Ormolu: Generating Runtime Monitors from Alloy Models

by

Dwayne Lloyd Reeves

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author .....
Department of Electrical Engineering and Computer Science
August 29, 2011

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

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Ormolu: Generating Runtime Monitors from Alloy Models

by

Dwayne Lloyd Reeves

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents Ormolu, a runtime monitor used for monitoring distributed systems. Given an Alloy model, Ormolu generates a database schema and translates the constraints of the model to queries over the database. The translation preserves the semantics of Alloy, especially in regards to its type system. Ormolu allows domain specific knowledge to be expressed in Alloy, where it can be checked and verified. The same model can then be used to check if the constraints of the model are still satisfied at runtime. The feasibility of Ormolu is examined in the domain of air traffic control at a local airport, using data provided by the Tower Flight Data Manager developed by Lincoln Laboratory.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

Acknowledgments

I would like to thank Lincoln Laboratory for sponsoring my research. Without their financial support I would not be able to pursue this graduate degree. I would also like to thank Robert Seater and William Moser from Lincoln Laboratory as well. Their guidance was essential in progressing my research beyond a theoretical idea to an actual working project.

A special thank you to Daniel Jackson for being a constant source of encouragement and advice throughout my time at MIT. As a lecturer in 6.005, you helped me understand the importance of quality in software engineering and understand the true challenge of writing correct programs. As a recitation instructor in 6.033, you brought excitement to technical papers and broadened my understanding of software systems. Finally as a thesis advisor you provided clear instructions whenever we met, and worked extremely hard to ensure I finished this thesis on time.

Thanks to Eunsuk Kang, Joe Near, Aleks Milicevic, Jonathan Edwards and all the members of the Software Design Group. Thank you for the laughs we shared, the conversations we had, and the time we shared together. The support each and everyone of you provided was invaluable when I felt discouraged or lost.

Thanks to all my friends, family, and church family. Without your constant prayers and words of encouragement I would not have had the strength to complete this thesis. A special thanks to my mother who kept me from losing faith in myself, and to God whose strength I rely on daily to make it through the day.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Every software system has a specification that it must satisfy. Software verification is the act of assuring software fulfills its specification. One approach for verifying software is performing static analysis. Static analysis utilizes formal methods to verify a program meets a requirement without actually executing the program. Static type systems are a familiar example. A static type system ensures that a program when executed will not contain type errors.

One method for performing static analysis is model checking. Given a model, a model checker determines if it satisfies the requirements of the system [3]. It is often impractical, if not impossible, to verify a system satisfies its specification statically, i.e., before a system is deployed and running. Runtime testing before deployment can reveal some instances where a system might fail to meet specification, but not all. Once a system is deployed there is a chance that it will take an untested path through the code that will lead to a software failure. For this reason, a companion system is often deployed that actively observes the events of a system and determines if a system enters a state that violates the specification. This companion system is known as a *runtime monitor*.

Model checking and runtime verification are closely related. In fact runtime monitors are often specified using a model checking language such as LTL or other temporal logics [12]. Temporal logics are excellent tools for specifying safety properties and liveness conditions, but as a specialized logic only have a narrow scope of expressible properties.

Temporal logics struggle in expressing constraints between several objects in a specific domain such as an airport. An airport has various objects such as aircrafts, runways, gateways, controllers, airlines, passengers, etc. that interact in complex ways. There are rules and regulations that govern the relationship between these objects in order to maintain safety of the system. An expressive language is needed to capture the specific domain knowledge that describes correct interactions of the system.

This is the motivation behind Ormolu[1], a runtime monitor generated from a specification written in Alloy. Alloy is a modeling language derived from first order relational logic that excels at notating relationships [9]. Given an Alloy model, Ormolu creates an embedded database, known as the analysis database, to store observations of the monitored system. The Alloy model also specifies queries that analyze the contents of the analysis database to ensure it has the properties of the model.

We begin by describing the steps taken to translate the objects and expressions of Alloy to SQL in Chapters 2 and 3. We then examine the role time and state play in Ormolu in

[1]Ormolu is an alloy of metals such as copper, zinc, or tin combined to resemble, or imitate gold.

Chapter 4, before presenting the architecture of Ormolu in Chapter 5. Chapter 6 concludes with an evaluation and discussion of the limitations of the approach. The evaluation is performed on the Tower Flight Data Manager (TFDM), an air traffic control tool being developed by MIT's Lincoln Laboratory.

1.1 Alloy

Alloy is a simple language used to model software systems. The language has been used in various ways including generating test tables for a database, analyzing behavior models, and as a specification for logic programs among others [6, 1, 14]. It combines relational algebra with first order predicate logic succinctly into a single lightweight but highly expressive logic.

