least one train wants to enter unit u...
     some (s.trainWish).u =>
       // ...then exactly one train gets permission.
       one (s.trainWish).u & s.trainMayMove
   }
   ...
}
```

Note that we do not give a specific strategy for deciding which trains may move; we simply give the weakest possible restriction on the value of the trainMayMove relation. This illustrates several advantages of declarative modeling. The user is freed from writing a more detailed specification; the model is less burdened with unimportant details and therefore more readable; the smaller model may be more tractable for the analyzer. Moreover, since we do not specify a specific strategy for moving trains, the analyzer checks *all* possible train movement strategies in one run. The Alloy model describes an abstraction of several concrete strategies; a property that holds for the abstraction will hold for any concretization of the abstraction. This situation is quite common in modeling; for example, in modeling a cache, we may omit the cache replacement strategy and simply say that some unspecified subset of cache lines gets dropped.

2.8.3 Encoding BMC analyses in Alloy: the railway example

We're now done with expanding the railway model, and can proceed to illustrating the use of Bounded Model Checking in the context of Alloy. First, we need to constrain the instance to model valid bounded traces of the railway system, starting from a valid initial state and following the transition relation:

```
fun Initial(s: State) {
   // there are no trains on the tracks
   no s.trainLoc
   // no train has yet started going through its plan
   no s.trainPlanPos
   // no train has yet completed its plan
   no s.trainDone
}

fun ValidTrace() {
   Initial(OrdFirst(State))
   all s: State - OrdLast(State) | let s' = OrdNext(s') | {
      TrainPhysics(s, s')
      TrainsObeySignals(s, s')
      TrainsFollowPlans(s, s')
   }
}
```

Note how declarative modeling leads to a clean separation between various aspects of the train motion; the constraints arising from physics, signals and planning are simply conjoined together to form the transition relation.

Now we can write the constraints forcing the bounded trace to illustrate violations of correctness properties. Let's start with a safety property. One type of train collision not captured by predicates on a single state, is when two trains from neighboring units exchange locations: train from unit `u1` move to neighboring unit `u2`, while the train from `u2` moves to `u1`. (Such a scenario is impossible when trains can't enter occupied units, but we have relaxed that restriction by allowing a train to enter a unit if that unit is being vacated.) As the trains exchange places, they may collide. To search for such a scenario, we can write the following Alloy constraints:

```
// The set of units occupied by the train
// in the given state.
fun TrainUnits(s: State, t: Train): set Unit
{ result = unitPaths.TrainPaths(s, t) }
```

```
fun TrainsPassEachOther(s, s': State, t1, t2: Train) {
   // both trains were on the track in state s
   some TrainUnits(s, t1)
   some TrainUnits(s, t2)
   // in state s', the trains exchanged places
   TrainUnits(s', t1) = TrainUnits(s, t2)
   TrainUnits(s', t2) = TrainUnits(s, t1)
}

fun SafeTrace() {
   all s: State | {
      // reuse the SafeState predicate from invariant
      // preservation testing: at most one train per unit,
      // all trains stand on open paths
      SafeState(s)

      // between this state and the next,
      // no pair of trains pass each other.
      let s' = OrdNext(s) | some s' => {
         all t1, t2: Train | t1 != t2 =>
          !TrainsPassEachOther(s, s', t1, t2)
      }
   }
}

fun SafetyViolation () {
   ValidTrace()
   !SafeTrace()
}

run SafetyViolation for 2 Unit, 3 Connector, 2 Path,
    6 Route, 2 SeqIdx, 2 Train, 3 State
```

Analysis of `SafetyViolation` reveals a simple counterexample, illustrated in Figures 2-8 and 2-9.

The track consists of two simple linear units. In the initial state, the track is empty. In the next state, trains `Train_0` and `Train_1` enter simultaneously from two sides. The routing plan calls for `Train_0` to move to unit `Unit_1`, and for `Train_1` to move to `Unit_0`. The train control logic decides that although `Unit_1` is occupied, it may be entered because its train will move to `Unit_0`; at the same time, `Unit_0` may be entered because its train will move to `Unit_1`. In other words, the value of `OrdNext(OrdFirst(State)).unitEmptiedTo` is `{<Unit_0,Unit_1>, <Unit_1,Unit_0>}`. The train control logic therefore allows the trains to exchange places, causing a safety violation.

One solution is to require that `s.unitEmptiedTo` not have cycles, for all `s`. This can be expressed with the following Alloy constraint:

```
fact NoTrainLoops {
   all s: State | no ^(s.unitEmptiedTo) & iden[Unit]
}
```

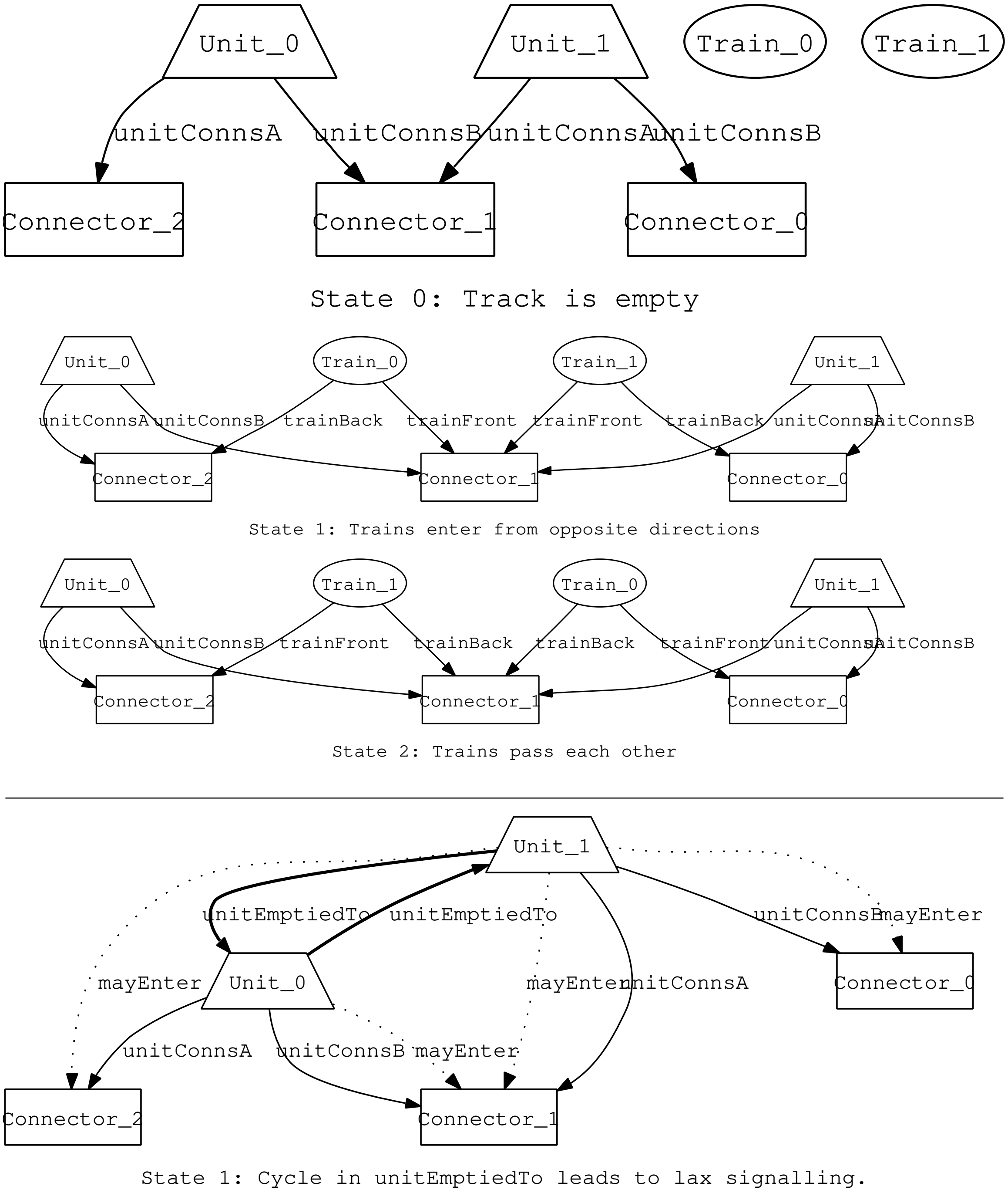
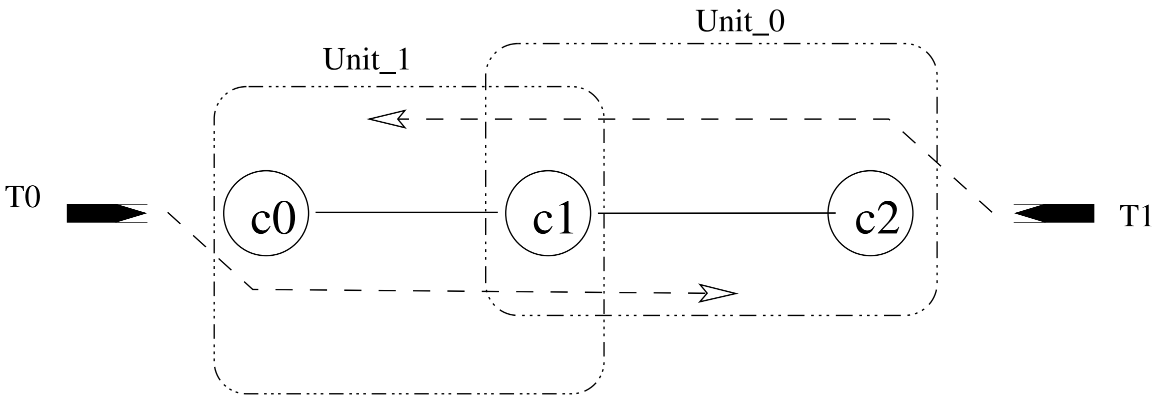
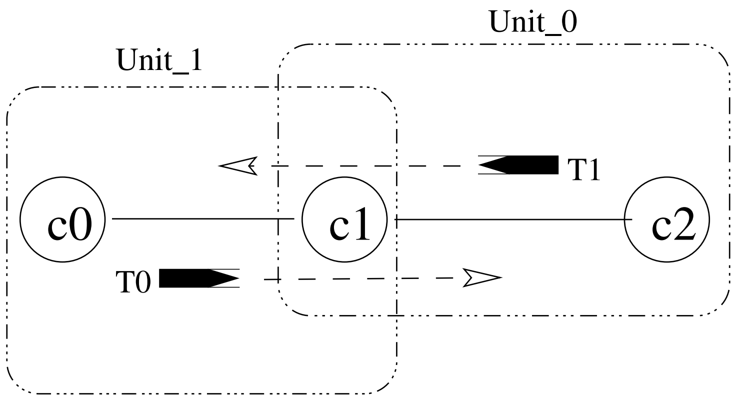


Figure 2-8: BMC analysis: safety property violation (relational view).

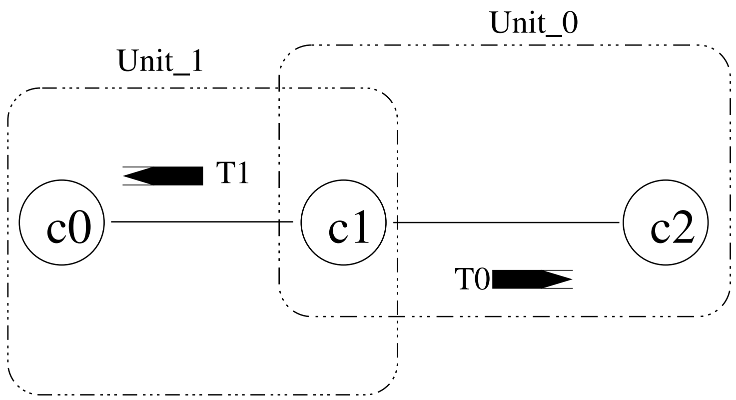
`^(s.unitEmptiedTo)` denotes the transitive closure of `s.unitEmptiedTo`, and `iden[Unit]` is a built-in relational constant that denotes the identity relation of type `Unit -> Unit`. The constraint says that the transitive closure of the graph has no self-loops – that is, that the graph is acyclic. With this additional constraint, the safety coun-



State_0: empty track, dashed lines show train plans.



State_1: Trains enter tracks, dashed lines show train plans.



State_2: Trains pass each other (unsafe).

Figure 2-9: BMC analysis: safety property violation (physical view).

terexample is eliminated. Note how the use of the transitive closure operator, inherited from Alloy Alpha, enables us to express the needed constraint concisely and naturally. Other model checkers [11, 33] do not support high-level graph operators such as transitive closure.

Now let us consider a liveness property: all trains eventually fulfill their plans. A counterexample would be a trace which ends in a loop before all plans are fulfilled. The following Alloy constraints will let us search for such a trace:

```
fun StatesAreEquiv(s1, s2: State) {
   s1.trainLoc = s2.trainLoc
}

fun TraceEndsWithLoop() {
   some s: State - OrdLast(State) | {
        StatesAreEquiv(s, OrdLast(State))
   }
}

fun PlanNotFulfilled() {
   // at the last state, not all trains are done
   Train !in OrdLast(State).trainDone
}

fun TrainRoutesDisjoint() {
   // to rule out some trivial counterexamples,
   // require that the routes of all trains
   // be disjoint
   all t1, t2: Train | t1 != t2 =>
     no SeqFunElems(TrainPlan(t1)) &
        SeqFunElems(TrainPlan(t2))
}

fun FindLivenessViolation ( ) {
   TrainRoutesDisjoint()
   ValidTrace()
   TraceEndsWithLoop()
   PlanNotFulfilled()
}

run FindLivenessViolation for 2 Unit, 4 Connector,
   4 Path, 4 Route, 2 SeqIdx, 2 Train, 3 State
```

Analyzing these constraints yields a counterexample, illustrated in Figures 2-10 and 2-11. There are two trains. Their plans do not share any paths. However, their plans share units, and this causes a liveness violation. In `State_1`, neither train can advance, and `State_2` is therefore equivalent to `State_1`. Since the system enters a loop before all trains fulfill their plans, the train plans will never be fulfilled. Fixing the problem will require non-trivial modifications to the model which we won't attempt here; but we have illustrated the ability of our setup to detect liveness violations through BMC analysis.

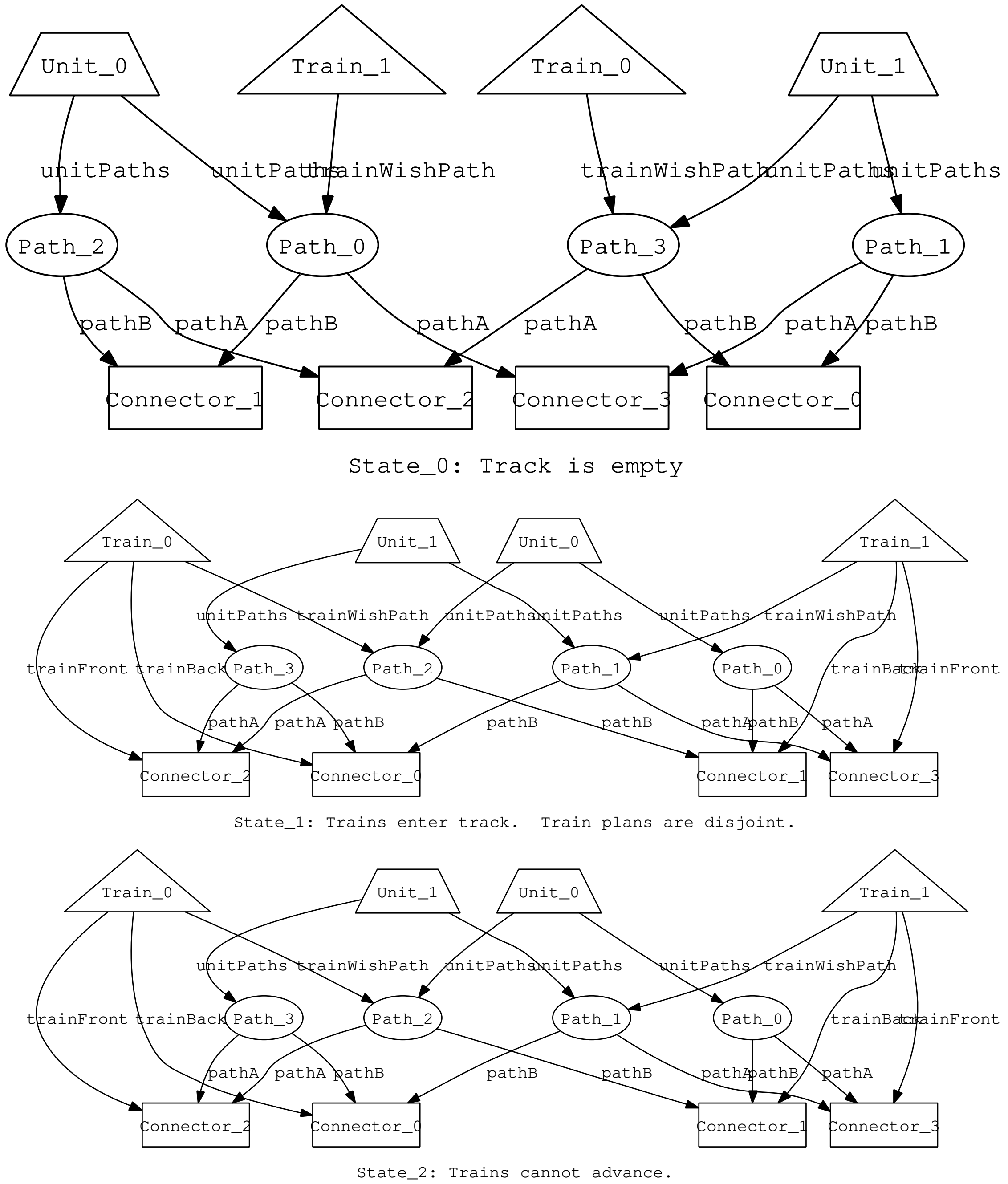
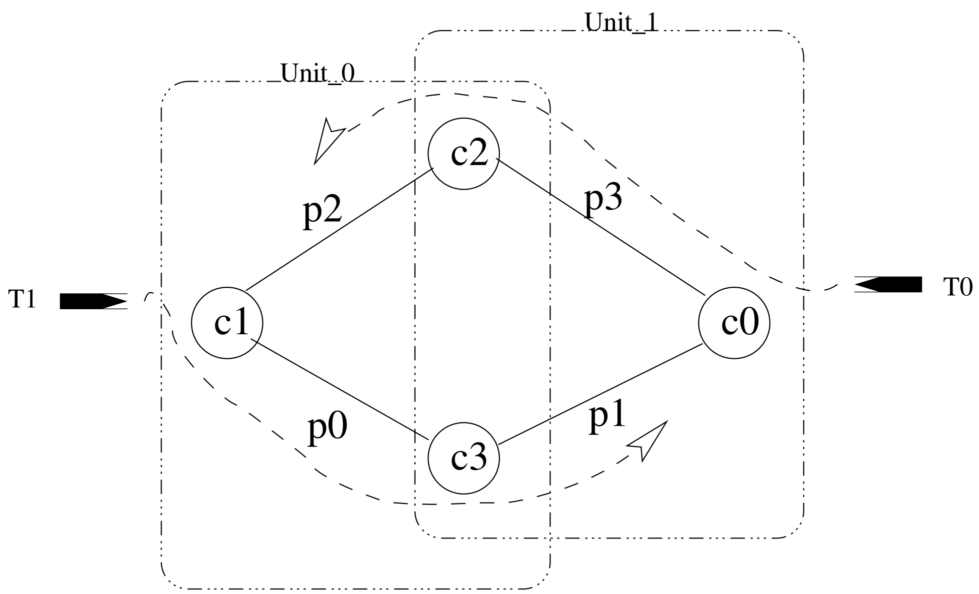


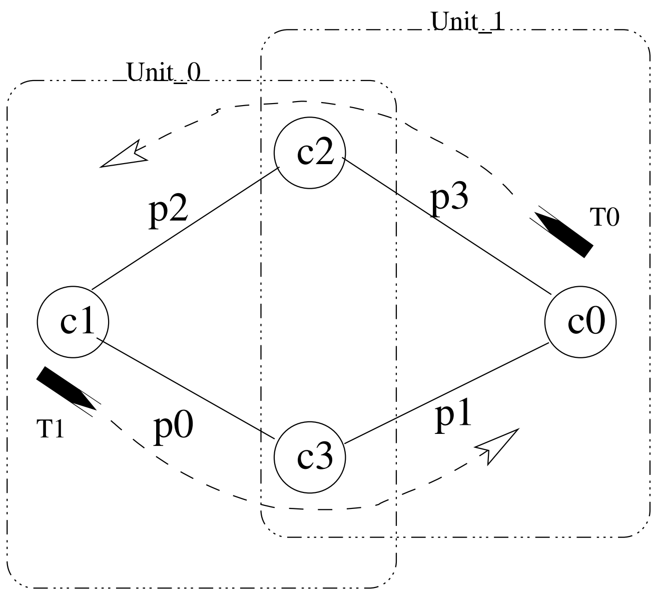
Figure 2-10: BMC analysis: liveness property violation (relational view).

2.9 Summary

In this chapter, we have illustrated pure-logic declarative modeling in Alloy on the example of a simple railway system. We have shown Alloy's ability to model algorithms that manipulate complex data structures, while keeping the model first-order. The pure-logic



State_0: empty track, dashed lines show train plans



State_1: each train entered the first step of its plan,but cannot proceed to the second.

Figure 2-11: BMC analysis: liveness property violation (physical view).

modeling paradigm supported a variety of modeling idioms in a uniform way. Declarative modeling allowed us to specify abstract models covering a variety of scenarios, such as the execution of an arbitrary train movement strategy on an arbitrary track topology under an arbitrary train plan. We have also shown how unsatisfiable core analysis can be used as part of the modeling process to find algorithmic errors masked by overconstraint.

Some of the modeling choices in our running example may seem arbitrary, and the question may arise – among the different ways of modeling a given system, how does the user choose the best one? For example, how does one make modeling choices that lead to the most tractable model? Unfortunately, given the unpredictability of the SAT solvers used for analysis, there is no simple answer. Part of the future work will be to provide the user with information that can be used to make models more tractable; for instance, a profiler may tell the user which parts of the model contribute the most to the size of the Boolean formula generated for analysis. Some relatively systematic control of search space size is enabled by objectification, as suggested in Section 2.4.5. And the scalability improvements described in subsequent chapters may reduce the need for the user to consider scalability when designing models. However, at this point there is no general method for rewriting models in a more scalable way without reducing the number of scenarios considered in the analysis.

This chapter showed that with pure-logic modeling, many analyses can be expressed with reasonable ease even without special language support for these specific analyses. Still, it may be asked: wouldn't it be better to have special language support for at least some of the common modeling patterns, such as bounded model checking. After all, objectification is also a modeling pattern, and having special language support for it (signatures) has proven very useful. One answer is that it's possible to use Alloy as an intermediate language for more specialized languages or "veneers", which are analyzed by translation to Alloy but make certain tasks more convenient. Such veneers have been created for modeling virtual functions [59], annotating Java code [59, 80], and specifying the structure of test cases [50]. While Alloy can act as an intermediate language to which veneers are translated, it remains very usable as a modeling language in which users write models directly. Objectification is a basic building block on top of which a variety of modeling patterns (such as bounded model checking) can be implemented. It is therefore sufficient, in the base Alloy language, to provide direct support for objectification [36, 44] but not for higher-level modeling patterns. Whenever the need to write some common pattern by hand becomes a problem, an appropriate veneer can be created. Also, it's possible to define standard Alloy libraries for common tasks, that can then be reused for a variety of models. Such generic libraries have been created, for example, for modeling groups of processes communicating via messages.

This chapter presented the Alloy language and analyzer from a user's perspective. Subsequent chapters will explain the algorithms used by the analyzer.

Chapter 3

Translation to SAT

This chapter describes how the question of satisfiability of an Alloy predicate can be reduced to the question of satisfiability of a Boolean formula in Conjunctive Normal Form (CNF). The main pupose of this translation is to quickly take advantage of new advances in satisfiability testing. The use of SAT solvers for analysis is inherited by Alloy from Alloy Alpha [37, 38].

Note that translating to CNF is only one possible way of testing satisfiability of Alloy predicates. Alternative approaches include writing a dedicated constraint solver for Alloy predicates, or translating to a constraint language other than CNF. (Plans for future work include the use of Quantified Boolean Formula (QBF) solvers [84] and Pseudo-Boolean Solvers [2] as alternatives to CNF-based SAT solvers). The users of Alloy Analyzer don't need to know anything about SAT solvers; the semantics of the tool from a user's point of view makes no reference to Boolean formulas.

Rather than describing the translation to CNF from Alloy, we will describe a more general translation from languages matching an abstract constraint schema (ACS). ACS is a schema for a family of constraint languages; Alloy is one instantiation of that schema. We use an abstracted setting for two reasons. First, the abstraction factors out the essential elements of the translation framework, simplifying the explanation. Second, some results in this thesis (Chapters 5 and 6, and elements of Chapter 4) apply to the abstract schema rather than only to Alloy. The abstraction makes it easier to see how other languages that instantiate the abstract schema might benefit from these results. The handling of quantified formulas in Chapter 5, and the computation of unsatisfiable cores in Chapter 6, are described in terms of the abstract schema. We will, however, explain how Alloy instantiates the abstract schema, and use Alloy examples for illustration.

3.1 Abstract Constraint Schema (ACS)

In this section we describe the Abstract Constraint Schema, which is an abstraction for a family of constraint languages. A language instantiating this schema allows writing of predicates on a finite collection of variables v_i. The variables take values from a finite universe U. The values of U are grouped into a finite number of types t_i, not necessarily disjoint; each variable v_i has a specific type $type(v_i)$, and takes values from that type. A type-respecting assignment of particular values to all variables v_i is called an *instance*.

In Alloy, the variables are the relations, and U contains all relational values that can appear in the model. The types t_i include the relation types used in the Alloy predicate. This includes relation types of the relations, as well as the types of all quantified variables and of all relation-valued intermediate expressions. Note that U is not determined by the Alloy predicate alone; it depends on the settings of basic type scopes. For given scope values, each relation type has a finite set of values. For instance, if the scope of the basic type `A` is 2, the relation type `A->A` has $2^{2\times 2} = 16$ values, from `{}` to `{<a0,a0>,<a0,a1>,<a1,a0>,<a1,1>}`. U contains the union of relation type values over all relation types used in the Alloy predicate. In addition, U contains the Boolean values $true$ and $false$, and a type $t_b = \{true, false\}$. In Alloy, t_b cannot be the type of the variables (relations), but can be the type of intermediate subformulas of the predicate, and is the type of the root node of the AST.

A predicate is expressed as an Abstract Syntax Tree (AST). [1] The leaves of the tree include the variables v_i, quantified variables (declared in quantifier nodes which are explained below), and constants. An assignment of values to all variables v_i induces a particular value from U in each AST node. The assignment satisfies the predicate iff the value induced in

[1]In practice, the predicate is expressed as text that is parsed into an AST; here we'll assume the predicate is already in AST form.

the root node is $true$.

The inner nodes of the tree are of two kinds: function nodes and quantifier nodes. Function nodes compute a deterministic function of their children. Each function node computes one of a set of predefined functions f_i, mapping a finite list of values from U to a result in U. The constants at the leaves can be viewed as nullary functions. Each AST node has a particular type t_i which describes the values from U that the node can take.

In Alloy, the functions f_i include relational operators (., +, ^) and Boolean operators (&&, ||, !). Each node (inner or leaf) has either a relation type or the Boolean type. The leaves always have relation types, and the root always has the Boolean type.

A quantifier node has a single child. The quantifier node declares a quantified variable, and specifies the domain of the variable (a list of values from U) and a combiner function (one of the f_i). The quantifier node computes its value by computing the value of the child for each value of the quantified variable, then applying the combiner function to the list of resulting values. An example of a formula that uses quantifiers is

$$f_{2[w_1 \in \{u_1, u_2\}]}(f_4(w_1, u_3, f_3(w_1, v_1), v_2))$$

It computes the value of $f_4(w_1, u_3, f_3(w_1, v_1), v_2)$ for $w_1 = u_1$ and for $w_1 = u_2$, then applies the function f_2 to the two resulting values.

In Alloy, quantifier nodes include quantified formulas (all x: A | F(x) and some x: A | F(x)) and comprehensions ({ x: A | F(x) }). For quantified formulas, the child parameterized by the quantified variable computes a Boolean value, and the combiner function (conjunction for all, disjunction for some) combines the Booleans from instantiations of the child to produce a Boolean result. If Alloy allowed arbitrary relational constants, quantified formulas could be rewritten using Alloy's conjunction and disjunction operators: e.g. all x: A | F(x) would be equivalent to F({<a_0>}) && F({<a1>}) for |A|=2. For comprehensions, the child computes a Boolean value for each possible tuple of the result, and the combiner constructs a *relational* value containing those tuples for which the child evaluated to true. For instance, for |A|=2, { x: A | F(x) } constructs a unary relation (a set) from the two Boolean values F({<a_0>}) and F({<a_0>});

the result contains the tuple `<a_0>` iff `F({<a_0>})` is `true`, and the tuple `<a_1>` iff `F({<a_1>}` is `true`.

The first step of the translation is to compute a ground version of the formula by grounding out the quantifiers. Grounding out involves obtaining ground versions of the quantifier's body. For example, the ground version of the formula `all t: Tick | t.up in adj` for `|Tick|={t0,t1,t2}` is

`({<t0>}.up in adj) && ({<t1>}.up in adj) && ({<t2>}.up in adj)`.

Constants such as `{<t0>}` refer to specific relational values (in this case a singleton set containing the tuple `<v0>`. Such constants cannot be written by the user directly in Alloy specifications, but are useful for representing ground forms of Alloy formulas. Note that while the lifted form of the Alloy formula (before grounding-out) is independent of the basic type scopes, the ground form is specific to a particular scope.

The ground formula is then translated to a Boolean formula in Conjunctive Normal Form. I will give a formal description of the translation framework. The framework assumes an abstract synax tree in ground form, and does not rely on the specific semantics of tree nodes defined by Alloy. This formal description will be useful in Chapter 6 when we describe debugging of overconstrained models

3.1.1 Translation

Satisfiability of the formula can be tested by converting it to a Boolean formula in conjunctive normal form (CNF). To convert an AST to CNF, we allocate to each AST node $n \in Tree$ a sequence of Boolean variables $bv(n) \in BV^*$ representing the node's value. The sequences of Boolean variables allocated to two nodes are identical if these are leaf nodes with the same AST variable, otherwise the sequences are disjoint. We define functions $enc : U \rightarrow Bool^*$ and $dec : Bool^* \rightarrow U$ for encoding and decoding values in U as binary strings. An assignment of Boolean values to all the Boolean variables allocated for AST nodes thus corresponds to assigning a value from U to each AST node. An assignment of U values to AST nodes is consistent if the value at each non-leaf node equals the result of applying the node's node function to the sequence of U values assigned to the node's children[2]. We translate an AST to CNF by generating CNF clauses on the Boolean variables allocated to AST nodes, so that the conjunction of these clauses is true of a given assignment to Boolean variables iff the Boolean assignment corresponds to a consistent assignment of U values to AST nodes.

The translation is done separately for each AST node. For each node, we produce a set of CNF clauses relating the Boolean variables allocated to that node, to the Boolean variables allocated to the node's children. Intuitively, the clauses are true iff the U value represented by the nodes's Boolean variables equals the result of applying the node's node function to the sequence of U values represented by the Boolean variables allocated to the node's children. The clauses output from translating an AST node depend only on the node function which the node computes of its children, and on the Boolean variables allocated to the node and the children.

For each node function f_i, we define a corresponding "CNF translation" function

$$\hat{f}_i : BV^*, BV^{**} \rightarrow \mathbb{P}\, Clause$$

$\hat{f}_i$ takes a sequence of boolean variables from the domain BV, corresponding to the result of the function, and a sequence of sequences of boolean variables corresponding to the arguments, and returns a set of clauses that encode the function in CNF. The correctness of this function is justified with respect to the encoding function and the semantics of f_i itself; its result evaluates to true iff the Boolean variables allocated to the result of f_i encode the value computed by applying f_i to the argument values encoded by the Boolean variables allocated to the arguments.

[2]This translation loses high-level information. However, the resulting CNF format permits extremely efficient backtrack search; in particular, good algorithms for constraint propagation, learning, and determining decision order during backtrack search depend on the CNF format of the formula. SAT solving on the CNF format has been the subject of intense research [63, 28]. These considerations more than make up for the loss of high-level information during translation to CNF. However, conveying high-level information to a SAT solver can result in improved search times. The use of symmetry (Chapter 4) and subformula sharing (Chapter 5) are examples of this; future work will focus on other ways to use high-level information to improve SAT solving.

For example, consider the Alloy node `in`, which takes two children of the same relation type and tests whether the left child is a subset of the right child (with relations viewed as sets of tuples). The corresponding translation function would take as arguments Boolean variables representing the presence of each possible tuple in the left child and in the right child, as well as a Boolean variable representing the result of the `in` test. It would output CNF clauses in terms of these variables, true if the result of the `in` test correctly reflects whether the right child contains all the tuples in the left child. More concretely, suppose we need to translate the Alloy formula `p in q`, where `p` and `q` are unary relations of type `A`; suppose the scope of `A` is 2, i.e. `A={A_0,A_1}`. Suppose also that the Boolean variables `p0` and `p1` represent whether `p` contains the tuples `<A_0>` and `<A_1>` respectively; that the Boolean variables `q0` and `q1` represent whether `q` contains the tuples `<A_0>` and `<A_1>` respectively; and that the Boolean variable `r` represents the truth of the formula `p in q`. Then, the node translation function for the `in` node would generate CNF clauses true iff `r <=> ((p0 -> q0 && (p1 -> q1)))` (`<=>` denoting Boolean equivalence and `->` Boolean implication).

Using these individual translation functions, we can now translate the tree. The function $transl : T \to \mathbb{P}\, Clause$ translates one AST node to CNF, and is defined as

$$transl(t) \equiv \text{let } t = Tree(f, ch) \mid \hat{f}(bv(t), map(bv, ch))$$

The CNF translation of an entire AST is then just the union of translations of its nodes:

$$translTree(t) = \cup_{n \in nodes(t)} transl(n)$$

Correct translation to CNF requires that for each node t, for any Boolean assignment $ba : BV \to Bool$ satisfying $transl(t)$, we have

$$\begin{aligned} &f(map(dec, map(\lambda\, cv \,.\, map(ba, cv), map(bv, ch)))) \\ &\quad = dec(map(ba, bv(t))) \end{aligned}$$

where the node t computes the node function f of its children ch. map here denotes a meta-function that maps a given function (first argument) over a given list (second argument); that is, it denotes a new list each element of which is obtained by applying the given function to the corresponding element of the given list. To test satisfiability, we constrain the Boolean variable(s) allocated to the root to represent the value $true$ from U, by adding the appropriate unit clauses.

Chapter 4

Symmetry breaking

In this chapter we describe techniques for using symmetry considerations to improve model checking performance. Many systems have indistinguishable components, which lead to isomorphic execution scenarios. For instance, if a system has two indistinguishable processes, each scenario starting with "process 1 grabs a lock" is equivalent to another scenario starting with "process 2 grabs a lock". "Equivalent" means that for any property, either both scenarios satisfy it or both violate it. When searching for a scenario violating a property, it suffices to check one representative of each class of isomorphic scenarios. Since the number of isomorphism classes can be much smaller than the number of distinct scenarios, symmetry-based reductions can greatly increase efficiency of search.

In Alloy, analyses are reduced to satisfiability problems which are then solved by a pluggable SAT solver. That means that modifying the solver to take advantage of available symmetries isn't possible. Instead, we modify the SAT problem in a way that drives the solver to explore only a small subset of solutions in each isomorphism class. We generate additional constraints (symmetry-breaking predicates) that are true of only some representative instances in each isomorphism class; these lead a backtracking solver not to explore portions of the search space that contain no isomorphism class representatives. Constructing the additional constraints involves a fundamental tradeoff between pruning power and constraint size: we would like a constraint that selects few representatives but can be expressed compactly so that it doesn't overwhelm the solver. In this chapter, we discuss ways of constructing effective yet compact symmetry-breaking predicates. We propose measures of predicate quality, and give experimental results confirming effectiveness of our predicates.

4.1 Symmetries of Alloy models

Many systems exhibit symmetry, for example in the form of interchangeable processes. Symmetry partitions the space of executions into equivalence classes. For any given property, the executions in a single isomorphism class either all satisfy or all violate the property. It is therefore sufficient to consider only one representative execution per isomorphism class. This can lead to an exponential reduction in the size of the search trees explored during backtracking search, as entire subtrees (sections of search space) can be eliminated if they do not contain any of the chosen isomorphism class representatives.

Let us look at how symmetry reductions apply to Alloy. Since Alloy does not allow the user to refer to specific atoms, all atoms within each basic type are interchangeable. If one instance of the model is obtained from another by permuting the atoms within each basic type, the truth value of any Alloy predicate will be the same on the original and the permuted instance. More precisely, a symmetry of an Alloy model specifies a permutation of atoms within each basic type, and acts on basic type atoms according to the permutation. The action of the symmetry extends naturally to tuples (by acting on each atom within the tuple), to relational values or sets of tuples (by acting on each tuple in the set), and finally to instances or groups of relational values (by acting on each relational value in the instance). The action of a symmetry on an Alloy instance does not change the truth of any Alloy predicate on that instance.

Consider the railway example from Chapter 2. Figures 4.1 and 4.2 show two isomorphic instances. The symmetry that relates them exchanges `Unit_0` with `Unit_1`, `Train_0` with `Train_1`, and maps the atoms of `Route` as follows: `Route_0 -> Route_1`, `Route_1 -> Route_2`, `Route_2 -> Route_0`. This can be written more compactly as: `[Unit (1,0), Train (1,0), Route (1,2,0)]`. For any property expressible in Alloy, the two isomorphic instances either both satisfy the property or both violate it.

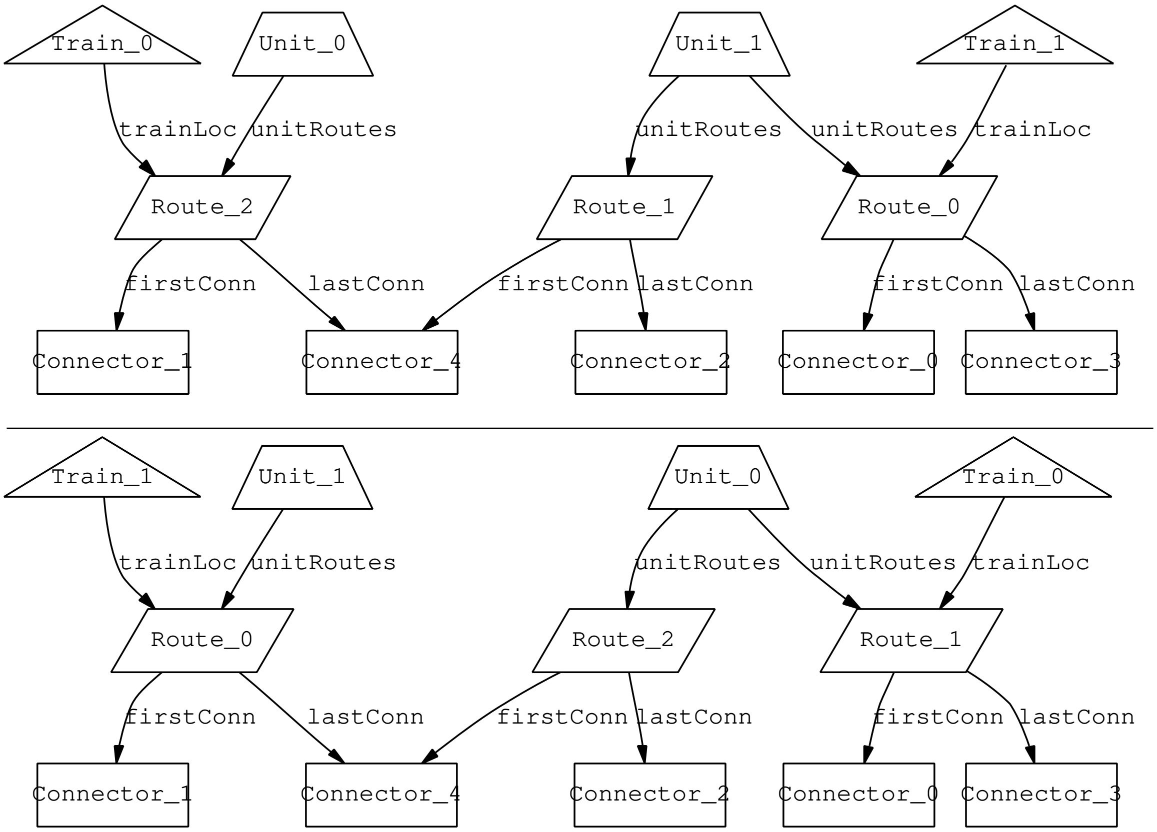


Figure 4-1: Isomorphic instances, related by the following symmetry: Unit (1,0), Train (1,0), Route (1,2,0). Relational view.

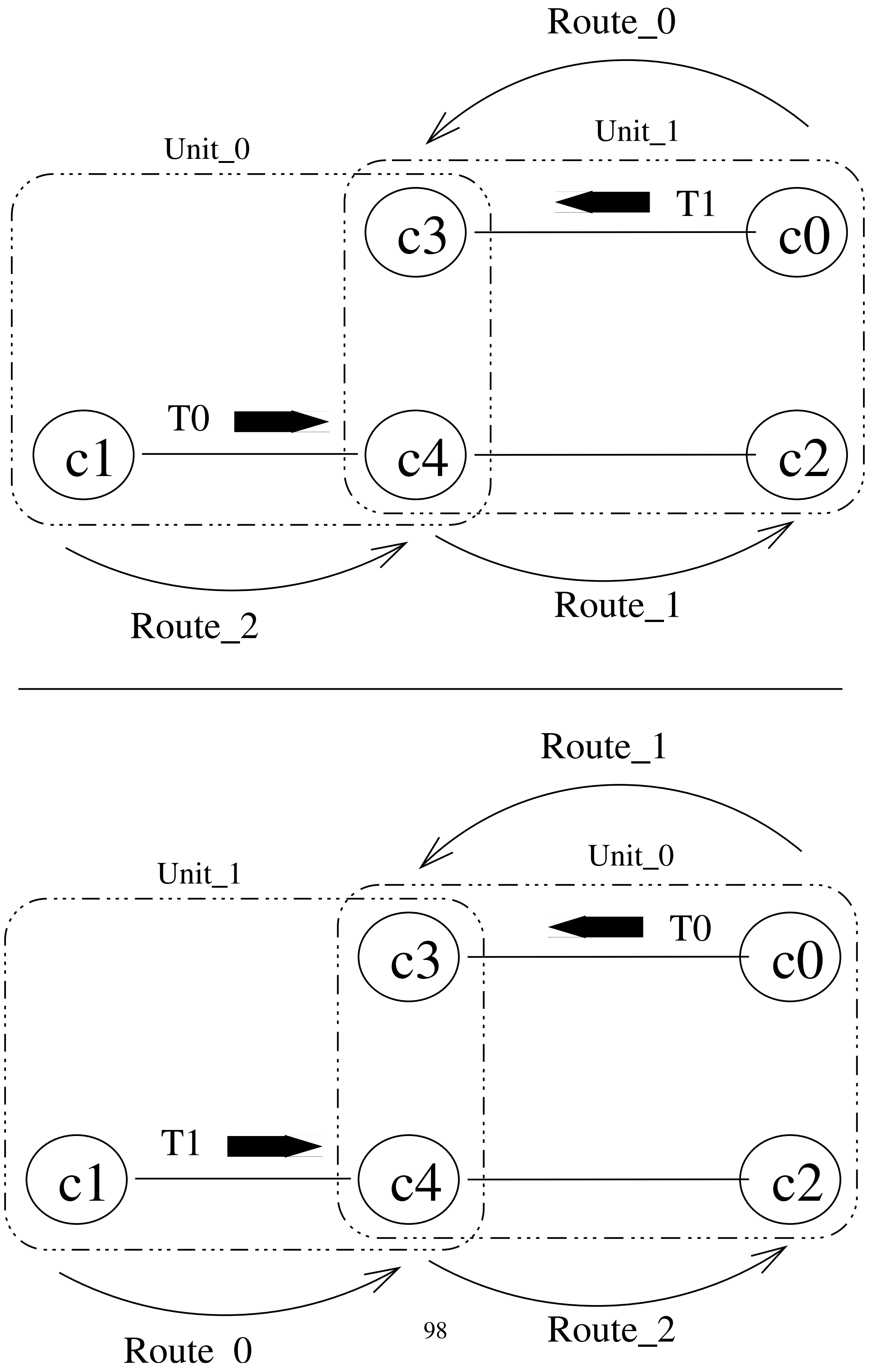


Figure 4-2: Isomorphic instances, related by the following symmetry: Unit (1,0), Train (1,0), Route (1,2,0). Physical view.

4.2 Symmetry-breaking predicates

Since we're using external SAT solvers to analyze Alloy models, we cannot modify the solver algorithms to take advantage of symmetry [1].

One way to use symmetry without changing the solver, not explored here, is to reformulate the Boolean problem in terms of a smaller number of variables: if there are 1000 instances but only 10 isomorphism classes, we only need 4 bits to represent all possible solutions [48]. For example, for a unary relation over a basic type of scope k, there are $k+1$ isomorphism classes; all can be represented with $\log k$ bits denoting the number of unary tuples in the relation. Two factors make this approach non-trivial. First, while in some cases the isomorphism classes are easily characterized and a natural encoding of them can be found, often this is not the case. Second, when the relations are used in higher-level expressions, compositional translation of the higher-level expressions to Boolean formulas will require us to construct Boolean formulas representing the presence or absence of individual tuples of the relations. During translation to CNF, these Boolean formulas will require the introduction of additional Boolean variables, negating any earlier savings in the number of Boolean variables.

Another way to use symmetry without changing the solver is via *symmetry-breaking predicates* [15]. To the formula that we would ordinarily give to the solver, we conjoin an additional constraint that is true of at least one instance (a "representative") in each isomorphism class. This restricts the search space while preserving satisfiability of the formula. During backtracking search, if all extensions of the current partial assignment are *not* the represenatives of their respective isomorphism classes

For unsatisfiable formulas – which typically take longest to analyze, since the entire search space must be considered – the addition of symmetry-breaking predicates clearly provides a benefit. For satisfiable formulas, addition of symmetry-breaking predicates – which results in the removal of perfectly good solutions – may seem like a bad idea: doesn't

[1]A precursor to Alloy, called Nitpick, did use elimination of symmetries to speed up search [40]. That tool did not have a pluggable backend and did not scale very well. When an early version of Alloy (now called Alloy Alpha) was built with a pluggable backend, problem symmetries were not used to speed up the search.

it reduce the chance of stumbling upon a solution early in the search? While this may be true, even for satisfiable formulas symmetry-breaking predicates can help by summarily excluding solutionless regions of searchspace during backtrack search. If a region contains no solutions and no isomorphism class representatives, it will be quickly excluded where without symmetry-breaking predicates it would have had to be searched.

Moreover, there are situations where we're interested not just in finding a satisfying assignment but in enumerating non-isomorphic satisfying assignments. For instance, one way to check sanity of an Alloy model is to simulate some instances – both to check that the allowed instances are well-formed and to make sure that the instances corresponding to specific scenarios are allowed. Simulation can be much more useful if each simulated instance is "essentially distinct" from the others – that is, not isomorphic to them. Another situation where isomorph elimination during solution enumeration helps is when Alloy is used for test case generation [50, 53]. Eliminating isomorphic test cases results in smaller test suites and reduces the testing time.

The difficulty with the symmetry-breaking predicate approach lies in the generation of a good symmetry-breaking predicate. To be effective, the predicate needs to allow small numbers of instances in each isomorphism class. But more precise predicates tend to be more complex, and a large predicate can slow down the SAT solver. Generating an *exact* predicate (one that allows exactly one instance per isomorphism class) is NP-complete [15]. However, in many important cases effective partial symmetry-breaking predicates can be generated. This thesis will describe how to construct symmetry-breaking predicates that are useful in practice, and how to measure their effectiveness.

Previous work [15] identified a generic scheme for constructing symmetry-breaking predicates. Recall that we encode an Alloy satisfiability problem as a Boolean satisfiability problem; each Alloy instance corresponds to a Boolean instance (i.e. an assignment to all the Boolean variables). For a fixed ordering of the Boolean variables, all Boolean instances are lexigographically ordered. A symmetry-breaking predicate can be constructed that is true of exactly the lex-leader Boolean instance in each isomorphism class. The predicate explicitly requires, for each symmetry, that applying this symmetry to a solution satisfying the predicate lead an equal or a lexigoraphically larger instance. The size of the predicate is linear in the number of problem symmetries. In many cases, the number of symmetries can be very large. For example, the number of symmetries of an Alloy model is the product of factorial of basic type scopes. While it's possible to construct a partial symmetry-breaking predicate by breaking only a subset of symmetries, it is unclear which symmetries should be selected.

4.3 Introduction to the symmetry-breaking problem

Consider a universe U of combinatorial objects representable by m-bit binary numbers. We will speak interchangeably of an object and its binary representation. Let U be divided into equivalence classes of isomorphic objects. A permutation θ of the m bits is a *symmetry* of the universe iff applying θ to any object $X \in U$ yields an object isomorphic to X. The set of all symmetries is the *symmetry group* of the universe U, denoted by Sym.

For example, n-node digraphs can be represented by $n \times n$ adjacency matrices, and two matrices A, B are isomorphic iff there exists a permutation θ of the n nodes such that $\theta(A) = B$, where $(\theta(A))_{i,j} = A_{\theta(i),\theta(j)}$. Note that θ is a permutation of the n *nodes* of the digraph, but it also acts on the n^2-bit *adjacency matrices*, because each permutation of the nodes induces a corresponding permutation of the adjacency matrix bits. The symmetry group Sym has order $n!$ and is isomorphic to σ_n, the symmetric group of order n.

Suppose you need to find an object X from a universe U, satisfying a property $P(X)$ (or determine that no such object exists). Suppose also that P is preserved under isomorphism, i.e. is constant on each isomorphism class. Enumerating all elements of U and testing P on each is clearly wasteful: it's enough to test P on one object per isomorphism class[2]. For some classes of objects, procedures exist for isomorph-free exhaustive generation [60, 41, 35]. Faster generation procedures may be developed at the cost of generating more than one labeled object per isomorphism class and/or repeating objects.

If no object in U satisfies P, the generate-and-test approach must explicitly generate

[2] Alloy does not use explicit enumeration, but the point applies to symbolic search as well: when examining regions of the search space, some regions can be removed as long as the remaining regions contain at leaset one object from each isomorphism class.

a complete representation of at least one representative per isomorphism class to verify unsatisfiability. On the other hand, backtracking methods [17] can rule out entire sets of objects without explicit generation, by determining that no object extending a *partial* binary representation satisfies P. If P can be encoded as a polynomial-size Boolean constraint on the bits of the fixed-length binary representation (as opposed to black-box computer code), backtracking methods for satisfiability can be used. Such methods can significantly outperform explicit generate-and-test approaches, as demonstrated by satisfiability encoding of planning problems [49].

Crawford et al [14] have proposed an approach to taking advantage of isomorphism structure in this framework. We define a *symmetry-breaking predicate* on U, $SB(X)$, which is $true$ on at least one *representative* object per isomorphism class. We then test for satisfiability of $P'(X) = P(X) \wedge SB(X)$. Since P is constant on each isomorphism class, P' is satisfiable iff P satisfiable. Moreover, P' is solved much faster than P by backtracking, because it is more constrained: the algorithm will backtrack if none of the extensions of its current partial instantiation are isomorphism class representatives selected by SB, whereas with the original predicate backtracking can only happen if all extensions of the current partial instantiation immediately violate P. Experiments show that symmetry-breaking predicates can reduce search time by orders of magnitude with no changes to the search algorithm [14, 47].

The difficulty of this approach lies in generating the symmetry-breaking predicate. In general, generating a *complete* symmetry-breaking predicate ($true$ of exactly one representative per isomorphism class) is NP-complete [14]; the practical choice is between *partial* symmetry-breaking predicates, $true$ of at least one (typically more than one) representative per isomorphism class. To be effective, the predicate must rule out a large fraction of objects from each isomorphism class. On the other hand, the predicate must be compact; otherwise, checking the predicate's constraints at each search node will slow down the search, erasing the benefit of expanding fewer search nodes. Balancing these contradictory requirements is the subject of this chapter.

The rest of the chapter is organized as follows. Section 4.4 summarizes prior approaches and points out their deficiencies. Section 4.5 describes the generation of symmetry-breaking predicates for several classes of combinatorial objects. Section 4.6 gives a uniform efficiency measure for symmetry-breaking predicates, and evaluates the predicates from Section 4.5 according to this measure. Section 4.8 gives some experimental evidence that the predicates described in this paper can improve search time. Section 4.9 describes directions for future work.

4.4 Prior work

In his original paper on symmetry-breaking predicates, Crawford proposes the following general framework for predicate generation. Fix an ordering of the bits in the object's binary representation. This induces a strict lexicographical ordering on all objects. Construct a symmetry-breaking predicate which is true on the *lexicographically smallest* object in each isomorphism class, as follows.

Let V be a fixed ordering of the bits of the binary representation. Then

$$\bigwedge_{\Theta \in Sym} V \leq \Theta(V)$$

is a symmetry-breaking predicate, true of only the lexicographically smallest object in each symmetry class. This predicate explicitly requires that *any* symmetry map either fix the the representative object, or map it to a lexicographically higher object – i.e. that the representative object be lexicographically smaller than any isomorphic object. (We assume that the symmetries of the problem are known; a recent symmetry-breaking framework that incorporates detection of available symmetries is described by Aloul et. al. [3].)

Unfortunately, in many important cases Sym is very large. For example, for n-node digraphs $|Sym| = n!$, because any permutation of the graph's nodes (and the corresponding permutation of adjacency matrix entries) leads to an isomorphic graph. Crawford suggests mitigating the problem by replacing Sym with a polynomial-size *subset* $Sym' \in Sym$, thus requiring that the object be lexicographically smallest with respect to only some of the symmetries.