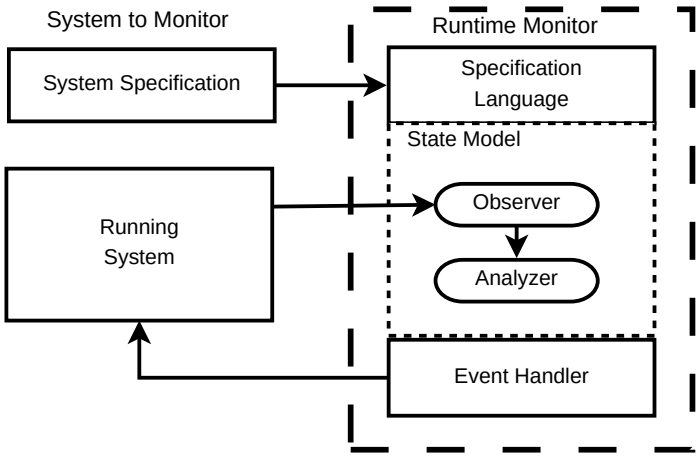
One of the salient features of Alloy is the ability to check and verify assertions of a model automatically. This is done, by translating the model into a SAT problem and using a SAT solver to search for counterexamples to assertions. Alloy can only check an assertion within a limited scope. A scope is an upper bound on the size of instances considered during analysis. Alloy relies on the small scope hypothesis, which claims many bugs in software systems can be detected in small examples [11].

1.2 Runtime Monitors

The first form of monitoring developed for computer programs were program execution monitors. These include debugging systems, runtime profile generators, and system performance monitors. These program execution monitors would serve as the basis for runtime monitors [15].

A runtime monitor is a system that observes the events of a target system and determines if it has entered a state that violates the target system's specification. There are three primary components of a monitoring system, depicted in Figure 1-1. It consists of a specification language that describes the specification, a state model that analyzes observations from the running system, and an event handler that reacts to the output of the state model. These components are described thoroughly in [7], but a brief overview is presented in the subsequent sections.

Figure 1-1: The Components of a Runtime Monitor



1.2.1 Specification Language

The specification language is a runtime monitor's internal representation of a system's specification. In some instances the specification is described directly in the language. In other cases the specification must be translated into the specification language, either manually or automatically using tools. Specification languages differ in runtime monitoring systems in several ways, including the type of language it is based on (first-order logic, relational algebra, etc.), the level of abstraction it provides for a domain, the properties it can capture, and the level at which properties can be checked, i.e., statements, events, system components, etc.

1.2.2 State Model

The state model is responsible for observing the events of a system and determining if the system enters a state that violates the specification described in the specification language. The state model is composed of two subsystems, the observer and the analyzer. The observer receives event traces from the running system, which it then passes to the analyzer. The analyzer updates the internal representation of the state of the system using the new information. Once updated, the analyzer checks that the current state satisfies the specification. If analysis determines a system fails to meet specification, an event will be generated and passed to the event handler.

The level of intrusion a state model requires to monitor a system can vary widely between different runtime monitors. For instance a state model may require denoting manual monitoring points in the source code of a system, at which point it will stop the execution of the system in order to update the state model. Another state model may subscribe to a message queue and perform analysis after a new message is retrieved without significantly impacting the performance of a system. Most state models fall somewhere in between.