Crawford gives no formal guidance on choosing the subset of symmetries to break or the fixed variable numbering to use. This paper begins to fill the gap by describing polynomial-size symmetry-breaking predicates for some common combinatorial objects. For some objects, we refine Crawford's algorithm by determining Sym' and V. For others, we present new predicate constructions, giving a concrete alternative to Crawford's lexicographic approach. Problem-specific symmetry-breaking predicates for some classes of combinatorial objects have previously been studied by Puget [68].

Crawford uses empirical measurements to gauge the effectiveness of his symmetry-breaking predicates. While such end-to-end tests are certainly useful, they give no hint of how much a given predicate can be further improved, and reflect peculiarities of a particular backtracking algorithm (such as the dynamic variable-ordering heuristic [17]) besides the inherent complexity reduction brought by the predicate. We present an alternative approach which directly measures predicate pruning power, and gives a quality measure relative to a complete symmetry-breaking predicate.

4.5 Generating symmetry-breaking predicates

In this section, we present methods for generating symmetry-breaking predicates on several classes of combinatorial objects: acyclic digraphs, permutations, direct products, and functions. These objects commonly occur in formal descriptions of system designs [43], the analysis of which motivates this work. Each subsection deals with one class of combinatorial objects, describing the binary representation, the isomorphism classes, and the construction of the symmetry-breaking predicate in terms of the binary representation.

4.5.1 Acyclic digraphs

Let U be the set of $n \times n$ adjacency matrices representing acyclic digraphs. Two matrices representing isomorphic digraphs are isomorphic. The symmetry group Sym has order $n!$.

Any acyclic digraph has an isomorphic counterpart that is topologically sorted with respect to a given node ordering. In terms of adjacency matrices, this means that every isomorphism class of adjacency matrices representing acyclic digraphs includes an upper-triangular matrix (since the lower triangle represents "backwards" edges from higher-numbered to lower-numbered nodes). Our symmetry-breaking predicate simply requires all entries below the diagonal to be $false$ [3]. This does not completely break all symmetries, but as measurements in section 4.6.1 show, breaks most.

Additionally, this symmetry-breaking predicate, together with the requirement that diagonal entries be $false$ (eliminating self-loops), implies the acyclicity constraint, so no additional constraints on the matrix are needed. By contrast, expressing the acyclicity constraint on general digraphs takes a constraint of size $\Omega(MatMult(n) \log n)$, where $MatMult(n)$ is the complexity of matrix multiplication. (The constraint involves computing the transitive closure of the adjacency matrix and asserting that it has no diagonal entries; the transitive closure is computed by using repeated squaring of the adjacency matrix.) Shorter constraints require less time to check at every search node, leading to faster search. In general, in cases where not all binary representations represent valid combinatorial objects from our universe U, constraints restricting the object to valid values are separate from the symmetry-breaking predicate. This example illustrates a new use of symmetry-breaking predicates: to obviate the need for some original problem constraints by removing the solutions these constraints were meant to remove.

Note that this symmetry-breaking predicate does not use Crawford's methodology. It's not even clear that a single fixed variable ordering exists which corresponds to this predicate. The next section on permutations gives another example of a symmetry-breaking predicate not based on lexicographic comparison.

4.5.2 Permutations

Let U be the set of $n \times n$ binary matrices representing permutations of n items. Matrix A represents the permutation mapping i to j iff $A_{i,j}$ is true. A matrix A represents a valid permutation (is a *permutation matrix*) iff every column and every row has exactly one $true$ bit.

[3] We could achieve the same effect by only allocating Boolean variables to above-the-diagonal tuples of the DAG relation, but for consistency we implement this the same way as other symmetry-breaking schemes – by conjoining additional constraints. Since SAT solvers immediately instantiate variables forced to a particular Boolean value by unit clauses, there is no differences between the two methods in terms of search time.

Two permutations are isomorphic if they have the same cycle structure, i.e. the same multiset of cycle lengths. Thus, an isomorphism class of permutation matrices corresponds to one permutation on a set of n indistinguishale objects. We define a canonical representative of each isomorphism class, and give a polynomial-size Boolean predicate on permutation matrices which is true only of the canonical representatives. We thus achieve full symmetry-breaking with a polynomial-size predicate.

The canonical form is most easily explained using cycle notation for permutations [81]. We require that each cycle consist of a continuous segment of items, that each item map to the immediately succeeding one or, for highest-numbered item in a cycle, to the smallest item in the cycle, and that longer cycles use higher-numbered items than shorter ones. For example, the permutation (12)(345) is in canonical form, but the isomorphic permutations (123)(45), (12)(354) and (15)(234) are not. Formally, given an $n \times n$ permutation matrix A, we have the following predicate in terms of the Boolean entries $A_{i,j}$:

$$\begin{aligned}(\forall i,j|(j > i+1) \Rightarrow \neg A_{i,j}) \bigwedge \\ ((\forall i,j|((j > i) \wedge A_{j,i}) \Rightarrow \\ ((\wedge_{k=i..(j-1)} A_{k,k+1}) \bigwedge (\wedge_{k=(j+1)..(2j-i)} \neg A_{k,j}))))\end{aligned}$$

In this predicate, the condition $(j > i+1) \Rightarrow \neg A_{i,j}$ requires that an item mapped to a higher-numbered item map to the immediately succeeding item: e.g. 3 must map either to 4 (in which case 3 is not the highest-numbered item in its cycle), or to an item numbered not higher than 3 (in which case 3 *is* the highest-numbered item in its cycle). The condition $\wedge_{k=i..(j-1)} A_{k,k+1}$, implied by a backward edge $A_{j,i}(i < j)$, says that every backward edge implies the corresponding forward cycle: e.g. if 5 maps to 3 then 5 must be the highest-numbered item in the cycle and the cycle must be (345). The condition $\wedge_{k=(j+1)..(2j-i)} \neg A_{k,j}$, implied by the presence of a cycle $\begin{pmatrix} i & i+1 & \dots & j-1 & j \end{pmatrix}$, requires the immediately succeeding cycle to be no shorter, in effect sorting cycles by increasing length: e.g. the cycle (345) excludes the cycles (6) and (67). Together with the original constraints restricting A to be a permutation matrix, these constraints permit exactly one permutation with a given multiset of cycle lengths, i.e. one permutation from each isomorphism class.

The size of this predicate is $O(n^3)$ (since its second and largest conjunct is generated by three nested loops iterating over n indices). $O(n^3)$ matches the order of growth of the original constraints. It may be possible to reduce this order of growth by introducing auxiliary Boolean variables, but since n is typically small (under 15) in our analyses, cubic growth has been acceptable.

4.5.3 Relations

Consider the direct product $D = D_1 \times \ldots \times D_k$ of k disjoint finite nonempty sets (we call them *domains*). We define our universe U to be $\mathbb{P}(D)$, the power set of D. Each element of U, called a *relation*, can be represented by $\prod_{i=1}^{k} |D_i|$ bits. Each bit corresponds to an ordered k-tuple $(d_1, \ldots, d_k)$, $d_i \in D_i$, and is $true$ in the binary representation of a relation iff the relation contains the corresponding ordered k-tuple. We will speak interchangeably of the bits and corresponding ordered k-tuples.

Isomorphism classes are defined by treating elements within each domain as indistinguishable. The symmetry group Sym of our universe U is isomorphic to a direct product of k symmetric groups: $Sym \cong \sigma_{|D_1|} \times \ldots \times \sigma_{|D_k|}$. An element $\Theta = (\theta_1, \ldots, \theta_k)$ of Sym maps a relation r to a relation r', such that r' contains an ordered tuple $(d_1, \ldots, d_k)$ iff r contains the ordered tuple $(\theta_1^{-1}(d_1), \ldots, \theta_k^{-1}(d_k))$.

With $|Sym| = \prod_{i=1}^{k} |D_i|!$, direct application of Crawford's method is impractical: the number of symmetries (and hence the size of Crawford's lex-leader predicate) grows super-exponentially with the sizes of the domains D_i. Nevertheless, it is possible to break all symmetries which permute a *single* domain with a linear-size predicate. Even though such symmetries represent only a tiny fraction of all symmetries, experiments show that this predicate rules out most of the isomorphic objects.

We start with an example for the case $k = 2$, then generalize to arbitrary k.

Consider a binary relation $r \in A \times B$, $A = \{a_0, a_1, a_2\}$, $B = \{b_0, b_1, b_2\}$. Let us use the following orderly numbering V for bits of the binary representation of r:

$$
\begin{array}{cccc}
 & b_0 & b_1 & b_2 \\
a_0 & 1 & 2 & 3 \\
a_1 & 4 & 5 & 6 \\
a_2 & 7 & 8 & 9
\end{array}
$$

Under this numbering, Crawford's symmetry-breaking condition for the symmetry exchanging a_0 with a_1 and fixing all other elements (denoted $a_0 \leftrightarrow a_1$) is

$$\overline{123456789} \leq \overline{456123789}$$

where $\overline{123456789}$ denotes the binary number obtained by concatenating the values of Boolean variables $1, \ldots, 9$. This condition simplifies to $\overline{123} \leq \overline{456}$. Together with the condition for $a_1 \leftrightarrow a_2$, we have

$$\overline{123} \leq \overline{456} \leq \overline{789}$$

which breaks all symmetries permuting only A. Similarly, the conditions for $b_0 \leftrightarrow b_1$ and $b_1 \leftrightarrow b_2$ together simplify to

$$\overline{147} \leq \overline{258} \leq \overline{369}$$

breaking all symmetries which permute only B. Together, these conditions allow only those relations for which permuting either the rows *or* the columns (but not both simultaneously) leads to a lexicographically higher (or the same) relation, according to the given bit ordering. These conditions still allow values of r mapped to lexicographically lower values by symmetries which permute *both* A and B.

In general, consider a relation $r \subseteq D_1 \times D_2 \times \ldots \times D_k$. We use Crawford's lexicographic method with the following numbering. Denoting the elements of D_i as $a_{i,0}, a_{i,1}, \ldots, a_{i,|D_i|-1}$, we number the bit corresponding to tuple $(a_{1,e_1}, \ldots, a_{k,e_k})$, $0 \leq e_i < |D_i|$, as

$$\sum_{i=1}^{k} (e_i \times \prod_{j=i+1}^{k} |D_j|)$$

Now consider a transposition $\theta = a_{i,p} \leftrightarrow a_{i,p+1}$. The effect of this transposition on the binary representation of r is to fix all k-tuples except those with p or $p+1$ as their i'th coordinate, and among the tuples with p or $p+1$ as their i'th coordinate, to swap k-tuples differing only in their i'th coordinate. Within each pair of swapped tuples, the tuple with $p+1$ in i'th coordinate is numbered *higher* than the tuple with p in i'th coordinate. Therefore, Crawford's $V \leq \theta(V)$ condition reduces to $P \leq P'$, where P lists the bits corresponding to k-tuples with p in i'th coordinate, in increasing order by number in our numbering, and P' lists the bits corresponding to k-tuples with $p+1$ in i'th coordinate, in increasing order by number in the numbering. Then the right-hand side of Crawford's $V \leq \theta(V)$ condition for $a_{i,p} \leftrightarrow a_{i,p+1}$ equals the left-hand side of the condition for $a_{i,p+1} \leftrightarrow a_{i,p+2}$, so asserting the condition for adjacent pairs of elements breaks *all* permutations which permute only D_i.

The size of this predicate, expressed in conjunctive normal form (CNF), is linear in the size of each domain. The size of a Boolean circuit expressing comparison between two n-bit binary numbers is $O(n)$ [14]. For each domain D_i, we have $|D_i|$ comparators of length $\prod_{j\in\{1,\ldots,i-1,i+1,\ldots,k\}} |D_j|$ for a total comparator size of $O(k \times \prod_{i=1}^{k} |D_i|)$. Measures of effectiveness of this predicate are given in section 4.6.2.

The fact that any matrix has a doubly lexical ordering has been shown previously [54]. Relations of arity greater than two were not considered, and the work was not related to generation of symmetry-breaking predicates. The use of doubly lexical orderings for symmetry-breaking predicates was reported independently in [21].

4.5.4 Functions

A function is a restricted kind of relation: a two-dimensional relation $r \in A \times B$ with each element of A (the domain) related to *exactly one* element of B (the range). Two functions are isomorphic iff they have the same multiset of preimage sizes. In analyses of relational specifications [43], functions occur more frequently than general relations. For functions, we give a polynomial-size symmetry-breaking predicate which breaks *all* symmetries.

First, we break all symmetries permuting only A by sorting the rows of r as binary numbers, as in the preceding section. For notational convenience, here we make the leftmost column (the bits corresponding to b_0) the least significant bit. Second, we require the columns to be sorted by the count of $true$ bits in each column. Formally, the constraints on r read

$$\begin{aligned} &(\forall i \in \{0, \ldots, |A|-2\}\Big| \\ &(\overline{r_{i,|B|-1}r_{i,|B|-2}\cdots r_{i,1}r_{i,0}} \le \overline{r_{i+1,|B|-1}r_{i+1,|B|-2}\cdots r_{i+1,1}r_{i+1,0}})) \bigwedge \\ &(\forall j \in \{0, \ldots, |B|-2\}\Big|(\big|\{i|r_{i,j}\}\big| \le \big|\{i|r_{i,j+1}\}\big|)) \end{aligned}$$

We show that together, these constraints define a *complete* symmetry-breaking predicate.

Since r represents a function, there are $|B|$ possible values for a row of r. Sorting the rows of r makes identical rows adjacent, so that the preimage of each $b_j \in B$ occupies a contiguous segment of A. In addition, for $i < j$, rows mapped to b_i represent smaller binary numbers than rows mapped to b_j. Therefore, elements of A mapped to $b_j \in B$ have lower indices in A than elements of A mapped to b_{j+1}. Alternatively, listing the elements of A in increasing order by index, we first list the elements that map to b_0 (if any), followed by the elements that map to b_1 (if any), and so on, with the elements that map to $b_{|B|-1}$ (if any) at the end of the list.

We now show that adding the second requirement, that the columns be sorted by cardinality (the count of $true$ bits in the column), forces a canonical form. Since all matrices in an isomorphism class have the same multiset of preimage sizes (i.e. column cardinalities), sorting the columns by cardinality uniquely determines the cardinality of each column. In other words, all matrices in an isomorphism class satisfying the column-sorting condition have the same cardinalities in the corresponding columns. But given the constraints described in the preceding paragraph, this uniquely determines the image in B of each $a_i \in A$. If $c_j = |\ \{i|r_{i,j}\}|$, i.e. c_j is the cardinality of th j'th column, then the first c_0 elements of A must map to $b_0 \in B$, the next c_1 elements of A must map to $b_1 \in B$, and so on.

For example, here are three isomorphic function matrices satisfying the row-sorting condition:

	b_0	b_1	b_2	b_3	b_4
a_0	1	0	0	0	0
a_1	1	0	0	0	0
a_2	1	0	0	0	0
a_3	0	0	1	0	0
a_4	0	0	0	0	1
a_5	0	0	0	0	1

	b_0	b_1	b_2	b_3	b_4
a_0	0	1	0	0	0
a_1	0	1	0	0	0
a_2	0	0	1	0	0
a_3	0	0	1	0	0
a_4	0	0	1	0	0
a_5	0	0	0	1	0

	b_0	b_1	b_2	b_3	b_4
a_0	0	0	1	0	0
a_1	0	0	0	1	0
a_2	0	0	0	1	0
a_3	0	0	0	0	1
a_4	0	0	0	0	1
a_5	0	0	0	0	1

Only the rightmost one also orders the column cardinalities, and is the only matrix in the isomorphism class allowed by our symmetry-breaking predicate.

4.5.5 Relations with only one isomorphism class

If the constraint imposed on a relation is such that only one isomorphism class of relations satisfies the constraint, all symmetries on the relation can be broken by setting the relation to one arbitrary member of the isomorphism class. One common case of this is where the relation denotes a total order. A 3×3 relation known to represent the "next" relation of a total order on a set A of 3 elements can be set to the fixed value

	a_0	a_1	a_2
a_0	0	1	0
a_1	0	0	1
a_2	0	0	0

The symmetry-breaking predicate in this case consists of unit clauses forcing each Boolean variable allocated to the relation to a particular Boolean value.

Table 4.1: Values used to measure efficiency of partial symmetry-breaking predicates.

value	formula	meaning
labeled	$\|U\|$	the number of distinct binary representations
unlabeled	from [69, 74]	the number of isomorphism classes
allowed	$\|\{X \in U \| SB(X)\}\|$	# of objects allowed by symmetry-breaking predicate
effic	$\frac{labeled-allowed}{labeled-unlabeled}$	percentage of excludable objects actually excluded
slack	$\frac{allowed}{unlabeled}$	maximum possible improvement factor

4.6 Measuring effectiveness of symmetry-breaking predicates

Symmetry-breaking predicates are designed to speed up search, so it would seem natural to judge their effectiveness by measuring the reduction in search time. This approach has several problems, however. Search times can be highly dependent on the particular backtracking algorithm, and on parameter settings such as the splitting heuristic [17]. The addition of the symmetry-breaking predicate changes the whole search tree (since splitting choices are determined by the entire constraint set), so the comparison to the original constraint problem is not completely clean. Machine-dependent effects such as cache locality can also bias the measurements. Most importantly, end-to-end measurements provide no clue to : how much of the reduction afforded by symmetry are we actually utilizing?

As an alternative measure of efficiency, we can directly measure the pruning power of a symmetry-breaking predicate by counting the number of objects satisfying the predicate. For a complete symmetry-breaking predicate, this number is the number of isomorphism classes. For a partial symmetry-breaking predicate, this number will be higher; the question is, how much higher? Where the number of isomorphism classes is known, we can obtain a precise measure of optimality of our partial symmetry-breaking predicate by comparing its pruning effect with the maximum possible pruning effect.

Table 4.1 describes the numbers computed to measure efficiency of partial symmetry-breaking predicates.

The numbers of isomorphism classes are taken from various works in combinatorics

Table 4.2: Acyclic digraphs: symmetry-breaking efficiency.

n	*labeled*	*unlabeled*	*allowed*	*effic*	*slack*
3	25	6	8	89.47%	1.3
4	543	31	64	93.55%	2.1
5	29,281	302	1024	97.51%	3.4
6	3,781,50	5,984	32,768	99.29%	5.5
7	1,138,779,265	243,668	2,097,152	99.84%	8.6

[69, 74, 29, 61]. The number of objects allowed by the predicate is computed by generating the corresponding satisfiability instance, and counting its solutions with the RELSAT solution counter [7]. Correctness of the implementation was verified by doing complete symmetry-breaking for several classes of objects by Crawford's explicit lexicographical method method, and checking that the number of allowed instances matches the number of isomorphism classes.

4.6.1 Acyclic digraphs

Table 4.2 gives efficiency for DAGs. Even though our symmetry-breaking predicate for DAGs (described in Section 4.5.1) is very compact, it still suffices to remove most of the symmetries. The relative efficiency of the predicate in removing symmetries increases as the size of the DAG (and the difficulty of the search problem) increases.

4.6.2 Relations

We compute the results for binary relations. The number of isomorphism classes of non-homogeneous binary relations is not published, but we can use the number of bipartite graphs, which are closely related to non-homogeneous binary relations. Every non-homogeneous binary relation $r : A \to B$ is a bipartite graph on $|A| + |B|$ nodes, with every edge connecting a node in A with a node in B. All isomorphic relations correspond to the same graph. On the other hand, every bipartite graph of n nodes corresponds to at least one non-homogeneous binary relation $r : A \to B$, with $|A| + |B| = n$. This means that the number of non-isomorphic bipartite graphs of n nodes lower-bounds the number of non-

Table 4.3: Relations: symmetry-breaking efficiency.

n	$labeled$	$unlabeled$	$allowed$	$effic$	$slack$
8	102,528	303	1,057	99.26%	3.5
9	1,327,360	1,119	3,828	99.80%	3.4
10	52,494,848	5,479	38,160	99.94%	7.0
11	1,359,217,664	32,303	228,852	99.99986%	7.0
12	107,509,450,752	251,135	3,970,438	99.99997%	16

isomorphic binary relations $r : A \to B$ with $|A| + |B| = n$. Therefore we can lower-bound the efficiency of our symmetry-breaking predicate by aggregating over relations whose dimensions sum to n.

4.7 Breaking symmetries on Alloy models

So far, we have considered the case where elements of the universe U of objects are single relations. In Alloy models, elements of the universe are Alloy instances, which are tuples of relations. We'll now extend the methods for breaking symmetries on one relation to the problem of breaking symmetries on a tuple of relations.

We cannot simply break symmetries on the individual relations, because different relations might be over the same basic types, and breaking symmetry on a relation destroys the symmetry of its basic types. If we have relations `heap, heap': Obj -> Obj` which are DAGs, we can only break DAG symmetries on one of them. After breaking DAG symmetries on `heap`, the basic type `Obj` is no longer symmetric. However, we can still break symmetries on basic types other than `Obj`.

We fix an ordering of relations, and break symmetries on one relation at a time. While looping through the relations, we keep track of the available basic type symmetries. Initially, all basic types are symmetric. After breaking symmetries on a relation `r: A -> B`, the basic types `A` and `B` are removed from the list of symmetric basic types. If, when we come to a relation, some of the basic types over which it is defined are not symmetric, we do not generate a symmetry-breaking predicate for symmetries that permute atoms of these basic types.

4.8 Experimental measurements

This section gives some empirical evidence that the predicates described in this paper actually improve search time, and that the predicate quality measure described in Section 4.6 helps select efficient predicates. In Table 4.4, analysis times under various degrees of symmetry-breaking are listed for several Alloy models. All times are in seconds. The leftmost column (NoSymm) gives analysis times without symmetry-breaking. The column "OurSymm" gives analysis times with Alloy's default symmetry-breaking options, which use the predicates described in this chapter. The columns "-1", "-2" and "-3" give analysis times with varying numbers of our symmetry-breaking predicates disabled, with "-3" corresponding to the largest number of predicates disabled. The columns "+1", "+2" and "+3" given analysis times with varying numbers of additional symmetry-breaking predicates added; the additional predicates are Crawford's lex-leader predicates for randomly chosen symmetries, and "+3" corresponds to the greatest number of additional predicates.

The table shows that our default choice of predicates can significantly reduce analysis times compared to when no predicates are used, and that partially disabling our predicates generally increases analysis times. On the other hand, the table shows that additional symmetry-breaking on top of our predicates yields only modest, if any, further reduction in search time. This is in correspondence with data in Tables 4.2 and 4.3 showing that our default symmetry-breaking predicates eliminate most, though not all, of the isomorphic solutions.

Removing larger numbers of isomorphic solutions does not uniformly reduce search time, because the addition of symmetry-breaking predicates – like any modification to the Boolean formula – can affect the order in which the SAT solver assigns Boolean variables. Nevertheless, our choice of symmetry-breaking predicates has a positive effect in most cases. In Alloy Analyzer, symmetry-breaking is turned on by default, and no users have reported the need to turn it off.

Table 4.4: Effect of symmetry-breaking predicates on search time.

Model	NoSymm	-3	-2	-1	OurSymm	+1	+2	+3
INS [51]	$>$ 10000s	683s	262s	107s	102s	83s	135s	136s
Chord [75]	$>$ 10000s	1038s	636s	524s	212s	224s	422s	559s
Firewire [71]	$>$ 10000s	101s	106s	179s	174s	203s	246s	224s
MutexRing [18]	3311s	828s	394s	206s	191s	104s	96s	157s
Synchronizer [65]	5709s	3821s	2783s	1458s	788s	1016s	2267s	4568s

4.9 Conclusion and future work

We have presented a uniform method to gauge the effectiveness *and* optimality of symmetry-breaking predicates. The method measures the inherent simplification of the constraint problem, which, unlike running-time measurements, does not depend on the details of a particular backtracking algorithm. The method hinges on our ability to lower-bound the number of isomorphism classes in the universe; these numbers are available for a wide variety of combinatorial objects.

We have also presented specific polynomial-size symmetry-breaking predicates for the types of states commonly occurring in analysis of relational specifications. Measurements show that these predicate exclude over 99% of excludable assignments, and come within an order of magnitude of the optimum. These illustrate the potential usefulness of predicates not derived from Crawford's conditions.

Most interestingly, breaking a random set of symmetries by Crawford's method with short comparators (comparing only a few highest-order bits) often leads to surprisingly effective predicates. Formalizing this observation into a formal randomized symmetry-breaking scheme will be a major goal of future work. Various ways to bias the random selection of symmetries will be investigated. For instance, Crawford's condition for a single symmetry Θ excludes $2^{n-|\Theta|}$ assignments, where $|\Theta|$ is the number of cycles in Θ. This suggests biasing selection towards symmetries with fewer cycles. On the other hand, overlap between sets of states excluded by the selected symmetries should be minimized. This work could relate to work on probabilistic isomorphism testing.

Currently, symmetry-breaking predicates are built based on the inherent symmetries of the problem but without regard to the actual property being tested. This forces the predicate to be true for at least one member of each isomorphism class – even for isomorphism classes ruled out by the property being tested. Future work will explore property-specific symmetry-breaking predicates that use the property being tested to yield a more compact predicate.

Chapter 5

Exploiting Subformula Sharing in Automatic Analysis of Quantified Formulas

Alloy formulas often include quantifiers, which must be grounded out before the formula is translated to CNF and given to a SAT solver. For example, a quantified formula such as `all x: A | F(x)` must be converted to the ground form `F({<a0>}) && F({<a1>}) && F({<a3>})` (for `|A|=3`) before it can be analyzed. The ground formula can get very large, especially for large scopes, deeply nested quantifiers, or complex quantifier bodies. Grounding out can be the bottleneck step of the analysis.

Often, the ground form contains many identical subformulas. The ground form would be much more compact if represented as a DAG in which identical subformulas are shared. However, getting the compact form requires producing the much larger unshared form first, which can be infeasible. This chapter shows how to go directly from the quantified formula to the compact ground form in which identical subformulas are shared, bypassing the unshared ground form.

The solution presented in this chapter is not specific to Alloy semantics. It applies whenever finite-domain quantifiers have to be grounded out. The chapter is therefore presented in terms of an abstract constraint schema of which the Alloy language is one instantiation. This is done to abstract away irrelevant details and to show that the technique can be of interest in non-Alloy contexts.

5.1 Introduction

Quantified formulas – statements such as $\forall x P(x)$ – are frequently used in formal specifications. They allow concise and natural formalization of system properties and, for this reason, are present in many constraint languages. Languages that permit some form of quantifiers include first-order logic, Alloy [44], and Murphi [16]. The recently developed Bounded Model Checking techniques express Linear Temporal Logic formulas as quantified formulas [9].

Constraints with quantifiers can be analyzed in one of two ways: They can be converted to a Quantified Boolean Formula and solved using a QBF solver [25], or the quantifiers can be ground out and the resulting ground form converted to CNF and solved with a SAT solver [28, 63]. Since the ground form can be much larger than the original quantified constraints, grounding out may not be practical in some cases. For the cases for which it is practical, grounding out and applying a SAT solver usually takes less time than converting to QBF and using a QBF solver [26].

In this chapter we present a technique that extends the range of problems for which the "ground out and convert to CNF" approach is practical. The technique speeds up grounding out, and results in smaller CNFs that are solved faster. The resulting CNFs encode subformula sharing information not otherwise available to the SAT solver. The intermediate information we compute about the quantified constraints may be of use to QBF solvers, and the speedup seen in CNF solvers suggests there might be similar benefits to QBF solvers.

The technique takes advantage of the large numbers of identical subformulas often present in ground constraints. Representing the ground form as a DAG allows identical subformulas to be shared. However, since the ground constraints are not explicitly represented in the original (quantified) constraints, identifying opportunities for sharing is nontrivial. Once a ground form is obtained, we could identify identical subtrees, but doing so would require first obtaining the (unshared) ground form, which can be infeasible. On the other hand, identifying isomorphisms in the quantified form (and then grounding out) will miss many opportunities for sharing in the ground form. In this chapter, we describe a technique for directly producing a DAG in which sharing is already present. We identify the structural isomorphisms of the ground form, but perform our analysis on the quantified form.

The remainder of the chapter is organized as follows. Section 5.2 gives an informal Alloy example illustrating the basic approach. Section 5.3 presents an abstract constraint syntax (Subsection 5.3.1), describes how grounding out is performed and how sharing information can be used (Subsection 5.3.2), introduces the notion of a template and describes how templates can be used to detect sharing (Subsection 5.3.3), and elaborates on how templates are detected (Subsection 5.3.4). Section 5.4 gives empirical measurements of improvements obtained by detecting sharing, including an example of a previously intractable problem which our technique makes analyzable. Section 5.5 concludes the chapter and indicates directions of future work.

5.2 Informal illustration

Consider a quantified Alloy formula of the form all s: State | T(F(s),F(OrdNext(s))). Such formulas arise in Bounded Model Checking analyses, explained in Section 2.7. The ground form of this formula is T(F({<s0>}),F({<s1>})) && T(F({<s1>},F({<s2>})) (assuming State={s0,s1,s2}). The subformula F({<s1>} is repeated twice. We'd like the grounding-out procedure to detect when it is about to create a duplicate formula, and to return a reference to a previously generated formula instead.

To achieve this, we first analyze the quantified formula and note that two of its subformulas, F(s) and F(OrdNext(s)), match a common template F(?) with ?=s and ?=OrdNext(s) respectively [1]; let's denote the template as T. During grounding-out, for each template we keep a cache of ground instantiations of the template already generated. Thus, when grounding out the quantifier body for s={<s0>}, we compute the

[1] We don't create the actual parameterized template representations such as F(?), only a unique ID for each template; but in the discussion we'll show the parameterized templates for clarity.

instantiations `F({<s0>}` and `F({<s1>})` of the template T. Then, when grounding out the quantifier body for `s={<s1>}`, we note that `F(s)` is an instantiation of the template `F(?)` for `?={<s1>}`, which we already have in the cache. Instead of computing a new ground form of `F(s)`, we return a reference to the previously computed ground form `F({<s1>})`, creating a DAG. Note that this both saves the grounding-out time and reduces the size of the final ground formula.

The template mechanism ensures the sharing of identical ground subformulas in a variety of situations. In the above example, the quantified formula contained two subformulas matching a common template; but sharing in the ground form can also result from a single subformula of the quantified form. Consider a quantified formula of the form `all p: A, q: B | F(G(p),H(p,q))`. The ground form, for `|A|=|B|=2`, looks like

```
F(G({<A_0>},H({<A_0>,<B_0>}))) && F(G({<A_0>},H({<A_0>,<B_1>}))) &&
F(G({<A_1>},H({<A_1>,<B_0>}))) && F(G({<A_1>},H({<A_1>,<B_1>})))
```

We'd like the two copies of `G({<A_0>}` in the ground form to be shared, as well as the two copies of `G({<A_1>}`. The template mechanism ensures this sharing: it determines that the quantified subformula `G(p)` matches the parameterized template `G(?)` with `?=p`; during grounding-out, it caches the ground instantiations of this template as they are produced; when the body of the quantifier is grounded out for `[p={<A_0>},q={<B_0>}]`, it caches the ground form `G({<A_0>})` with the key of `{<A_0>}`; then when the body of the quantifier is grounded out for `[p={<A_0>},q={<B_1>}]` it reuses the ground form `G({<A_0>})` from the cache.

Another situation in which sharing occurs is when ground-out branches of two different quantified formulas can be shared, even though the quantified formulas themselves cannot be shared because they combine their branches differently. For instance, the quantified formula may have quantified subformulas `all x: A | F(x)` and `some y: A | F(y)`. In the ground form, the ground nodes corresponding to `F({<A_0>})`, `F({<A_1>})` and so on can be shared, even though the ground nodes corresponding to the quantified formulas themselves express different functions (conjunction and disjunction) and cannot be shared.

One other common case handled by the template mechanism is when the quantified formula simply contains two identical subformulas, not parameterized by free quantified variables. For example, the quantified formula may contain the subformula "A in B" in two places. The template mechanism will determine that the two subformulas match the same template "A in B" with the empty argument list [], and will ensure that they're shared in the ground form. A variant of this case is where a quantified formula contains a subformula without quantified variables; for example, all x: A | F(x,R1.R2), where R1 and R2 are relations rather than quantified variables. Each ground branch of the quantifier will contain a copy of R1.R2; the template mechanism will ensure that these copies are shared, without actually generating more than one copy at any point.

We'll now informally describe how template detection is done on the quantified formula, prior to grounding out. For each AST node n, we determine a template and a list of arguments with which the node matches the template. The arguments are free quantified variables of n, or more generally Alloy expressions built out of these free quantified variables and constants. For any assignment to the free quantified variables of n, n represents a particular ground subtree and the template arguments with which n matches its template represent relational constants. For example, the node F(x,R1.R2) has one free quantified variable, x. The node matches the template F(?,R1.R2) with argument list [x]. For an assignment x={<A_0>}, the node represents the ground subtree F({<A_0>},R1.R2) and the template argument list has the value [{<A_0>}].

The templates and template arguments that we detect for all AST nodes satisfy the following *template invariant*. Suppose two AST nodes n_1 and n_2 match the same template, with argument lists L_1 and L_2 respectively. Let a_1 and a_2 be assignments of values to the free quantified variables of n_1 and n_2 respectively. If L_1 under a_1 evaluates to the same sequence of values as L_2 under a_2, then n_1 under a_1 represents the same ground subtree as n_2 under a_2. For instance, suppose the quantified formula contains subformulas all x: A | F(x) and some y: A | F(y). Then the AST nodes F(x) and F(y) match the same template, with argument lists [x] and [y] respectively. Under the quantified variable assignments x={<A_0>} and y={<A_0>}, the two template argument lists have the same value [{<A_0>}]; correspondingly, the two AST nodes both represent the ground form `F({<A_0>})`. When grounding-out an AST node, the value of its template argument list under the current assignment to the quantified variables is used as a hash key into the cache associated with the node's template, and tells us whether we already have the ground form we're about to generate.

Template detection is done by walking the quantified formula's Abstract Syntax Tree in depth-first order; for each node, we first determine the templates matched by the children, then the template matched by the node. We either determine that the currently visited node matches a previously seen template, or create a new template for the node.

There are two base cases. First, all AST leaves denoting a given relation match the same template with an empty argument list; we create one such template per relation. Any two leaves matching such a template have the same ground form, so the template invariant is satisfied. Second, all constant-valued AST nodes – for which the subtree rooted at the node does not include any relations – match the same template, with the node itself as the sole template argument. For example, if `s` and `t` are quantified variables, the the AST nodes `s` and `OrdNext(t)` match the same template with argument lists `[s]` and `[OrdNext(t)]` respectively. The template invariant is trivially satisfied, since any quantified variable assignments that make the argument lists evaluate to the same value also make the two AST nodes evaluate to the same value.

Now, suppose we're visiting a non-leaf, non-constant-valued AST node. First, we determine the templates matched by the node's children. Next, we determine whether the node matches the same template as a previously visited node. Two nodes match the same template if their respective children match corresponding templates, and if the two nodes compute the same function of their children. If the current node doesn't match a previously seen template, we create a new template. In either case, the template argument list is obtained by concatenating the template argument lists of the children. For example, suppose the quantified formula includes subformula `(x.P)+(y.Q)`, where `x` and `y` are quantified variables and `P` and `Q` are relations. We first visit the children and determine that they match templates `[?.p]` and `[?.Q]` with argument lists `[x]` and `[y]` respectively. We then set the template argument list for the root of the subformula (the + operator) to `[x,y]`. Suppose later we visit a subformula `(z.P)+(w.Q)`, where `z` and `w` are quantified variables. We'll determine that it matches the same template as the previously visited (x.P)+(y.Q), because both have two children matching the templates [?.p] and [?.q] and combine them with +. The template argument list for (z.P)+(w.Q) becomes [z,w]. The template invariant is satisfied: e.g. quantified variable assignments [x={<A_0>},y={<A_1>}] and [z={<A_0>},w={<A_1>}] make both template argument lists equal to [{<A_0>},{<A_1>}], and also make both quantified subformulas have the ground form [{{<A_0>}.P)+{<A_1>}.Q].

5.3 Detecting and Using Sharing

In the following desctions, we give a formal description of how templates are detected on the quantified formula and then used during grounding-out to reduce the size and speed up the generation of the ground formula.

5.3.1 Abstract Constraint Schema

Rather than using a specific constraint language, we define an abstract schema that serves as a schema for constraint languages with quantifiers. The only restriction we place on the constraint languages is that quantifiers range over finite domains (so that grounding-out is possible). This abstract schema separates our techniques from the semantics and properties of any particular language.

In Chapter 3, we defined an abstract constraint schema for expressing predicates on a collection of variables. A predicate is expressed as an abstract syntax tree (AST), where each inner node computes a predefined function of its children (the root computing a Boolean which becomes the value of the predicate). Leaf nodes of the tree include the variables the predicate is constraining, quantified variables, and constants. Inner nodes include quantifier nodes. A quantifier node has one child, and defines a finite set of values for a free quantified variable in its subtree. The quantifier node's value is computed by applying its node function to the result of evaluating the quantifier body, as the quantified variable runs over the range. This abstraction can express the standard quantifiers, $\forall$ and $\exists$, but is general enough to express quantified constructs such as set comprehension and