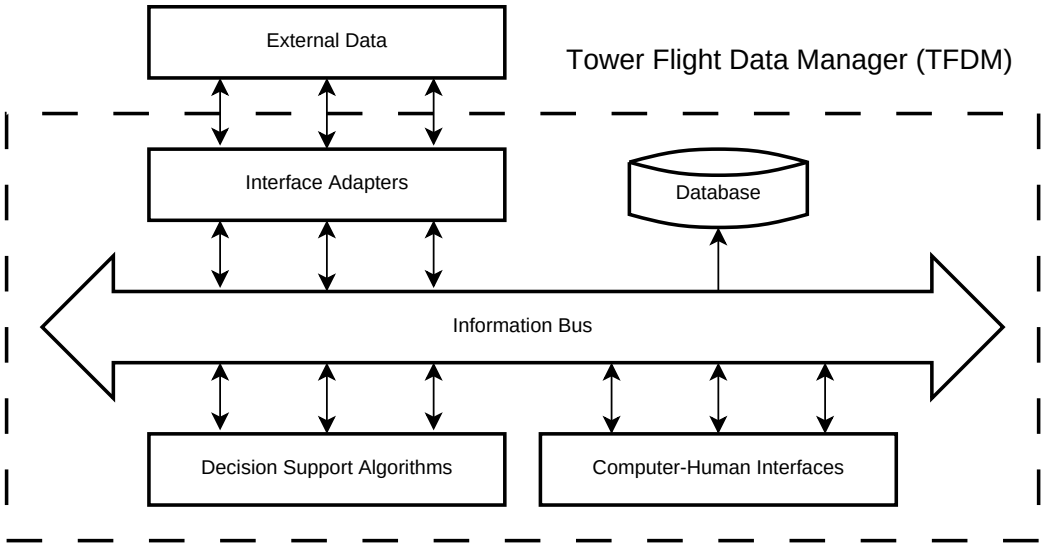
1.2.3 Event Handler

Once a violation is detected by the analyzer in the state model component, the event handler will be notified. The event handler determines how the runtime monitor reacts to violations to the specification. If a runtime monitor is used to enforce the specification it may interact with the target system in order to correct the violation. If a monitor is used to enforce safety, it may react by shutting down part or all of the target system. The runtime monitor may be a passive observer that only logs when the specification is violated. The level of control a user has in determining the actions of the event handler also depends on the runtime monitor. Some will respond to all events the same way, while others allow a user to determine how different events should be handled

1.3 Tower Flight Data Manager

The Tower Flight Data Manager (TFDM) is a prototype air traffic control system in development at MIT's Lincoln Laboratory. The system attempts to combine various external sources of information necessary for managing flights at a local airport, in order to provide decision support tools to operational staff. External sources of information may include flight plan data, traffic flow constraints, flight operations data, weather and hazard conditions, along with terminal and surface surveillance. As a result TFDM will benefit airports by reducing delays, enhancing safety, and improving the robustness of operations.

Figure 1-2: Tower Flight Data Manager System Architecture



TFDM's system architecture relies heavily on a set of publish/subscribe message channels called the information bus. The information bus is composed of various Java-based modules which are necessary to support its role as the primary means of communication in the system. External data is fed into the system via interface adapters that publish the data on the information bus. Decision support algorithms responsible for various operations of an airport, such as runway assignment or taxi routing, will collect data from the bus, perform computations, and then republish the results on the bus. Computer-human interfaces, such as display screens, will display the results of the decision support algorithms that were published on the information bus. All messages sent over the information bus are logged into a database and ultimately moved into a long-term data warehouse over time.

Chapter 2

Translation of Alloy Object Model

In this chapter we demonstrate how Alloy signatures and fields are translated to a series of expressions written in SQL's Data Definition Language (DDL). Tables and foreign key constraints are used to emulate the type system of Alloy. Views over these tables provide a consistent abstraction for writing expressions. These elements form the foundation of Ormolu's analysis database.

2.1 Relations in Alloy and SQL

Alloy is derived partly from relational algebra. The primitive structure in relational algebra is the relation. Given sets $S_1, S_2, \ldots, S_n$, a *relation* is a set of tuples of degree n, whose first element is in S_1, whose second element is in S_2, and so on [4, 5]. Relations of degree, or arity, 1, 2, or 3 are called *unary*, *binary*, and *ternary* relations respectively. Higher degrees of n are called *n-ary* relations.

In Alloy, new relations are defined using signatures and fields. SQL is also derived from relational algebra, and defines new relations using tables or views. Given the common foundation Alloy and SQL share, it should be possible to translate Alloy's notation of relations (signature and fields) to SQL's notation of relations (tables and views).

2.2 Simple Translation

Consider the model presented in Listing 2.1 that models an aircraft's location at an airport. It defines six unary relations – `Location`, `Gate`, `Runway`, `Occupied`, `Taxiway`, and `Aircraft` – and three binary relations – `connections`, `clearConnections`, and `location`.

The simplest translation to SQL is presented in Listing 2.2. It is a direct translation of the relational structure. The unary relations in Alloy are translated to tables with a single column. The binary relations in Alloy are translated to tables with two columns. Each table is given a name that correlates to a name in the model.

While the translation is satisfactory, there are some potential deficiencies. The most significant is the failure to capture the semantics of Alloy's type system. If Alloy's type system is not enforced, inconsistencies can occur. For instance, `Gate` is a subtype of `Location`. Any value in `Gate` must also appear in `Location`. The translation in Listing 2.2 does not enforce this constraint and it is possible to have gates that are not locations. This will result in expressions having different meanings in Ormolu than they do in Alloy. The expression `Gate in Location` always evaluates to true in Alloy, but if the simple translation is used

Listing 2.1: Model of Aircraft Location

```
abstract sig Location {}
sig Gate extends Location{}
sig Runway extends Location {}
sig Occupied in Gate + Runway{}
sig Taxiway extends Location {
  connections: set Gate + Runway,
  clearConnections: connections - Occupied
}

sig Aircraft {
  location: Location
}

fact{
  all a: Aircraft |
    a.location in Gate + Runway => a.location in Occupied
}

pred HasDoubleOccupancy[a1: one Aircraft, a2: one Aircraft] {
  a1 != a2
  (a1 + a2).location in Occupied
  a1.location = a2.location
}

fun DoubleOccupancies[]: Aircraft->Aircraft {
  {a, b: Aircraft | HasDoubleOccupancy[a, b]}
}

run {no DoubleOccupancies}
check {no DoubleOccupancies}
```