```
U: all values      V: variables           Q: quantified variables
N: nodes           F: node functions      M: templates

is_var(n: N): Bool     // is n a leaf AST node
                       // representing a variable?
node_var(n: N): V      // if isvar(n), return the variable at n
node_func(n: N):F      // what function of its children
                       // does n compute?
node_chldrn(n:N): N*   // return the children of n
node_templ(n:N): M     // the template matched by node n
node_args(n:N): N*     // the argument list with which n matches
                       // node_templ(n)
is_quant(n:N): Bool    // is n a quantifier node?
quant_var(n:N): Q      // the quantified variable declared at n
quant_body(n:N): N     // the sole child node of a quantifier node
quant_range(n:N): U*   // the range of the quantified variable of n
```

Figure 5-1: Definition of notation.

integer summation.

Figure 5.3.1 gives the formal notation used in this chapter. In the context of Alloy, `U` contains relational and Boolean values; `V` contains the relations; `Q` contains the quantified variables declared in quantified formulas such as `all x: A | F(x)` and `some x: A | F(x)` and in comprehension expressions such as `{ x: A | F(x) }`; `F` contains Boolean and relational operators.

5.3.2 Grounding Out

An AST can be converted into a quantifier-free (ground) form by *grounding out* the quantifier nodes.

```
groundout(n: N, qvarvals: Q->U): N {
  return new_node(node_func(n), !is_quant(n) ?
    map(lambda c . groundout(c,qvarvals), node_chldrn(n)) :
    map(lambda u . groundout(quant_body(n),
        qvarvals[quant_var(n)->u]), quant_range(n))) }
```

`qvarvals` gives the values of free quantified variables used in `n`'s subtree. `qvarvals[quant_var(n)->u]` is `qvarvals` with the quantified variable defined at node `n` set to value `u`. `new_node` constructs a new ground node with the specified node function and children.

We would like an "oracle" that keeps track of the ground forms already produced, and tells whether a particular invocation of `groundout` will generate an already-produced ground form. The difficulty lies in determining whether the ground form *about to be* generated matches an existing ground form. Actually generating the ground form and then checking it against already-generated ground forms is not a good solution: since the ground form can be very large, generating it and testing it for isomorphism with existing form only to possibly discard it could seriously impair the efficiency of grounding-out.

In the following sections, we describe how to construct an "oracle" that tells whether an about-to-be-generated ground form of a node has been generated previously, by adding *template annotations* to the quantified tree. The oracle is constructed by analyzing the (small) quantified tree prior to grounding-out; this information is then used during the constructiong of the much larger ground form.

5.3.3 Using Templates to Detect Sharing

Here we describe how template information is used during grounding out. Later, in Section 5.3.4, we describe how templates are detected.

Before grounding out, we compute a *template annotation* for each node of the AST. In effect, we represent every node as an instantiation of a parameterized template. That is, for each node, we discover the template it instantiates and the parameters with which the node instantiates the template. During grounding out, for each template we keep track of ground forms of all nodes that match the template. When we visit a node, we look in its template's cache of ground forms to see whether the ground form we're about to generate is already available.

More specifically, the template annotation of a node `n` comprises a template name `node_templ(n)` and a list of template arguments `node_args(n)`. Each template argument is a constant-valued node in the subtree rooted at `n`. (A node is constant-valued if its subtree contains no non-quantified variables. The ground form of a constant-valued node simplifies to a single value.)

The template information lets us quickly determine whether two given invocations of `groundout` will produce the same ground form. Formally, template information satisfies the following *template invariant*:

```
node_templ(n1) = node_templ(n2)
 && argsMatch(node_args(n1),node_args(n2),A1,A2)
=> groundout(n1,A1)=groundout(n2,A2)

argsMatch(args1, args2: N*, A1, A2: Q->U): Bool
   forall(lambda a1 a2 . eval(a1,A1)=eval(a2,A2), args1, args2)

eval(n: N, a: Q -> U): U { let f=node_func(n) in if(!is_quant(n))
  then { f(map(lambda c . eval(c,a), node_chldrn(n))) }
  else { f(map(lambda u . eval(c,a[quant_var(n)->u]),
               quant_range(n))) }
```

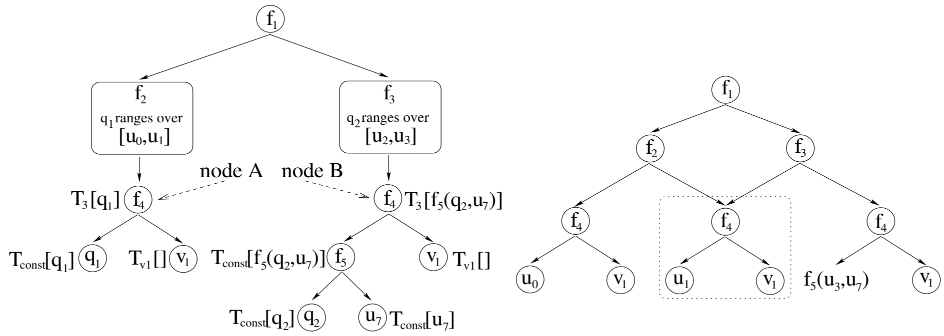


Figure 5-2: Using templates to effect sharing during grounding-out. The DAG on the right is the grounding-out of the AST on the left. Rounded rectangles indicate quantifier nodes. Nodes A and B match the same template T_3. During grounding-out, node A for $q_1 = u_1$ has the same ground form (dotted rectangle) as node B for $q_2 = u_2$, if $f_5(u_2, u_7) = u_1$.

During grounding out, for each template we keep a cache of ground forms keyed on the value of the template argument list. When `groundout` is called to produce the ground form of node n under the quantified variable settings qvarvals, we evaluate the template arguments of n under qvarvals, and use the resulting list of values as a key into the cache of ground forms kept for n's template.

The use of templates to produce sharing of common subformulas is illustrated in Figure 2. Nodes A and B match the same template T_3. During grounding-out we maintain a cache for T_3, mapping argument list values to ground forms. Initially the map is empty. When groundout visits node A with q1=u1, it computes the key into T_3's cache to be [u1] (evaluating node A's template argument list [q1]). This gives a cache miss; groundout computes a ground form (dotted rectangle in Figure 2) and stores it in T_3's cache with the key of [u1]. When groundout subsequently visits node B with q2=u2, it computes the key into T_3's cache as [f5(u2,u7)]. Suppose f5(u2,u7)=u1. Then groundout will get a cache hit, retrieve the previously computed ground form from T_3's cache and return it immediately. The cached ground form will therefore be shared among two nodes, as shown in the rightmost DAG in Figure 2.

5.3.4 Detecting Templates

We describe the template detection algorithm and illustrate it on the running example in Figure 2.

Template detection is done by a single depth-first traversal of the quantified AST. For each node, we determine the template name and template arguments satisfying the template invariant defined in section 5.3.3. When we visit a node, we first recursively determine the template information for the node's children, then use that information to determine the correct template annotation for the node itself. We either decide that the node instantiates a new template, not matched by any previously visited node; or, that it instantiates the same template as a previously visited node. In either case, we determine the actual arguments with which the node we're visiting instantiates the old or new parameterized template.

First, consider two base cases. All leaf nodes referencing a given non-quantified variable match the same template, with no arguments. Any two such nodes have the same ground form, so the template invariant is trivially satisfied. In the running example, the two

v_1 nodes both match the template T_{v1}, with empty argument list [ ].

Another base case involves constant-valued nodes. A node is constant-valued if its subtree references no non-quantified variables; all leaves are either constants or quantified variables. All such nodes match a single template, T_{const}, with the node itself as the sole argument. Since the ground form of a constant-valued node simplifies to a single value from `U`, the template invariant is trivially satisfied. [2] In the running example, the constant-valued nodes q_1 and $f_5(q_2, u_7)$ match T_{const} with argument lists $[q_1]$ and $[f_5(q_2, u_7)]$ respectively. For any quantified variable setting A_1 that sets q_1 and A_2 that sets q_2, the template invariant asserts that whenever q_1 under A_1 evaluates to the same ground form (i.e. to the same element of U) as $f_5(q_2, u_7)$ under A_2, the two constant-valued nodes have the same ground form.

Now we consider template detection for a non-leaf, non-constant node that does not define a quantified variable. There are three such nodes in the running example: node `A`, node `B` and the root; here we will focus on the first two. We need to determine if the node we're visiting matches a previously seen template, or a new template. (Recall that we're traversing the AST in depth-first order and for each AST node determining the template it matches; for the running example, assume that we always visit the left subtree before visiting the right.)

The node `n` we're visiting matches a previously seen template `T` if, for any previously visited node `n'` that matches `T`, the following holds: 1) `n` and `n'` have the same numbers of children, and the corresponding children match the same template; 2) `n` and `n'` compute the same function (e.g. Boolean conjunction) of their children. Regardless or whether `n` matches a previously seen template or a new template, the template argument list with which the currently visited node `n` matches its template is obtained by concatenating the template argument lists of `n`'s children.

For example, when we visit node `A`, it matches a previously unseen template; we compute the template argument list by concatenating $[q_1]$ with [ ] to obtain $[q_1]$. When we

[2] In our actual implementation the AST nodes have types, and there is one template for constant-valued nodes of each type. This prevents AST nodes that will never ground out to the same ground form from matching a common template.

subsequently visit node B, we need to test whether it matches the previously seen template T_3. We take a previously seen node that matches T_3, node A. We observe that nodes A and B compute the same node function (f_4), have the same number of children (2), the left children of both match T_1, and the right children match T_{v1}. We therefore determine that node B matches template T_3, with argument list $[f_5(q_2, u_7)]$.

We will now show that the template annotations computed as described above satisfy the template invariant. Suppose nodes n1 and n2 both match template t and satisfy the three tests; and quantified variable assignments A1,A2: Q->U meet the condition

```
argsMatch(node_args(n1),node_args(n2),A1,A2)
```

Since the template arguments of n1 and n2 were obtained by concatenating the template arguments of their children, we have

```
forall(lambda c1 c2 . argsMatch(node_args(c1),node_args(c2),A1,A2),
       node_ch(n1), node_ch(n2))
```

Assuming child template information is correct, the corresponding children of n1 and n2 ground out to the same ground forms (under A1 and A2 respectively). Since n1 and n2 combine their children using the same functions, the ground forms of n1 and n2 are the same.

Quantifier Nodes

Computing template arguments for a quantifier node has an added complication; template arguments for the child may include the quantified variable introduced at the node, however, the template arguments for this node itself cannot include that variable. This impacts how we compute the template arguments for the node; we cannot simply take the template arguments of the child (as we would do for a one-child non-quantifier node).

Suppose n1 and n2 are two quantifier nodes whose bodies b1 and b2 match the same template with argument lists arg1 and arg2 respectively. Suppose also that n_fn(n1)=n_fn(n2) and qrange(n1)=qrange(n2). Let q1=qvar(n1) and q2=qvar(n2). We'd like to construct args1′ and args2′ such that for any u in U, argsMatch(args1′,args2′,A1,A2) implies argsMatch(args1,args2,A1[q1->u],A2[q2->u]). We first show that

such `args1′` and `args2′` are valid template argument lists for `n1` and `n2`; we will later explain how to construct them. Let `A1`, `A2` be quantified variable settings such that `argsMatch(args1′,args2′,A1,A2)`. Then for any `u` in `qrange(n1)`, we have `argsMatch(args1,args2,A1[q1->u],A2[q2->u])` which in turn implies `grndout(b1,A1[q1->u])=grndout(b2,A2[q2->u])`. Since `n_fn(n1)=n_fn(n2)` and `qrange(n1)=qrange(n2)`, it follows that
`grndout(n1,A1)=grndout(n2,A2)`.

We now explain how to construct `args1′` and `args2′`. We derive `args1′` and `args2′` from `args1` and `args2` (lines 23-29). We start with empty `args1′` and `args2′`. For each pair of corresponding arguments `a1,a2` from `args1,args2` we create a fresh non-quantified variable `fv` and a fresh template detector `td`. We create modified versions `a1′,a2′` of `a1,a2`, where `q1,q2` are replaced by `fv`, and have `td` do template detection on `a1′` and `a2′`. If the templates detected for the nodes `a1′` and `a2′` are different, then the quantifier nodes `n1` and `n2` do not match a common template. Otherwise, we append the template arguments of `a1′` to `args1′` and the template arguments of `a2′` to `args2′`. Note that the template arguments of `a1′` and `a2′` do not reference `q1` and `q2`. If for each pair the templates detected for `a1′` and `a2′` match, then `n1` and `n2` do match the same template with arguments `args1′` and `args2′`.

5.4 Results

We present preliminary performance results for a benchmark suite of six Alloy [44] models.

dijkstra: a model of Dijkstra's algorithm for mutex ordering to prevent deadlocks. We check that the algorithm works correctly for 10 processes and 10 mutexes, for traces up to length 10.

stable_mutex_ring: a model of Dijkstra's self-stabilizing K-state mutual exclusion algorithm for rings [19]. We run a function which finds a non-repeating trace of the system with 3 nodes and 17 steps.

ins: a model of an intentional naming system [51]. We check a structural correctness

	num vars	num vars	num clauses	num clauses
model	sharing	no sharing	sharing	no sharing
dijkstra	40631	55463	95948	123758
stable_mutex_ring	16309	19897	38440	50237
ins	22742	timeout	110564	timeout
chord	22856	43102	58106	104117
shakehands	7706	16575	20539	54673
life	37322	92464	161289	383541

Table 5.1: Formula sizes for benchmarks with and without sharing detection.

	ground-out	ground-out	mchaff	mchaff	bermkin	berkmin
model	sharing	no sharing	sharing	no sharing	sharing	no sharing
dijkstra	6.02	7.82	2.74	9.70	19.87	67.98
stable_mutex_ring	0.91	1.25	18.77	32.65	6.40	24.20
ins	4.01	timeout	2.17	timeout	2.58	timeout
chord	1.01	3.74	55.03	93.04	21.87	48.07
shakehands	0.54	0.89	295.58	timeout	2.14	57.20
life	4.87	13.17	2.04	44.11	3.28	20.67

Table 5.2: Runtimes for benchmarks with and without sharing detection. All times are in seconds.

condition for 4 nodes and 2 name records.

chord: a partial model of the Chord distributed hashtable lookup algorithm for rings [76]. We check a structural correctness condition for 3 nodes and 5 Chord identifiers.

shakehands: a model of a logic puzzle by Paul Halmos involving handshakes between pairs of people. We run a function which solves the puzzle for 10 people.

life: a model of Conway's Game of Life. We run a function which finds an execution of 3 time steps on a 12 point grid.

This suite reflects a variety of modelling idioms, including the BMC-style [9] models which motivated this work. It also balances checking conditions that are satisfiable (stable_mutex_ring, shakehands, and life) with those that are not (dijkstra, ins, chord). The benchmarks were run on a Pentium III 1GHz laptop with 256MB of RAM running Windows 2000.

Table 5.1 shows the effects of our sharing detection algorithm on the size of the generated CNF formula. We measure both the number of variables and the number of clauses. Sharing detection consistently reduces the size of the generated CNF by a large amount, more than a factor of 2 in some cases. For the *ins* model, no CNF was generated without sharing because the grounding out phase ran out of memory, illustrating how sharing detection has made some previously intractable models analyzable.

Runtime comparisons for our benchmarks are given in Table 5.2 (all times are in seconds). We present times for grounding out and solving with two different modern SAT solvers, mchaff [63] and BerkMin [28]. The "no sharing" columns give runtimes with sharing detection disabled. We see consistent and often dramatic improvements with sharing detection enabled for both grounding out and solving. The improvements are seen for both SAT solvers, indicating that the better performance with sharing is independent of differing solver techniques. The *ins* model is particularly interesting, as it is easily analyzable with sharing detection and intractable without. For the *shakehands* model with sharing detection disabled, mchaff was unable to find a solution after 15 minutes of runtime. We plan to implement more optimizations for the sharing detection in the near future, including handling of commutative operators, and we expect to have more results like the *ins* model, where sharing detection makes the difference in tractability.

Two possible factors contribute to the performance improvements. First, the CNF encodings of formulas with shared subtrees is more compact; only one batch of Boolean variables and clauses is needed to encode the shared subtree. As a result, SAT solver operations such as unit propagation execute faster. Second, the subformula sharing information implicitly encoded in the smaller CNF may prevent the solver from performing redundant computations. Understanding the relative importance of these factors will be one direction of future work.

We have described a new algorithm for exploiting structural redundancy in quantified formulas during grounding out. The algorithm reduces running time and memory usage of the groundout procedure, and produces easier-to-solve CNFs. The technique does not depend on the details of the constraint language, and applies to languages that include non-standard quantification constructs.

Results on a variety of software models suggest that the approach is practical. It never worsens performance; often it produces a significant improvement, and in one documented case it made a previously intractable model tractable.

The template annotations produced for nodes of the source tree have simple semantics; they implicitly encode information about the ground form. QBF solvers similarly attempt to derive information about the ground form, without explicitly grounding out. It would be interesting to see whether QBF solvers can use the sharing information to achieve the speedups seen with CNF solvers.

5.5 Conclusion

We have described a new algorithm for exploiting structural redundancy in quantified formulas during grounding out. The algorithm reduces running time and memory usage of the groundout procedure, and produces easier-to-solve CNFs. The technique does not depend on the details of the constraint language, and applies to languages that include non-standard quantification constructs.

Results on a variety of software models suggest that the approach is practical. It never worsens performance; often it produces a significant improvement, and in one documented case it made a previously intractable model tractable.

The template annotations produced for nodes of the source tree have simple semantics; they implicitly encode information about the ground form. QBF solvers similarly attempt to derive information about the ground form, without explicitly grounding out. It would be interesting to see whether QBF solvers can use the sharing information to achieve the speedups seen with CNF solvers.

Chapter 6

Debugging overconstrained declarative models using unsatisfiable cores

This chapter explains the problem overconstraint in declarative modeling, and proposes a solution. An overconstrained model has fewer behaviors than the modeled system. This can prevent us from finding buggy behaviors in a real system, if the model erroneously excludes these behaviors. The possibility of overconstraint greatly reduces users' confidence in correctness reports from a model checker: if no counterexamples to a property are found, is this because the system is correct or because the model is wrong?

What's needed is an automated method for identifying overconstraints in models. In this chapter we describe a method which is based on the ability of SAT solvers to identify an unsatisfiable core of an unsatisfiable CNF formula. An unsatisfiable core is a subset of CNF clauses that by itself makes the formula unsatisfiable. When translating the Alloy model to a CNF formula, we keep track of which Alloy constraints produced which CNF clauses; this lets us map back an unsatisfiable core of the CNF to an unsatisfiable core of the Alloy model. We discuss techniques for obtaininig unsatisfiable cores on the Alloy model that are meaningful to the user, and describe some case studies illustrating the usefulness of overconstraint debugging.

6.1 Introduction

When a model violates a property, the model checker produces a counterexample illustrating the violation. From the counterexample, the user can easily determine whether the counterexample illustrates an error in the modeled algorithm or only an error in the model. By contrast, when a model satisfies a property, the model checker simply reports the absence of counterexamples. Whether the modeled algorithm is actually correct, or the algorithm is broken but the model inadvertently excludes the buggy traces, the model checker's report is the same. This creates many possibilities for missing errors, and undermines the confidence in the results of a model check. In an extreme case, a model has no error traces because it has no traces at all.

The problem is especially severe in tools like Alloy where the model is specified declaratively rather than operationally. It is easy to inadvertently overconstrain a declarative model, ruling out traces that are possible in the modeled algorithm. Even when a modeler suspects an overconstraint, identifying the conflicting constraints is often a great source of frustration. Currently, the only systematic technique for finding causes of conflict is to manually disable individual constraints until the culprits are identified. This task can be lengthy and runs the risk of introducing new errors into the model. The model checker provides no help to the user in finding the overconstraint, other than to report whether a given version of the model is still overconstrained. The lack of a debugger for overconstraints has been one of the biggest complaints of the users of Alloy.

Some work on detecting vacuous satisfaction of properties has been done in the context of temporal logic model checking [8, 6]. In temporal logic model checking, a finite state machine is described by specifying the initial states and the transition relation as Boolean formulas, and a correctness property is specified as a temporal logic formula. The work on vacuity detection shows how to determine whether any subformulas of the correctness property are irrelevant; replacing an irrelevant subformula with an arbitrary subformula would not change the satisfaction of the correctness property by the finite state machine. These results do not apply to debugging overconstraints in Alloy, because in Alloy the overconstraint often occurs in the specification of the algorithm rather than of the property, and because both the algorithm and the property are specified in first-order logic. Published vacuity detection methods can identify the presence of overconstraint in the algorithm (if the entire correctness property is irrelevant), but cannot pinpoint the parts of the algorithm description responsible for overconstraint.

This chapter will address the problem of debugging overconstrained declarative models, where the analysis is done by translation to SAT. Recall that an Alloy model is analyzed by translation to a Boolean formula in conjunctive normal form. When a SAT solver reports that a CNF formula is unsatisfiable, it can report a subset of clauses used in deriving unsatisfiability [86]. This subset is referred to as an "unsatisfiable core" of the CNF formula. Since the CNF was obtained by translating an Alloy model, it is possible to map the unsatisfiable core back onto the Alloy model. In other words, we can identify model constraints which by themselves would produce all clauses in the unsatisfiable core. These model constraints can be shown to the user. If some of the constraints written by the user (whether in the model or in the property) turn out to be irrelevant to showing absence of counterexamples, this could indicate that the model is overconstrained. In case of severe overconstraint, the small part of the model responsible for excluding all solutions would immediately be identified.

This chapter is organized as follows. Section 6.2 shows how unsatisfiable core extraction works from a user's perspective. Section 6.3 gives an informal description of how unsatisfiable core extraction works. Section 6.4 gives a detailed formal description of unsatisfiable core extraction.

6.2 Example of using unsatisfiable core extraction

In this section we describe one example of how unsatisfiable core detection was utilized by Alloy users [24] while modeling a real-world published system. The Alloy model described cryptographic protocols for implementing Controlled Physical Random Functions [23, 22]. Here we'll describe only the model elements relevant to illustrating unsatisfiable core usage; the detailed commented model appears in Appendix B, while the protocols are described in the cited references.

A Physical Random Function (PUF) is a hash function implemented as a particular hardware circuit. The function depends on physical properties of the particular physical circuit (such as timing delays), and is therefore hard to evaluate without physical access to the circuit. A Controlled Physical Random Function (CPUF) is a PUF that can only be accessed via an algorithm that is physically bound to the PUF in an inseparable way. CPUFs can be used to establish a shared secret between a physical device and a remote user. This enables several applications, such as certifying that a specific computation was carried out on a specific processor.

Abstractly, a CPUF can be viewed as a collection of challenge-response pairs (CRPs), or a one-way hash function that computes a response value from a challenge value. Protocols have been developed for managing CRPs of a given CPUF, in the presence of multiple mutually mistrusting parties [23]. These protocols have been modeled in Alloy [24].

The Alloy model has signatures `Principal` and `Val`, representing principals and values. Principals include legitimate and malicious users, system elements (such as the CPUF), and some special-purpose principals described below. Values include all numbers appearing in a protocol: randomly generated values, results of encryption and results of computing one-way hash functions.

An instance of the model describes a two-phase interaction between principals. In the first phase, each principal draws some set of values; each value is drawn by exactly one principal. The drawn values make up the entire set of values used in the interaction; no new values are produced after the initial drawing. After the values are drawn, the principals know disjoint sets of values. In the second phase, principals tell values to each other according to some rules. The telling occurs in a sequence of steps; but because principals never forget values they've been told, the exact order of telling is not important. At the end, each principal knows some set of values that is a superset of the set of values drawn in the first stage. At the end, the question of why a principal p knows a value v can be answered as either "p drew v at the beginning" or "some other principal p' told v to p". The model includes constraints describing the conditions under which a particular principal will share a particular value with another principal; the Alloy Analyzer is used to look for error scenarios in which the constraints do not prevent a malicious principal from learning a secret value.

A special principal, `Computer`, is used to model memoryless computation such as encryption/decryption, pairing, and hashing. For example, if value `A` represents the encryption of value `B` with key `K`, the rules for value movement ensure that `Computer` will only tell `B` to those principals that already possess `A` and `K`. In particular, `Computer` is used to model one-way hashing by the PUF: `Computer` will tell the PUF response to a given challenge only to those principals possessing the challenge. The model includes a set (unary relation) `PUFResponse`, representing those `Val`ues output by the PUF, and a relation `isRespTo: PUFResponse -> Val` representing the value to which each `PUFResponse` is a response. All `PUFResponse` values are drawn by `Computer`, which can then choose to tell these values to some other `Principals`:

```
fact { draws.PUFResponse in Computer }
```

One of the modeled operations, *Renewal*, involves starting with a known challenge-response pair and obtaining another one. The definition of the operation began as follows:

```
fun Renewal(OldChall: Val, OldResp: PUFResponse) {
   User.draws = OldResp + OldChall
   ...
```

An attempt to simulate the renewal operation failed to find any solutions. The overconstraint debugger pinpointed the overconstraint, as illustrated in Figure 6-1. The problem was that in the definition of *Renewal* states that both the challenge and the response comprising the original challenge-response pair are drawn by the user, while an earlier fact states that all responses are drawn by `Computer` – even those responses not computed during the modeled interaction. This contradiction ruled out all solutions regardless of what the remaining constraints said. While this particular overconstraint occurred during simulation, it also could have happened during the checking of an assertion; in that case, a genuine error in the modeled algorithm could have been masked by this overconstraint.

6.3 Computing unsatisfiable cores: informal description

In this section, we give an informal explanation of how unsatisfiable core extraction works. Everything explained here is later formalized in Section 6.4.

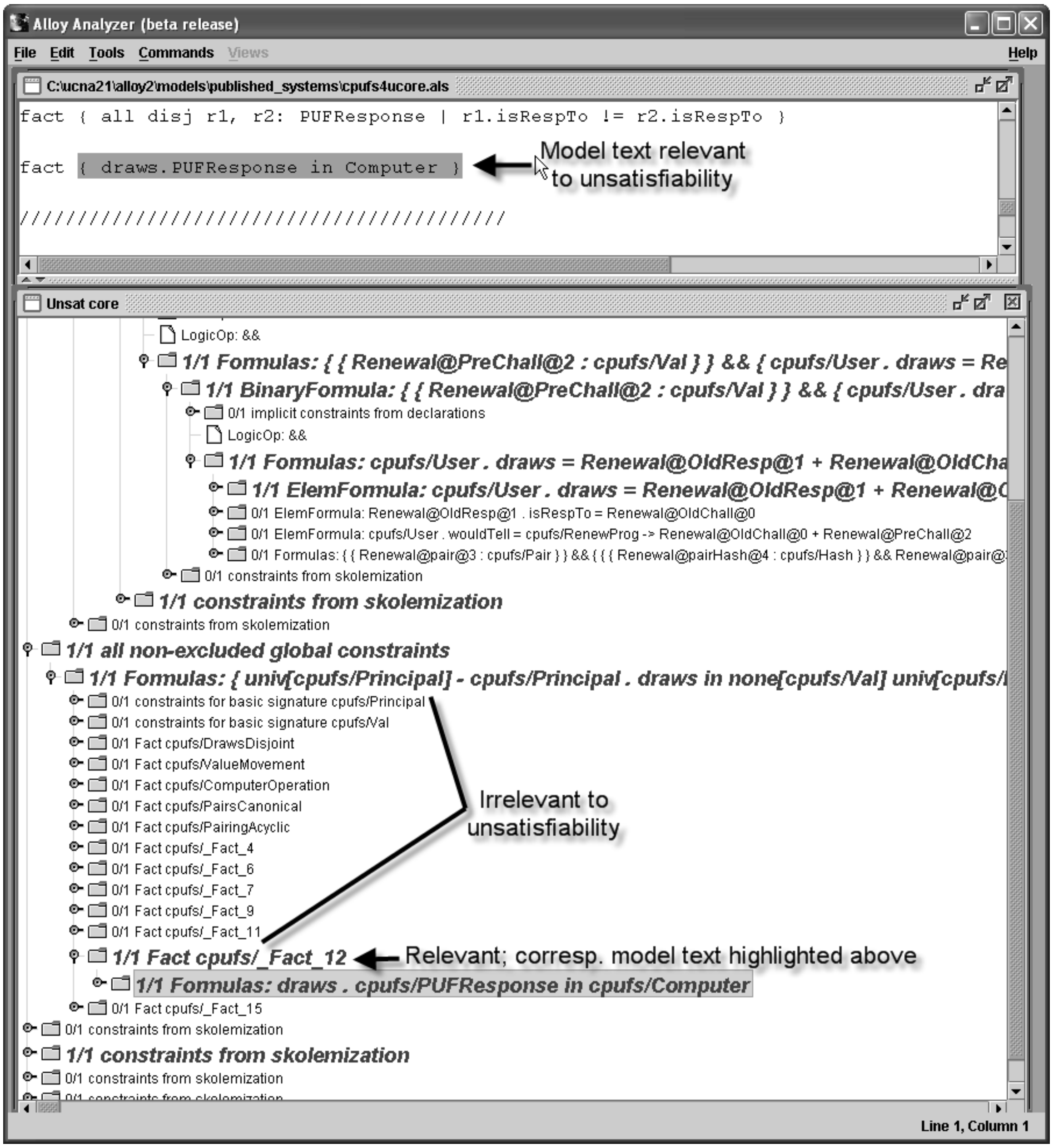


Figure 6-1: Unsatisfiable core – user interface. The lower window shows the unsatisfiable core highlighted on the Abstract Syntax Tree, while the upper window shows the corresponding model text. AST nodes in the unsatisfiable core are shown in bold italic. The annotation "Irrelevant to unsatisfiability" was manually added to the figure; the two slanted lines to its left bracket a group of facts found to be irrelevant to the unsatisfiability proof.

First, we need a definition of unsatisfiable cores for Alloy models. Roughly speaking, the core must be a subset of the model that is still unsatisfiable. For CNF formulas, the definition is simple: any subset of CNF clauses is still a valid CNF formula which is either satisfiable or not. However, an Alloy model is free-form text; not every subset of the text is a valid Alloy model. We define the core not on the model but on the desugared, ground-out abstract syntax tree (AST). (The user interface does let the user see the regions of model text corresponding to specific AST nodes, as shown in Figure 6-1.) The ground form of Alloy formulas was explained in Section 5.2; in short, it is obtained by grounding out all quantifiers. A quantified Alloy formula such as `all x: A | sole x.r` grounds out to

`sole {<A_0>}.r && sole {<A_1>}.r && sole {<A_2>}.r` for `|A|=3`

where `{<A_i>}` denotes the relational constant containing the single tuple `<A_i>`.

Even on the ground AST, properly defining unsatisfiable cores is non-trivial. Simply removing non-core branches can yield a malformed AST which cannot be evaluated on instances (and hence cannot be satisfiable or unsatisfiable). For example, if the single child of a Boolean AST node representing the NOT operator is removed, how is the NOT node (and thus the remaining AST) to be evaluated on Alloy instances?

Another problem with simply removing non-core AST branches is that it's hard to prove that the resulting AST, even if well-formed, is unsatisfiable. We'd like to say that the CNF translation of the core AST includes the CNF clauses comprising the unsatisfiable core of the CNF translation of the original AST. But the CNF translation of an AST depends on the entire structure of the AST; once the structure changes, the correspondence between CNF clauses in the translation of the pruned AST and CNF clauses in the translation of the original AST is lost.

For these reasons, we define unsatisfiable cores of ground ASTs not by pruning non-core branches but by *relaxing* non-core AST nodes. Relaxing an AST node means changing its semantics so it computes an arbitrary function of its children, rather than the function prescribed by the original semantics. For instance, let `N` represent a Boolean conjunction AST node `(A && B)`, with two subformula `A` and `B`. In the original AST, `N` evaluates to *true* on a given instance if and only if both `A` and `B` evaluate to *true* on that instance. If `N` is relaxed, it may evaluate to either $true$ or $false$ on any instance, regardless of what `A` and `B` evaluate to on that instance. Similarly, if `N` is a relational image AST node `(A . B)`, relaxing `N` lets it evaluate to any relational value of the proper relation type, regardless of the relational values to which `A` and `B` evaluate. An AST in which some nodes have been relaxed is called a *relaxed AST*.

In the original AST, there is only one valid evaluation of all AST nodes for a given instance. In the relaxed AST, a given instance may induce a number of possible valid evaluations. A relaxed AST is unsatisfiable iff all possible valid evaluations of it yield $false$ at the root. (A valid evaluation is one in which the non-relaxed nodes obey their original semantics). Relaxing any subset of nodes of the original AST produces a well-formed relaxed AST, with a well-defined notion of unsatisfiability. An unsatisfiable core of the original AST is a subset of AST nodes such that relaxing all non-core nodes yields an unsatisfiable relaxed AST.

We now explain how an unsatisfiable AST core can be derived from an unsatisfiable CNF core. As explained in Chapter 3, a ground Alloy AST can be translated to CNF in a satisfiability-preserving way by allocating Boolean variables to represent the value of each AST node, and for each AST node `N` generating clauses expressing the function that `N` computes of its children. For instance, if `N` represents a Boolean conjunction AST node `(A && B)`, then Boolean variables b_N, b_A, b_B are allocated and CNF clauses are generated requiring b_N to be true iff both b_A and b_B to be true. An additional unit clause is added requiring the Boolean variable representing the Boolean root value of the tree to be $true$. Thus, each CNF clause is generated from a particular AST node.

When the resulting CNF is found unsatisfiable and a clause subset comprising an unsatisfiable core is extracted, we obtain the unsatisfiable core of the AST as those AST nodes from which at least one of the clauses in the unsatisfiable CNF core was generated. Relaxing AST nodes not in the unsatifiable AST core yields an unsatisfiable relaxed AST. To see why, note that a satisfiability-preserving CNF translation of a relaxed AST can be obtained by taking the CNF translation of the original AST and removing clauses generated from the relaxed nodes. For any relaxed AST obtained by relaxing nodes outside the unsatisfiable core, the satisfiability-preserving CNF translation will include all clauses of the unsatisfiable CNF core. Thus, the relaxed AST itself is unsatisfiable.

One question that arises is how unsatisfiable core detection will interact with subformula sharing described in Chapter 5. If some subformulas are shared before conversion to CNF, won't this increase the size of the mapped-back unsatisfiable core of the Alloy model? In practice, this is not a problem because only lower-level subformulas – disconnected from higher-level subformulas – will be added to the unsatisfiable core because of sharing. For example, suppose the Alloy formula has the form $A \wedge (F \vee G) \wedge (H \vee G)$, and the subformula $(F \vee G)$ constitutes an unsatisfiable core. If the repeated subformula G was shared during translation to CNF, then both instances of G will be marked as belonging to the unsatisfiable core when the core is mapped back to the Alloy formula. However, the higher-level formula $(H \vee G)$ *won't* be marked as part of the core, and so the user will be able to determine that the entire subtree $(H \vee G)$ is irrelevant even if some of its children are marked as belonging to the core. In other words, the higher-level AST node will still be relaxed, and that will tell the user that the entire AST subtree rooted at that node can be ignored. A post-processing step can be performed on the mapped-back unsatisfiable core to remove such "stray" subformulas while preserving unsatisfiability of the relaxed AST.

6.4 Computing unsatisfiable cores: formal description

This section gives a formal description of the process for extracting unsatisfiable cores. To emphasize the generality, and the method's independence from the details of Alloy, the description is given not for Alloy formulas but for a more abstract class of formulas (multi-valued circuits) of which ground Alloy formulas with shared subtrees are an instantiation. We first formalize multi-valued circuits, including the evaluation of circuits on particular assignments (instances). We then formalize the notion of relaxing some nodes of multi-valued circuits, informally explained in Section 6.3. Next, we formalize the translation of multi-valued circuits of Boolean DAGs, and define the notion of unsatisfiable cores for Boolean DAGs. Then, we formalize the translation of multi-valued circuits to Boolean DAGs. Finally, we formalize the mapping of unsatisfiable cores of Boolean DAGs to unsatisfiable cores of multi-valued circuits.

6.4.1 Multi-valued circuits

Definition 1. *A* multi-valued circuit (MVC) *is a tuple* $(V, U, N, Rt, ch, var, p)$ *describing a non-deterministic multi-valued function on a set of variables* V*, represented as a circuit.* U *denotes the finite universe of possible circuit node values.* N *is the set of circuit nodes. The circuit is a directed acyclic graph with a single root* $Rt \in N$*.* ch*,* var *and* p *are functions on nodes, describing various node attributes. For a node* $n \in N$*,* ch_n *denotes the sequence of* n*'s children*[1]*;* var_n *denotes the variable read by a leaf node, or* NIL *if* n *does not take its value from a variable;* p_n *denotes a predicate specifying the function that* n *computes of its children.* $p_n(u, [u_1, \ldots, u_k])$ *holds iff* n *may compute the value* u *when* n*'s* k *children have values* $[u_1, \ldots, u_k]$ *respectively.*

Definition 2. *An* assignment *to an MVC is a total function* $a : N \to U$ *that assigns a value* $a_n \in U$ *to each node* $n \in N$*. (Note that an assignment gives values to all circuit nodes, not just to the leaves.) We denote the set of all possible assignments by* A*. An assignment* $a \in A$ *is a* consistent assignment *if it satisfies the following conditions:*

1. *To any two leaves reading the same variable,* a *assigns the same value:*

$$\forall n_1, n_2 : N \mid (v_{n_1} = v_{n_2} \wedge v_{n1}! = NIL) \Rightarrow (a_{n_1} = a_{n_2})$$

2. *The value at each node is consistent with the values at its children:*

$$\forall n : N \mid p_n(a_n, map(a, ch_n))$$

An MVC is satisfiable *if there exists a consistent assignment* a *such that* $a_{Rt} = true$*; such an assignment is called a* satisfying assignment.

Definition 3. Relaxing *a set of MVC nodes means replacing their node predicates with tautologies, thus allowing them to compute an arbitrary function of their children. Formally, let* $M = (V, U, N, Rt, ch, var, p)$ *be a multi-valued circuit. Let* $S \subseteq N$ *be a subset*

[1] We'll sometimes write f_n instead of $f(n)$ to denote the result of applying a function f to an argument n.)

of circuit nodes. Then $Relax(M, S)$ *denotes the circuit* $M' = (V, U, N, Rt, ch, var, p')$*, where* p'_n *is a tautology if* $n \in S$ *and is* p_n *otherwise.*

Definition 4. *An* unsatisfiable core *of an MVC* $M = (V, U, N, Rt, ch, var, p)$ *is a subset* C *of* N*, such that* $Relax(M, N - C)$ *is unsatisfiable.*

Theorem 1. *If* C *is an unsatisfiable core of an MVC* $M = (V, U, N, Rt, ch, var, p)$*, then any superset of* C *is also an unsatisfiable core.*

Proof. Let $C' \subseteq N$ be a superset of C. Suppose $M' = Relax(M, N-C') = (V, U, N, Rt, ch, var, p')$ has a satisfying assignment a. We'll show that a must also be a satisfying assignment of $M'' = Relax(M, N - C) = (V, U, N, Rt, ch, var, p'')$. Note that M' and M'' share the node set N. To any two nodes of N reading the same variable, a assigns the same value because a is a consistent assignment of M'. Also, for a node $n \in C$, p''_n is the same as p'_n, because $n \in C'$; and $p'_n(a_n, map(a, ch_n))$ must hold because $n \in C'$ and a is a consistent assignment of M'. For a node in $N - C$, p''_n is a tautology. Thus, $p''_n(a_n, map(a, ch_n))$ holds for every $n \in N$. Finally, $a_{Rt} = true$ because a is a satisfying assignment of M'. Then a is a satisfying assignment of M'', which contradicts the fact that C is an unsatisfiable core of M. □

6.4.2 Boolean DAGs

One frequently occurring instantiation of multi-valued circuits is a Boolean DAG (BDAG). A BDAG is an MVC of the form $M = (V, U, N, Rt, ch, var, p)$ where $U = \{false, true\}$. Satisfiability of a BDAG can be tested either directly using circuit-based SAT solvers [20, 57], or by converting to CNF [67] and using a CNF-based SAT solver [63, 28, 55].

6.4.3 CNF translation of Boolean DAGs

A *CNF translation* [67] of a BDAG $B = (V, Bool, N, Rt, ch, var, p)$ is a tuple $(BV, Cl, n2bv, n2cl)$. BV is a set of Boolean variables, Cl is a set of CNF clauses, each clause being a set of literals (Boolean variables or their negations). $n2bv : N \to BV$ is a function allocating

Boolean variables to BDAG nodes, and $n2cl : N \rightarrow Cl$ is a function translating each BDAG node to a set of CNF clauses.

$n2bv_n$ is the Boolean variable allocated to node n. To any two BDAG nodes, $n2bv$ allocates the same Boolean variable iff the nodes read the same BDAG variable:

$$\forall n1, n2 : N \mid n2bv_{n_1} = n2bv_{n_2} \Leftrightarrow (v_{n_1}! = NIL \wedge v_{n_1} = v_{n_2})$$

An assignment $ba : BV \rightarrow Bool$ to the CNF variables BV corresponds to an assignment $a : N \rightarrow Bool$ to the BDAG, according to the rule $a_n = ba_{n2bv_n}$.

$n2cl$ is a function denoting the set of CNF clauses translated from each BDAG node. $n2cl_n$ denotes the CNF clauses translated from n. These clauses constrain the Boolean variables in $n2bv_n$ and $map(n2bv, ch_n)$, and are satisfied when an assignment to these variables satisfies p_n. Formally, a CNF assignment $ba : BV \rightarrow Bool$ satisfies the clauses in $n2cl_n$ iff

$$p_n(ba_{n2bv_n}, map(ba, map(n2bv, ch_n)))$$

If $ba : BV \rightarrow Bool$ is a CNF assignment satisfying $n2cl_n$ for all $n \in N$, its corresponding BDAG assignment $a : N \rightarrow Bool$ is a consistent assignment of the BDAG. The BDAG is satisfiable iff the CNF

$$\{(n2bv_{Rt})\} \cup (\bigcup_{n \in N} n2cl_n)$$

is satisfiable; a satisfying assignment to this CNF corresponds to a satisfying assignment to the BDAG. Thus, a CNF-based SAT solver can be used to test the satisfiability of a BDAG.

6.4.4 Unsatisfiables cores of Boolean DAGs

For an unsatisfiable CNF, a CNF-based SAT solver can identify an unsatisfiable core of the CNF: an unsatisfiable subset of the clauses. We'd like to map this subset to an unsatisfiable core of the BDAG. We'll prove the following:

Theorem 2. *Consider a BDAG $B = (V, Bool, N, Rt, ch, var, p)$ with CNF translation*

$T = (BV, Cl, n2bv, n2cl)$. *If* $Core_{CNF} \subseteq Cl$ *is an unsatisfiable core of the CNF, then*

$$Core_{BDAG} = \{n : N \mid (n2cl_n \cap Core_{CNF}) \neq \emptyset\}$$

is an unsatisfiable core of the BDAG.

Proof. Suppose $Core_{BDAG}$ is not an unsatisfiable core of the BDAG; then $B' = Relax(B, N - Core_{BDAG})$ is satisfiable. From the CNF translation $T = (BV, Cl, n2bv, n2cl)$ of B, we can obtain a CNF translation of B' as $T' = (BV, Cl, n2bv, n2cl')$ where $n2cl'_n = n2cl_n$ for $n \in Core_{BDAG}$ and $n2cl'_n = \emptyset$ for $n \notin Core_{BDAG}$. We have $T' \supseteq Core_{CNF}$: any clause in $Core_{CNF}$ comes from translation $n2cl_n$ of some node $n \in N$; that node is included in $Core_{BDAG}$ and its entire translation is then included in T'. Therefore, T' is not satisfiable. On the other hand, since B' is satisfiable, T' must be satisfiable, giving a contradiction. $\square$

6.4.5 Translating MVCs to BDAGs

Satisfiability of a general MVC can be reduced to satisfiability of a BDAG. The satisfiability of the BDAG can then be tested either directly by circuit-based SAT solvers, or by CNF-based SAT solvers after conversion to CNF. Here we formalize the reduction from MVC satisfiability to BDAG satisfiability. In the next section we will see how BDAG unsatisfiable cores can be mapped to MVC unsatisfiable cores.

The translation is illustrated in Figure 6.4.5. Given an MVC $M = (V, U, N, Rt, ch, var, p)$, a BDAG translation of it is a tuple $T = (k, enc, dec, B, v2bv, n2bn, n2bnAux)$. k is the number of bits used to encode U values. $enc : U \to Bool^k$ and $dec : Bool^k \to U$ are functions that encode and decode U values as k-bit binary strings. $B = (BV, Bool, BN, BRt, bch, bvar, bp)$ is the BDAG to which the MVC was translated. $v2bv : V \to BV^k$ allocates to each MVC variable $v \in V$ a sequence $v2bv_v$ of k BDAG variables. $n2bn : N \to BN$ maps each MVC node to a sequence of k BDAG nodes. $n2bn$ is constructed so that for a consistent assignment $a : N \to U$ to M and a consistent assignment $ba : BN \to Bool$ to B, $enc(a_n) = map(ba, n2bn_n)$. The Boolean function computed by each B node in $n2bn_n$ is fully determined by enc and M, though the exact Boolean circuit used to implement this function is not determined. $n2bnAux_n$ denotes auxiliary BDAG nodes used in construct-

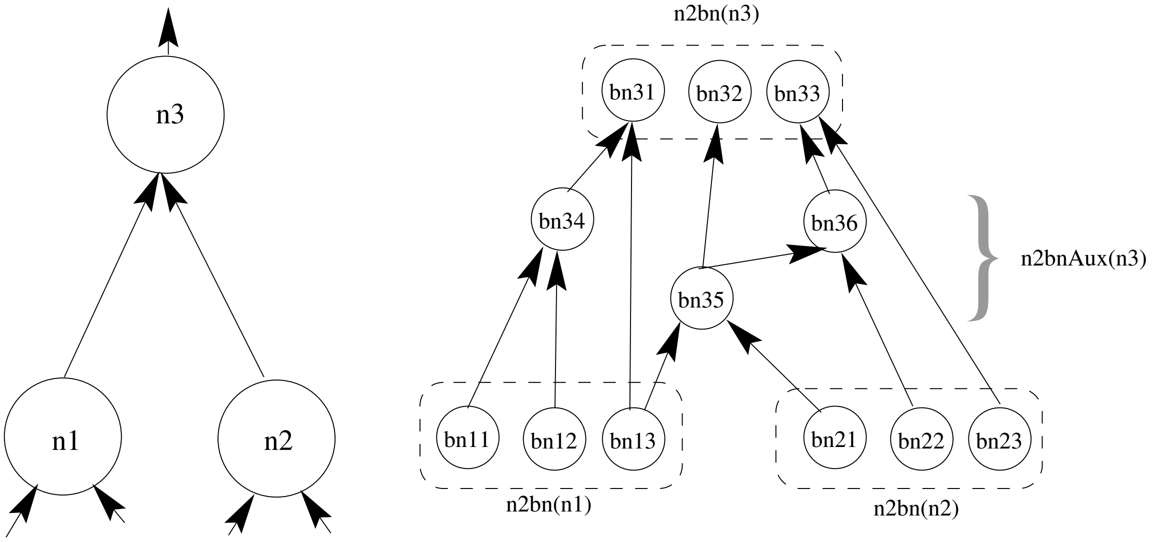


Figure 6-2: Translation of an MVC to a Boolean DAG. MVC node values (members of U) are encoded as 3-bit binary strings. MVC node n_i translates to a sequence of three Boolean DAG nodes $bn_{i1}, bn_{i2}, bn_{i3}$. Translation of n_3 is constructed in terms of translations of its children n_1 and n_2, with the help of auxiliary Boolean nodes bn_{34}, bn_{35} and bn_{36}.

ing the translation of n from the translation of n's children. The BDAG nodes $n2bnAux_n$ are constructed in terms of the BDAG nodes in $map(n2bn, ch_n)$. The BDAG nodes $n2bn_n$ are constructed in terms of BDAG nodes in $n2bnAux_n$ and $map(n2bn, ch_n)$. For any two distinct nodes n, n', $n2bn_n$ and $n2bnAux_n$ do not share any nodes with $n2bn_{n'}$ and $n2bnAux_{n'}$.

6.4.6 Determining unsatisfiable cores of MVCs

Given an unsatisfiable core $Core_{BDAG} \subseteq BN$ of the translation of M, we'd like to find an unsatisfiable core $Core_{MVC}$ of M. We define $Core_{MVC}$ as

$$Core_{MVC} = \{n : N \mid (n2bn_n \cap Core_{BDAG}) \neq \emptyset\}$$

.

Theorem 3. *$Core_{MVC}$ is an unsatisfiable core of M.*

Proof. Suppose $Core_{MVC}$ is not an unsatisfiable core of the BDAG; then $M' = Relax(M, N - Core_{MVC})$ is satisfiable. From the BDAG translation $T = (k, enc, dec, B, v2bv, n2bn, n2bnAux)$

of M, we can obtain a BDAG translation of M' as $T' = (k, enc, dec, B, v2bv, n2bn', n2bnAux)$ where $n2bn'_n = n2bn_n$ for $n \in Core_{MVC}$ and $n2bn'_n = tautology$ for $n \notin Core_{BDAG}$. We have $T' \supseteq Core_{BDAG}$: any node in $Core_{BDAG}$ comes from translation $n2bn_n$ of some node $n \in N$; that node is included in $Core_{MVC}$ and its entire translation is then included in T'. Therefore, T' is not satisfiable. On the other hand, since B' is satisfiable, T' must be satisfiable, giving a contradiction. $\square$

Recall the translation of Alloy predicates to CNF described in Section 3. Suppose that the CNF C translated from our AST is unsatisfiable, and the SAT solver identifies an unsatisfiable core $C' \subseteq C$. We define a predicate $irrel : T \rightarrow Bool$ on AST nodes, which is true for nodes whose translations contributed no clauses to the unsatisfiable core:

$$irrel(t) \equiv \{t \mid transl(t) \cap C' = \emptyset\}$$

Claim: For any node n for which $irrel(n)$ holds, we can replace the node function f_i with an *arbitrary* node function f_j without making the AST satisfiable. To show this, we argue that the CNF translation of the mutated AST will still include the unsatisfiable core.

Proof: The function bv, which allocates Boolean variables to AST nodes, does not depend on node functions; the sequence of Boolean variables allocated to a given AST node depends only on the overall structure of the AST and the position of the node within the AST. Therefore, the same sequences of Boolean variables are allocated to all AST nodes in the mutated AST as in the original AST.

For any node whose node function has not changed, $transl$ will thus output the same clause set. Any node n whose clause set contributed to the core will still have the same node function, and $transl$ will output the same clause set for that node. Each clause of the unsatisfiable core is thus present in the translation of the mutated AST, meaning that the mutated AST is still unsatisfiable.

While the notion of mutating the node function may seem strange, it is essential to getting a good semantic guarantee of correctness of the unsatisfiable core. A naive alternative might be to remove the tree nodes from which no clauses in the core were produced. This may leave the tree in a malformed state, not expressing any valid predicate. Also, the allocation of Boolean variables to tree nodes would change completely, making it difficult to prove that the mutated tree expresses an unsatisfiable predicate. Mutating the node function has the desired effect in common cases. For example, when a top-level conjunct is irrelevant, changing the node function of that conjunct to "constant true" amounts to removing that tree branch.

6.5 Case study in overconstraint debugging: Iolus

We have applied overconstraint debugging to Alloy analysis of Iolus, a scheme for secure multicasting [62, 77][2]. In Iolus, nodes multicast messages to other nodes within a group whose membership changes dynamically. Scalability is achieved by partitioning groups into subgroups, arranged in a tree, each with its own Key Distribution Server (KDS) maintaining a local encryption key shared with members of the subgroup. When a member joins or leaves a subgroup, its KDS generates a new local key and distributes it to the updated list of subgroup members. This was modelled by specifying that after a member joins or leaves, there is a key shared by the new members, and no others. By mistake, the model said the key was shared by the members of the entire *group* – thus including all nodes in contained subgroups. This severely restricted the trace set, potentially masking errors.

We attempted to detect this overconstraint using our constraint core functionality. We first checked an assertion stating that no node can read messages sent to the group when that node was not a group member, one of the correctness properties of the system. There was no counterexample, and unfortunately, the extracted core included most of the constraints in the model. This result can be explained as follows. The error in the model is only a *partial* overconstraint; while the error excludes some legal traces of the system, it still allows many traces violating the correctness property. Therefore, it is not surprising that most of the other constraints in the system are still required to establish correctness. Just because the core contains most of a model does not, unfortunately, imply that the model is free of overconstraint.

[2]The Alloy model of Iolus was written by Mana Taghdiri [77]; the Iolus case study was done by Manu Sridharan [73], who provided this description.

One method of finding overconstraints in this situation is to check correctness properties on a restricted set of traces, where it is still expected that most constraints of the model must be in the core. For the Iolus model, we attempted to check the aformentioned correctness property on traces that had at least three key distribution servers (constraining the size of relations is a typical way to restrict the search space). With this additional restriction, the core no longer included the constraints defining the transitions of the system or the formula stating the property, a clear indication of overconstraint.

Two observations should be noted. First, when an overconstraint is more partial and subtle (as in this case), some thinking by the user will be necessary to find its source, even after the constraint core identifies its existence. This issue is fundamental; when several formulas in a model together overconstrain the system, the core can help to identify them by eliminating irrelevant formulas from consideration, but the reason why the remaining formulas contradict each other may still not be obvious. Second, while this process of checking assertions in restricted spaces to find overconstraints lacks automation, it still has important advantages over the process of finding these overconstraints manually (without core extraction). Previously, a user who suspected an overconstraint in a model would search for it by explicitly checking that classes of legal traces were not ruled out by the system. Our new method of inspecting cores over restricted sets of traces gives more useful information; even if a class of traces is not entirely ruled out by a model, the core may show that important constraints are irrelevant for that class, showing where the overconstraint lies.

6.6 Performance of overconstraint detection

For core extraction we have used a recent modification of the Zchaff satisfiability solver that added core extraction functionality [86]. We found that Zchaff's performance supports interactive identification of overconstraints. The modified solver's performance on unsatisfiable instances was comparable to the performance of the original solver. We have also done some experiments with the BerkMin solver [27, 28]; preliminary experiments indicate that BerkMin's performance is similar to Zchaff's.

An unsatisfiable core can be refined by iterating the solver on the core, pruning away additional clauses irrelevant to unsatisfiability. Running 10-20 such iterations can often reduce the core by about 30%. Since subsequent iterations run on smaller CNF files, the overhead of iteration is often insignificant, especially for severely overconstrained models. However, in our preliminary experiments we have found no significant benefit in additional iterations in terms of what portion of the model was identified as relevant.

6.7 Limitations of using unsatisfiable cores for debugging overconstraints

When we rely on the SAT solver to find an unsatisfiable core, we have no formal guarantees on the quality of the cores that will be found. The particular core that is found depends, among other things, on the decision order and learning strategy of the SAT solver. The solver only tries to minimize the number of CNF clauses in the core; however, when the CNF unsatisfiable core is mapped back to an Alloy unsatisfiable core, factors other than the number of clauses affect the usefulness of the resulting Alloy core. For instance, core extraction is most informative when it shows the irrelevance of a large, contiguous section of the user's model. However, the SAT solver may choose a core that contains a few clauses from each section of the model rather than a similarly-sized core that contains most clauses from some sections and no clauses from others. Since a section of the user's model can only be marked irrelevant if no clauses produced from it are in the CNF core, the former core will cause most of the Alloy model to be included in the mapped-back core. Finding ways to drive the SAT solver to produce more useful CNF cores will be part of future work.

Chapter 7

Conclusion and Future Work

In this thesis, we have described a number of techniques for extending the reach of model checking and illustrated the use of these techniques in the Alloy language and model checker. The techniques include modeling idioms, scalability techniques, and usability improvements.

In modeling idioms, we described *objectification* – a scheme for modeling algorithms that manipulate multiple instances of heterogeneous, graph-like data structures. The complex structures are first-class objects, in the sense that they can be elements of sets and relations. At the same time, the model remains first-order and amenable to automatic analysis. Another modeling idiom we described is *pure-logic modeling*, where the model is not restricted to describing a state machine but instead describes a generic constraint problem. This gives the user great flexibility in defining new kinds of analyses and variations of existing analyses. At the same time, common idioms such as state machines and messaging can still be expressed relatively easily. Common idioms can also be captured in standard Alloy libraries, and in veneer languages that desugar to Alloy [79, 50, 59, 52].

In scalability techniques, we described the use of *symmetry breaking*. We described symmetry-breaking predicates for commonly occurring constructs such as relations and DAGs. We also described a way to gauge the quality of symmetry-breaking predicates relative to an ideal symmetry-breaking predicate, by using solution-counting. Another scalability technique we described is detection and use of *subformula sharing* during analysis of quantified formulas.

In usability improvements, we described an *overconstraint debugger* based on unsatisfiable core extraction. This addresses one of the most frustrating aspects of using a declarative model checker: not knowing whether no counterexamples were found because the algorithm is correct or because the model is wrong.

We have also shown how to integrate all these techniques in a single tool, while retaining simplicity and uniformity of the modeling language. The resulting tool has been used much more widely than its predecessor Alloy Alpha.

A number of areas remain for future work. One important question is how to make the modeling process more systematic. Right now, there is no guaranteed, systematic way to make a model more tractable without reducing the set of scenarios covered by analysis. It would be good to give the user some "profiling" information they could use to adjust their model. The difficulty is that there is no simple measure of "hardness" of CNF problems. One proxy for hardness is formula size; we could tell the user which parts of the model are responsible for the bulk of the generated CNF clauses. A more interesting question is whether we can use information gathered during a search – such as the length of time required to derive some short clauses – to give the user suggestions on making their model more tractable.

Another question is how Alloy can be used for unbounded model checking – that is, for proving that a property is correct for all scopes. One approach is to use windowed induction [72]; checking that a bad state cannot be reached from an initial state in k steps, and that a bad state cannot be reached from an arbitrary state in $k-1$ steps, proves that a bad state cannot be reached by any number of steps. Encoding of such checks in Alloy should be straightforward. Another approach would be to use unsatisfiability proofs produced by SAT solvers for particular scopes. Right now we only use the unsatisfiable core of CNF clauses used in the unsatisfiability proof, and only for debugging overconstraints. It would be good to use the entire proof of unsatisfiability generated by the SAT solver, perhaps to suggest invariants that can be used to prove the property for all scopes. Yet another approach is to add fixpoint operators to the Alloy language, using BDDs for analysis. One more direction that can allow unbounded verification is to combine model checking with theorem proving [5].

Another direction of future work is to add more support for commonly occurring idioms. While it's important to have the flexibility of pure-logic design in the base language, some modeling problems occur frequently enough that special language support for them would be justified. Support can be added as language features, as standard libraries, or as veneer languages that desugar into Alloy.

Appendix A

Text of the railway model

```
//
// A model railroad, illustrating the various
// features of Alloy:
//
//    -- declarative modeling
//    -- handling of graph-like structures
//    -- handling of complex data structures
//

module RR

open std/ord
open std/seq
open std/util

//
// Definition of track topology
//

sig Unit {
   unitConnsA, unitConnsB: set Connector,
   unitPaths: set Path
}

sig Connector { }

sig Path {
   pathA, pathB: Connector,
   pathConns: set Connector  // equal to pathA + pathB
}

fact BasicUnitConstraints {
  all u: Unit | {
     // each side of the unit has at least one connector
     some u.unitConnsA  &&  some u.unitConnsB
     // the two sets of connectors (left and right) are disjoint
     no u.unitConnsA & u.unitConnsB
     // each path in a unit connects a left connector
     // to a right connector
     all p: u.unitPaths |
        p.pathA in u.unitConnsA && p.pathB in u.unitConnsB
     // units are rectangular, so when this unit
     // connects to another unit,  only one side of
     // this unit is used.
     // in other words, no other unit can touch both
```

```
        // sides of this unit.
        all otherUnit: Unit - u | {
            let sharedConns =
                u.(unitConnsA + unitConnsB) &
                otherUnit.(unitConnsA + unitConnsB) |
                sharedConns in u.unitConnsA ||
                sharedConns in u.unitConnsB
        }
    }
}

fact BasicPathConstraints {
  pathConns = pathA + pathB

  all p: Path | {
    // each path belongs to exactly one unit
    one unitPaths.p
    // each path has exactly one connector at each end
    one p.pathA  &&  one p.pathB
    // path atoms are canonicalized: only one path
    // atom per connector pair
    all otherPath: Path - p |
      (otherPath.pathA = p.pathA &&
       otherPath.pathB = p.pathB) => otherPath = p
  }
}

fact BasicConnectorConstraints {
  // At most two units share a connector
  all c: Connector | (# (unitConnsA + unitConnsB).c) < 3
}

// also: for each connector in a unit, there is at least one path.

//
// Some particular kinds of units.
//

sig LinearUnit extends Unit { }

fact LinearUnitStructure {
  all lu: LinearUnit | one lu.unitConnsA && one lu.unitConnsB
}

sig JunctionUnit extends Unit {
   mainLine, sideLine: Connector
}

fact JunctionUnitStructure {
   all ju: JunctionUnit |
     one ju.unitConnsA &&
     ju.unitConnsB = ju.mainLine + ju.sideLine &&
     ju.mainLine != ju.sideLine
}

//fact OnlyStandardUnits { Unit = LinearUnit + JunctionUnit }

////////////////////////////////////////////////////////////////////////

//
// A Route is a sequence of Path's.
//
sig Route {
   routePaths: SeqIdx -> Path,
   firstConn, lastConn: Connector
}

fact RoutesWellFormed {
```

```
  all r: Route | let paths = r.routePaths | {
    // routePaths represents a valid sequence of Paths
    SeqFunValid(paths)

    // adjacent Path's in the sequence must share an endpoint,
    // and must come from different units.
    all i: SeqFunInds(paths) - OrdFirst(SeqIdx) |
       let p = SeqFunAt(paths, i),
           p_next = SeqFunAt(paths, OrdPrev(i)) {
      some p.pathConns & p_next.pathConns
      unitPaths.p != unitPaths.p_next
    }

    // the first connector is the connector of the first path
    // that is not a connector of the second path
    r.firstConn in
      SeqFunAt(paths, OrdFirst(SeqIdx)).pathConns -
      SeqFunAt(paths, OrdNext(OrdFirst(SeqIdx))).pathConns
    // the last connector is the connector of the last path that is
    // not a connector of the next-to-last path
    r.lastConn in
      SeqFunAt(paths, SeqFunLastIdx(paths)).pathConns -
      SeqFunAt(paths, OrdPrev(SeqFunLastIdx(paths))).pathConns

    // first and last connector are distinct; must specify this
    // explicitly because for routes
    // consisting of one path, the above two constraints
    // don't imply this
    sole SeqFunInds(paths) => r.firstConn != r.lastConn
  }
}

////////////////////////////////////////////////////////////////////////

sig Train {
}

sig State {
   // which paths are open, i.e. can physically accomodate a train
   // stretching from one end of the path to the other?
   openPaths: set Path,
   // signal state: may a new train enter this unit
   // through this connector?
   mayEnter: Unit -> Connector,
   // train locations
   trainLoc: Train ->? Route,
   occPaths: set Path
}

fact {
  all s: State | s.occPaths =
     SeqIdx.((Train.(s.trainLoc)).routePaths)
}

fact TrackPhysics {
  all s: State {
    // of all paths in a given unit emanating
    // from a given connector,
    // only one can be open at a time.  this reflects the
    // physical property of rails: a train entering a unit at a
    // connector will deterministically end up on one particular
    // path in that unit emanating from that connector.
    all u: Unit, c: Connector |
       sole pathConns.c & u.unitPaths & s.openPaths
  }
}

// Denotes the set of paths occupied by the given train
```

```
// in the given state.
fun TrainPaths(s: State, t: Train): set Path {
   result = SeqFunElems((t.(s.trainLoc)).routePaths)
}

// Denotes the connector at the beginning of the train,
// in the given state.
fun TrainFirstConn(s: State, t: Train): option Connector {
   result = (t.(s.trainLoc)).firstConn
}

// Denotes the connector at the end of the train,
// in the given state.
fun TrainLastConn(s: State, t: Train): option Connector {
   result = (t.(s.trainLoc)).lastConn
}

// Denotes the units (at most two) that come together
// at the given connector
fun ConnUnits(c: Connector): set Unit
{ result = (unitConnsA+unitConnsB).c }

// Does the connector belong to only one unit (as opposed to
// joining together two units)?
fun IsHangingConnector(c: Connector) { sole ConnUnits(c) }

// Does a path have a "hanging" endpoint (i.e. one that is not
// connected to another unit)?  Train can appear from such a path
// and disappear into such a path.
fun IsHangingPath(p: Path) {
   IsHangingConnector(p.pathA) ||
   IsHangingConnector(p.pathB)
}

// Denotes the train location in the given state,
// or the empty set if this train is not on the track
// in the given state.
fun TrainLoc(s: State, t: Train): option Route {
   result = t.(s.trainLoc)
}

fun TrainAppears(s, s': State, t: Train) {
   no TrainLoc(s,t)
   one TrainPaths(s', t)
   IsHangingConnector(TrainFirstConn(s', t))
}

fun TrainDisappears(s, s': State, t: Train) {
   one TrainPaths(s, t)
   IsHangingConnector(TrainLastConn(s, t))
   no TrainLoc(s,t)
}

fun TrainStays(s, s': State, t: Train) {
   TrainLoc(s,t) = TrainLoc(s',t)
}

fun TrainMovesToNeighboringPath(s, s': State, t: Train) {
   let r = t.(s.trainLoc), r' = t.(s'.trainLoc) | {
     some r
     some r'
     // r' differs from r by removing the first path and
     // tacking a new path onto the end.

     // at the last connector of the route occupied by the train,
     // there are at most two open paths.  (because at any connector
     // at most two units meet, and within any unit there is at most
     // one open path emanating from each connector.)
```

```
      // one of these two paths is the last path of the route
      // occupied by the train.  the other (if it exists and is open)
      // is the path in a neighboring unit, onto which the
      // train may ride.

      let openPathsAtTrainEnd =
          (r.lastConn.~pathConns & s.openPaths),
          newPath = openPathsAtTrainEnd - TrainPaths(s, t) | {
          some newPath &&
          r'.routePaths = SeqFunAdd(SeqFunRest(r.routePaths), newPath)
          r'.lastConn != r.lastConn
      }
   }
}

fun TrainMovesToNeighboringPath2(s, s': State, t: Train) {
   let oldPaths = TrainPaths(s, t), newPaths = TrainPaths(s', t),
       addedPaths = newPaths - oldPaths,
       removedPaths = oldPaths - newPaths,
       r = t.(s.trainLoc), r' = t.(s'.trainLoc) | {
      newPaths in s.openPaths
      r.lastConn in addedPaths.pathConns
      r.firstConn in removedPaths.pathConns
      some addedPaths
      some removedPaths
      r'.lastConn != r.lastConn
   }
}

assert EquivDefs {
   all s, s': State, t: Train | {
     TrainPaths(s, t) in s.openPaths => {
        TrainMovesToNeighboringPath(s, s', t) iff
        TrainMovesToNeighboringPath2(s, s', t)
     }
   }
}

//check EquivDefs for 2 Unit, 10 Connector, 8 Path, 2 Route, 2 SeqIdx, 1 Train, 2 State

fun TrainPhysics(s, s': State) {
   all t: Train |
       TrainStays(s, s', t) ||
       TrainAppears(s, s', t) ||
       TrainMovesToNeighboringPath(s, s', t) ||
       TrainDisappears(s, s', t)
}

fun TrainsRespectSignals(s, s': State) {
   all t: Train | let r = t.(s.trainLoc), r' = t.(s'.trainLoc) |
       (
          // if the last path of the new
          // train location is different from what it was...
          SeqFunLast(r'.routePaths) !in SeqFunElems(r.routePaths) =>
          // then the train had a right to enter the specified unit.
          r.lastConn in
           (unitPaths.SeqFunLast(r'.routePaths)).(s.mayEnter)
       )
}

fun OccupiedPaths(s: State): set Path {
   //result = SeqIdx.((Train.(s.trainLoc)).routePaths)
   result = s.occPaths
}

fun OccupiedUnits(s: State): set Unit {
   result = unitPaths.(OccupiedPaths(s))
}
```

```
fun SafeState(s: State) {
   let occupiedPaths = OccupiedPaths(s) | {
      // in each unit, at most one path is occupied by a train
      all u: Unit | sole u.unitPaths & occupiedPaths
      // trains reside only on open paths
      occupiedPaths in s.openPaths
   }
}

fun SignalPolicy (s: State) {
   // if a unit is occupied by a train, it may not be entered via any connector
   // (the actual property below specifies the contrapositive)
   all c: Connector, u: Unit |
      c in u.(s.mayEnter) => no u.unitPaths & OccupiedPaths(s)
}

fun UnitPolicy(s, s': State) {
   s'.openPaths = s.openPaths
}

fun ShowInvariantViolation(s, s': State) {
   TrainPhysics(s, s')
   TrainsRespectSignals(s, s')
   SignalPolicy(s)
   UnitPolicy(s,s')
   SafeState(s)
   !SafeState(s')
}

//run ShowInvariantViolation for 2 Unit, 8 Connector, 6 Path,
// 4 Route, 2 SeqIdx, 2 Train, 2 State

---------------------------------------------------------

static sig GlobalTrainPlan {
   // for each train, the route to follow through the tracks
   trainPlan: Train ->! Route
}

fun TrainPlan(t: Train): SeqIdx ->? Path {
   result = t.(GlobalTrainPlan.trainPlan).routePaths
}

fact WellFormedTrainPlans {
   all t: Train | let r = t.(GlobalTrainPlan.trainPlan) | {
     // every train's plan starts on a hanging connector,
     // from which a train can appear onto the modeled tracks.
     IsHangingConnector(r.firstConn)

     // every train's plan ends on a hanging connector,
     // from which the train can leave the station.
     IsHangingConnector(r.lastConn)
   }

   // the routes of any two trains are disjoint
   all t1, t2: Train | t1 != t2 =>
     no SeqFunElems(TrainPlan(t1)) & SeqFunElems(TrainPlan(t2))
}


sig PlanState extends State {
   // for each train, its position along its plan;
   // empty set if the train has not yet started
   // or has already finished its plan.
   trainPlanPos: Train ->? SeqIdx,

   // for each train, the next unit on its
```

```
   // plan.  the train may or may not be able to
   // enter that unit.
   trainWish: Train -> Unit,

   // trains that may move.  for each unit,
   // of the trains that want to enter that
   // unit, only one may move.  but it will
   // only actually move if its target unit
   // is free or is being vacated.
   trainMayMove: set Train,

   // trains that have completed their plan
   trainDone: set Train,

   // for some occupied units, evidence assuring us that
   // they will definitely be freed on the next tick
   // (and that it is therefore safe to move a train into them).
   // if <u1,u2> is in unitEmptiedTo, then u1 is
   // occupied and the train from u1 will move to u2,
   // according to our plan for moving trains
   // from the current state.
   unitEmptiedTo: Unit -> Unit
}

// Denotes the set of trains occupying the given
// unit in the given state.
fun TrainsAt(s: State, u: Unit): set Train {
   result = { t: Train | some TrainPaths(s,t) & u.unitPaths }
}

fact DefineTrainWishes {
   // Define the relation trainWish:
   // For each train, to what unit does the train
   // want to move according to its plan?
   all s: State, t: Train - s.trainDone |
      let planStep = if some t.(s.trainLoc) then
         // If train is already on the tracks and moving
         // according to its plan: to next step in the plan
         OrdNext(t.(s.trainPlanPos)) else
         // If train hasn't yet started its plan: to the
         // first step of the plan
         OrdFirst(SeqIdx) |
        t.(s.trainWish) =
         // Denote the unit containing the next
         // path in the train's plan.
         unitPaths.(SeqFunAt(TrainPlan(t),planStep))

   // Define the relation unitEmptiedTo:
   // For each occupied unit u1 that we plan to
   // empty, record the unit u2 _to_ which we
   // plan to send the train from u1.  So, u2
   // serves as a "witness" that u1 will in fact
   // be emptied -- and that therefore we may let
   // another train enter u1.
   all s: State, u1, u2: Unit | {
     u2 in u1.(s.unitEmptiedTo) iff {
        u1 in OccupiedUnits(s) &&
        (let t = TrainsAt(s, u1) | {
            // the train at u1 wants to go to u2
            u2 in t.(s.trainWish) &&
            // and of the trains wanting to go to u2,
            // the train at u1 is given the right to so
            t in s.trainMayMove &&
            // either u2 is unoccupied...
            (u2 !in OccupiedUnits(s) ||
             // ...or the train at u2 will move
             // to some ther unit u3: <u2,u3> is in
             // s.unitEmptiedTo for some u3
```

```
            u2 in (s.unitEmptiedTo).Unit)
        })
      }
    }
}

fun TrainsFollowPlans(s, s': PlanState) {
   // Trains that are done with their plan in state s
   // are still done with their plan in state s'
   s.trainDone in s'.trainDone

   // of all the trains wanting to enter a unit, one may move
   all s: PlanState, u: Unit | {
     // if at least one train wants to enter unit u...
     some (s.trainWish).u =>
       // ...then exactly one train gets permission.
       one (s.trainWish).u & s.trainMayMove
   }

   // For each train, determine its next location.
   all t: Train {
      let r = TrainLoc(s,t), r' = TrainLoc(s',t),
          // planPos and planPos', of type SeqIdx,
          // give the current and next position
          // of the train in its plan.
          planPos = t.(s.trainPlanPos),
          planPos' = t.(s'.trainPlanPos) | {
         {
            // if the train is at the end of its plan,
            // it must disappear
            planPos = SeqFunLastIdx(TrainPlan(t)) => {
               TrainDisappears(s, s', t)
               t in s'.trainDone
               no planPos'
            } else { // t is not at the end of its plan
              t !in s'.trainDone
              // the target unit is the next unit
              // in the train's plan.
              let targetUnit = t.(s.trainWish) | {
                 // should the train move into its
                 // target unit, or stay in place?
                 (targetUnit in OccupiedUnits(s) -
                  (s.unitEmptiedTo).Unit ||
                 t !in s.trainMayMove) => {
                    // either the target unit is occupied
                    // and will not be emptied, or
                    // this train is not chosen to move now.
                    // either way, the train
                    // stays where it is and does not
                    // advance along its plan.
                    planPos' = planPos
                    r' = r
                    TrainPaths(s, t) in s'.openPaths
                 } else {
                     // the target unit is free or will be
                     // freed, and this train is chosen
                     // to move into it.

                     // the train advances along (or begins)
                     // its plan...
                     planPos' = if some planPos then
                      OrdNext(planPos) else OrdFirst(SeqIdx)
                     // ...and moves to the target path.
                     let targetPath = SeqFunAt(TrainPlan(t), t.(s'.trainPlanPos)) | {
                        targetPath in TrainPaths(s', t)
                        // we make the target path open, and
                        // give the train the green light to
                        // enter its target unit.
```

```
                        targetPath in s'.openPaths
                        r.lastConn in targetUnit.(s.mayEnter)
                    }
                }
             }
           }
        }
     }
  }
}

fun Initial(s: State) {
   // there are no trains on the tracks
   no s.trainLoc
   // no train has yet started going through its plan
   no s.trainPlanPos
   // no train has yet completed its plan
   no s.trainDone
}

fun ValidTrace() {
   Initial(OrdFirst(State))
   all s': State - OrdFirst(State) | let s = OrdPrev(s') | {
      TrainPhysics(s, s')
      TrainsRespectSignals(s, s')
      TrainsFollowPlans(s, s')
   }
}

fun StatesAreEquiv(s1, s2: State) {
   s1.trainLoc = s2.trainLoc
}

fun TraceEndsWithLoop() {
   some s: State - OrdLast(State) |  {
        StatesAreEquiv(s, OrdLast(State))
   }
}

fun TrainReachesDest() {
   some t: Train, s: State |  {
     some TrainPaths(s, t)
     # SeqFunInds(TrainPlan(t)) > 1
     TrainPaths(s, t) in SeqFunLast(TrainPlan(t))
   }
}

fun PlanNotFulfilled() {
   Train !in OrdLast(State).trainDone
}

fun Simulate ( ) {
   ValidTrace()
   TraceEndsWithLoop()
   PlanNotFulfilled()
}

-- run Simulate for 2 Unit, 4 Connector, 4 Path, 4 Route,
--    2 SeqIdx, 2 Train, 3 State

fun TrainUnits(s: State, t: Train): set Unit {
   result = unitPaths.TrainPaths(s, t)
}

fun TrainsPassEachOther(s, s': State, t1, t2: Train) {
   some TrainUnits(s, t1)
   some TrainUnits(s', t1)
   some TrainUnits(s, t2)
```

```
   some TrainUnits(s', t2)
   TrainUnits(s', t1) = TrainUnits(s, t2)
   TrainUnits(s', t2) = TrainUnits(s, t1)
}

fun SafeTrace() {
   all s: State | {
      SafeState(s)
      let s' = OrdNext(s) | some s' => {
         all t1, t2: Train |
           t1 != t2 =>
            !TrainsPassEachOther(s, s', t1, t2)
      }
   }
}

fact {
  all s: State | IsDAG(s.unitEmptiedTo)
  //IsDAG(OrdFirst(State).unitEmptiedTo)
}

fun SafetyViolation () {
   ValidTrace()
   !SafeTrace()
}

run SafetyViolation for 3 Unit, 8 Connector, 6 Path,
   8 Route, 2 SeqIdx, 3 Train, 3 State
```

Appendix B

Text of the CPUFs model

```
module cpufs

//
// Model of Controlled Physical Random Functions (CPUFs)
//
// Reference: Blaise Gassend, ``Physical Random Functions'', MIT M.Eng. thesis, 2002.
// http://www.mit.edu/people/gassend/publications/MastersThesisPhysicalRandomFunctions.pdf
//
//

open std/util

///////////////////////////////////////////

//
// Principals and Values
//

//
// An instance of this model represents an interaction between a group of principals,
// which happens in two stages:
//
// 1. Initially, each principal draws some set of values; each value is drawn by exactly
// one principal.  The drawn values make up the entire set of values used in the
// interaction; no new values are produced after the initial drawing.  After the
// values are drawn, the principals know disjoint sets of values.
//
// 2. Principals tell values to each other according to some rules.  The telling
// occurs in a sequence of steps; but because principals never forget values
// they've been told, the exact order of telling is not important.  At the end,
// each principal knows some set of values that is the superset of the set of values
// drawn in the first stage.
//
// At the end, the question of why a principal p knows a value v can be answered
// as either "p drew v at the beginning" or "some other principal p' told v to p".

sig Val {
}

sig Principal {
   // draws: the set of values drawn by this principal at the beginning.
   // the sets of values drawn by any two distinct principals are disjoint.
   draws: set Val,

   // wouldTell: for each of the other principals and for each
   // value, whether this principal would be willing to tell that value
```

```
   // to that principal.
   wouldTell: Principal -> Val,

   // knows: the values this principal knows at the end of the interaction.
   // the value of this relation is defined by the fact 'ValueMovement' below,
   // and used in the definition of wouldTell for specific principals.
   knows: set Val
}

fact DrawsDisjoint {
   // any two distinct principals draw disjoint sets of values.
   all disj p, p': Principal | no p.draws & p'.draws
}


fact ValueMovement {
   // a principal knows the value at the end iff, starting with the principal
   // that drew the value, there is a chain of wouldTell's to this principal.
   all v: Val | v.~knows in Computer + (v.~draws).*(wouldTell.v)
}

/////////////////////////////////////////

//
// Some notable principals:
//


//
// Computer: performs all "memoryless" computation
//
static disj sig Computer extends Principal { }

fact ComputerOperation {
   all p: Principal, v: Val | v in p.(Computer.wouldTell) => {
     PairerWouldTell(p, v) ||
     EncrypterWouldTell(p, v) ||
     HasherWouldTell(p, v) ||
     //PUFWouldTell(p,v) ||
     GetSecretWouldTell(p,v) ||
     GetResponseWouldTell(p,v)
   }
}

//
// Pairer: glues values into pairs, takes them apart again.
//


disj sig Pair extends Val {
   pairA, pairB: Val
}

fact PairsCanonical {
   all p1, p2: Pair | (p1.pairA = p2.pairA && p1.pairB = p2.pairB) => p1 = p2
}

fact PairingAcyclic {
   IsDAG(pairA + pairB)
}

fact { draws.Pair in Computer }

fun PairerWouldTell(p: Principal, v: Val) {
   // either v is a pair and p knows both components of it
  (v in Pair && (v.(pairA + pairB) in p.knows)) ||
  // or p knows some pair of which v is a component
  (v.~(pairA + pairB) in p.knows)
```

```
}

//////////////////////////////////////////

//
// Encrypter: tells ciphertext from plaintex and key,
//      tells plaintext given ciphertext and key.
//

//static disj sig Encrypter extends Principal { }

disj sig Ciphertext extends Val {
   isEncrOf: Val,
   isEncrWith: Val
}

fact {
   all c1,c2: Ciphertext | (c1.isEncrOf = c2.isEncrOf && c1.isEncrWith = c2.isEncrWith) => c1 = c2
}

fact { draws.Ciphertext in Computer }

fun EncrypterWouldTell(p: Principal, v: Val) {
      // either v is ciphertext and p knows both the plaintext and the key...
      (v in Ciphertext && v.(isEncrOf + isEncrWith) in p.knows) ||
      // ...or v is the plaintext of a ciphertext, and p knows both the ciphertext and the key
      (v.~(isEncrOf + isEncrWith) in p.knows)
}

///////////////

//
// Hasher: computes hash values
//

disj sig Hash extends Val {
   isHashOf: Val
}

fact {
   all disj h1, h2: Hash | h1.isHashOf != h2.isHashOf
}

//fact { draws.Hash in Hasher }

fun HasherWouldTell(p: Principal, v: Val) {
   v in Hash
   v.~isHashOf in p.knows
}

//////////////////////////////////////////

//
// PUF: computes a hash function
//

disj sig PUFResponse extends Val {
   isRespTo: Val
}

fact { all disj r1, r2: PUFResponse | r1.isRespTo != r2.isRespTo }

fact { draws.PUFResponse in Computer }

//////////////////////////////////////////

//
// GetSecretComputer and GetResponseComputer:
```

```
//

disj sig Program extends Principal {
   progHash: ProgHash
}

disj sig ProgHash extends Val { }

fact { all p1, p2: Program | p1.progHash = p2.progHash => p1 = p2 }

fun GetSecretWouldTell(p: Principal, v: Val) {
    v.isHashOf.pairA = p.progHash
    v.isHashOf.pairB.isRespTo in p.knows
}

fun GetResponseWouldTell(p: Principal, v: Val) {
   v.isRespTo.isHashOf.pairA = p.progHash
   v.isRespTo.isHashOf.pairB in p.knows
}

static disj sig Oscar extends Principal { }
static disj sig User extends Principal { }

fun SecurityViolation ( ) {
   some Oscar.knows
}

static disj sig RenewProg extends Program { }

fun Renewal(OldChall: Val, OldResp: PUFResponse, PreChall: Val) {
   User.draws = OldResp + OldChall + PreChall
   OldResp.isRespTo = OldChall
   User.wouldTell = RenewProg -> (OldChall + PreChall)
   some pair: Pair, pairHash: Hash | pair.pairA = RenewProg.progHash &&
       pair.pairB = PreChall && pairHash.isHashOf = pair &&
       let newChall = pairHash, newResp = pairHash.~isRespTo | {
         newChall + newResp in User.knows
         newChall + newResp in Oscar.knows
   }
}

run Renewal for 4 Val, 4 Principal, 1 Computer,  1 User, 1 Oscar, 1 Program
```

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Symposium on Operating Systems Principles*, pages 186–201, 1999.

[2] F. Aloul, A. Ramani, I. Markov, , and K. Sakallah. Pbs: A backtrack search pseudo-boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[3] Fadi Aloul, Igor Markov, and Karem Sakallah. Efficient symmetry-breaking for boolean satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[4] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the small scope hypothesis. http://www.lacim.uqam.ca/ plouffe/OEIS/SSH.pdf, September 2002.

[5] K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, Malente, Germany, May 2003.

[6] Roy Armoni, Limor Fix, Alon Flaisher, Orna Grumberg, Nir Piterman, Andreas Tiemeyer, and Moshe Vardi. Enhanced vacuity detection in linear temporal logic. In *Proceedings of 15th Computer-Aided Verification conference*, July 2003.

[7] R. Bayardo and J. Pehoushek. Counting models using connected components. In *AAAI Proceedings*, 2000.

[8] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[9] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Design Automation Conference*, 1999.

[10] Dines Bjrner. Formal software techniques in railway systems. In Eckehard Schnieder, editor, *9th IFAC Symposium on Control in Transportation Systems*, pages 1 – 12, June 2000.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, 1999.

[12] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives*, number 803 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[13] Michelle Crane and Juergen Dingel. Runtime conformance checking of objects using alloy. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[14] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[15] J.M. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry breaking predicates for search problems. In *Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, 1996.

[16] Alan J. Hu David L. Dill, Andreas J. Drexler and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[17] Rina Dechter and Daniel Frost. Backtracking algorithms for constraint satisfaction problems. Technical Report 56, UC-Irvine, 1999.

[18] Edsger Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643 – 644, 1974.

[19] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[20] Feng Lu, Li-C. Wang, and Kwang-Ting Cheng. A circuit sat solver with signal correlation guided learning. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 10892–10897, Munich, Germany, 2003.

[21] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. Van Hentenryck, editor, *Proceedings of Eighth International Conference on Principles and Practice of Constraint Programming (CP02)*. Springer-Verlag, 2002.

[22] Blaise Gassend. Physical random functions. Master's thesis, Massachusetts Institute of Technology, 2003.

[23] Blaise Gassend, Dwaine Clarke, Srinivas Devadas, and Marten van Dijk. Controlled physical random functions. In *18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2002.

[24] Blaise Gassend and Marten van Dijk, January 2004. Personal communication.

[25] E. Giunchiglia, M. Narizzano, and A. Tacchella. Qube: A system for deciding quantified boolean formulas satisfiabilit. In *In Proc. International Joint Conference on Automated Reasoning (IJCAR - 01)*, 2001.

[26] Enrico Giunchiglia, May 2002. Personal communication.

[27] Eugene Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*, Munich, Germany, March 2003.

[28] Evgueni Goldberg and Yakov Novikov. Berkmin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, March 2002.

[29] F. Harary and E.M.Palmer. *Graphical Enumeration*. Academic Press, 1973.

[30] Brant Hashii. Lessons learned using alloy to formally specify mls-pca trusted security architecture. In *2nd ACM Workshop on Formal Methods in Security Engineering – From Specifications to Code*, Washington D.C., October 2004.

[31] Brant Hashii. Using alloy to formally specify mls-pca trusted security architecture. Northrop Grumman Corporation, January 2004.

[32] G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[33] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[34] R. Iosif and R. Sisto. dspin: A dynamic extension of spin. In *Proc. of the 6th SPIN Workshop, LNCS 1680*, 2001.

[35] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1):41–75, August 1996.

[36] Daniel Jackson. Personal communication.

[37] Daniel Jackson. An intermediate design language and its analysis. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, November 1998.

[38] Daniel Jackson. Automating first-order relational logic. In *Proceedings ACM SIGSOFT Conference on Foundations of Software Engineering*, San Diego, November 2000.

[39] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256 – 290, April 2002.

[40] Daniel Jackson, Somesh Jha, , and Craig Damon. Faster checking of software specifications by eliminating isomorphs. In *Proc. ACM Conf. on Principles of Programming Languages (POPL96)*, January 2003.

[41] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, March 1998.

[42] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, June 2000.

[43] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *Proceedings of International Conference on Software Engineering, Limerick, Ireland*, 2000.

[44] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2001.

[45] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, August 2000.

[46] Daniel N. Jackson. A micromodularity mechanism: Talk at fse '01. http://sdg.lcs.mit.edu/ dnj/talks/fse01/alloy-fse-01.pdf.

[47] David Joslin and Amitabha Roy. Exploiting symmetry in lifted csps. In *AAAI97*, 1997.

[48] David Karger, December 2004. Personal communication.

[49] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 1992. http://portal.research.bell-labs.com/orgs/ssr/people/kautz/papers-ftp/satplan.ps.

[50] Sarfraz Khurshid. Generating structurally complex tests from declarative constraints, December 2003. MIT PhD thesis.

[51] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

[52] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Seattle, WA, November 2002.

[53] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, S. Margherita Ligure - Portofino (Italy), May 2003.

[54] Anna Lubiw. Doubly lexical ordering of matrices. *SIAM Journal on Computing*, 16:854–879, 1987.

[55] I. Lynce and J. P. Marques-Silva. Building state-of-the-art sat solvers. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, July 2002.

[56] Ins Lynce and Joo P. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of The Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, May 2004.

[57] M. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik. Combining strengths of circuit-based and cnf-based algorithms for a high-performance sat solver. In *Proceedings of 39th Design Automation Conference(DAC2002)*, pages 747–749, June 2002.

[58] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for testing java programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*, November 2001.

[59] Darko Marinov and Sarfraz Khurshid. Valloy: Virtual functions meet a relational language. In *11th International Symposium of Formal Methods Europe (FME 2002)*, Copenhagen, Denmark, July 2002.

[60] B. D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26:306 – 324, 1998.

[61] Brendan McKay. Personal communication. http://cs.anu.edu/au/people/bdm/nauty/.

[62] Suvo Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings ACM SIGCOMM'97*, pages 277 – 288, Cannes, September 1997.

[63] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.

[64] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[65] Tina Nolte. Exploring filesystem synchronization with lightweight modeling and analysis, August 2002. MIT Masters thesis.

[66] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[67] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[68] Jean-Francois Puget. On the satisfiability of symmetrical constrained satisfaction problems. In Henryk Jan Komorowski and Zbigniew W. Ras, editors, *Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93*, pages 350–361, June 1993.

[69] R.C.Read. *An Atlas of Graphs*. Oxford University Press, 1998.

[70] J. Rintanen. Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI - 99)*, 1999.

[71] J. M. T. Romijn. A timed verification of the (iee) 1394 leader election protocol. In S. Gnesi and D. Latella, editors, *Proceedings of the Fourth International (ERCIM) Workshop on Formal Methods for Industrial Critical Systems (FMICS '99)*, pages pages 3–29, 1999.

[72] Mary Sheeran, Satnam Singh, and Gunnar Stlmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108 – 125, 2000.

[73] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proc. of 19th IEEE International Conference on Automated Software Engineering (ASE 2003)*, 2003.

[74] Neil J. A. Sloane. Sloane's on-line encyclopedia of integer sequences. http://www.research.att.com/ njas/sequences/.

[75] Ian Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conferene on application, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*. ACM Press, 2001.

[76] Ion Stoica, Robert Morris, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM 2001*, August 2001.

[77] Mana Taghdiri. Lightweight modelling and automatic analysis of multicast key management schemes. Master's thesis, Massachusetts Institute of Technology, 2002.

[78] Mana Taghdiri and Daniel Jackson. A lightweight formal analysis of a multicast key management scheme. In *Proc. of the 23rd IFIP International Conference on Formal*

Techniques for Networked and Distributed Systems (FORTE 2003), pages 240 – 256, October 2003.

[79] Mandana Vaziri. A tool for checking structural properties of java code based on a constraint solver, December 2003. MIT PhD thesis.

[80] Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, April 2003.

[81] Herbert Wilf. East side, west side: an introduction to combinatorial families with maple programming. http://www.cis.upenn.edu/ wilf/eastwest.pdf, 1999.

[82] Kirsten Winter and Neil J. Robinson. Modelling large railway interlockings and model checking small ones. In Michael Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2003)*, 2003.

[83] Pamela Zave. A formal model of addressing for interoperating networks. submitted to FME 05, January 2005.

[84] L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Proceedings of 8th International Conference on Principles and Practice of Constraint Programming (CP2002)*, September 2002.

[85] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Proceedings of The Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, May 2003.

[86] Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*, Munich, Germany, March 2003.