Listing 2.2: Simple Translation of Location Model Objects

```
CREATE TABLE Location (col1 VARCHAR(100))
CREATE TABLE Gate (col1 VARCHAR(100))
CREATE TABLE Runway (col1 VARCHAR(100))
CREATE TABLE Occupied (col1 VARCHAR(100))
CREATE TABLE Taxiway (col1 VARCHAR(100))
CREATE TABLE Aircraft (col1 VARCHAR(100))
CREATE TABLE connections (col1 VARCHAR(100), col2 VARCHAR(100))
CREATE TABLE clearConnections (col1 VARCHAR(100), col2 VARCHAR(100))
CREATE TABLE location (col1 VARCHAR(100), col2 VARCHAR(100))
```

the expression could evaluate to false in a SQL expression. It is critical for a translation to enforce type constraints as well as reflect the relational structure of Alloy objects.

2.3 Signature Schemas, Tables, and Views

Signatures serve several important roles in Alloy. A signature:

- introduces a new basic type in Alloy's type system
- introduces a new set of atoms
- contains declarations of new fields

Given these distinct roles, a signature is translated to a database schema. A *schema* describes the structure of a database and introduces a new namespace. A signature's schema contains:

- one or more type tables that store atoms of a given type
- a relation view that returns the atoms contained in a signature
- tables and views for fields declared in the signature
- stored procedures that handle updates to signature and field tables

This section discusses the type table and the relation view of a signature.

2.3.1 The Universal Set

The universal set, denoted by the identifier `univ` in Alloy, is the top of Alloy's type system. All signatures have `univ` as an implicit supertype. The universal set can be viewed as a container for all atoms in an instance of Alloy. The same is true for the translation of `univ` to SQL.

The listing below contains the definition of the `univ` type table. The `univ` type table consist of two columns, key and atom. The atom column stores the unique string representation of an atom. This string is passed into the system and is the external identifier of an atom. No other type tables store this string; instead the key that references the atom string is stored. The key is an integer generated by the database whenever a new atom string is inserted.

Below the type table is the relation view. The relation view is used in Alloy expressions. It has the same name as the schema. The relation view selects only the key from the type table and renames the column as col1. The columns of a relation are always labeled col$i$, where i is the position of the column in the relation. Since signatures are unary relations only one column is needed.

Listing 2.3: Translation of Univ

```
-- Univ Type Table --
CREATE TABLE univ.type (
  key  INTEGER
       GENERATED ALWAYS AS IDENTITY
       PRIMARY KEY,
  atom VARCHAR(100)[1]
       UNIQUE)

-- Univ Relation View --
CREATE VIEW univ.univ (col1) AS
  SELECT key FROM univ.type
```

2.3.2 Top-level Signatures

A top-level signature introduces a new type whose parent is the constant `univ`. `Location` in Listing 2.1 is an example of a top-level signature. The type table of a top-level signature is similar to the type table of `univ`, except it only stores an atom's key. The key is not generated automatically, instead it must reference the key of the `univ` type table. This is known as a *foreign key constraint*.

Listing 2.4: Translation of Top-level Signature

```
-- Location Type Table --
CREATE TABLE Location.type (
  key INTEGER
      PRIMARY KEY
      REFERENCES univ.type(key)
      ON DELETE CASCADE)

-- Location Relation View --
CREATE VIEW Location.Location (col1) AS
  SELECT key FROM Location.type
```

If a foreign key constraint is declared, then an insertion is rejected if the value of the column is not present in the referenced column. Foreign key constraints enforce type constraints when new atoms are inserted into a signature. `ON DELETE CASCADE` instructs the database to delete a tuple if it is deleted in the reference table. This enforces type constraints when atoms are removed from a signature. Whether atoms are added or deleted, `Location` will always be a subset of `univ`. The relation view is identical to the relation view of `univ`.

2.3.3 Extension Signatures

An extension signature introduces a new type whose parent is another signature. `Runway` in Listing 2.1 is an example of an extension signature. The type table of an extension signature is identical to a top-level signature's type table, except the table referenced in the foreign key constraint is the type table of the signature that is extended. In the case of `Runway`, this would be `Location`.

Listing 2.5: Translation of Extension Signature

```
-- Runway Type Table --
CREATE TABLE Runway.type (
  key INTEGER
      PRIMARY KEY
      REFERENCES Location.type(key)
      ON DELETE CASCADE)

-- Runway Relation View --
CREATE VIEW Runway.Runway (col1) AS
  SELECT key FROM Runway.type
```

2.3.4 Subset Signatures

A subset signature is a signature that is a subset of one or more parent signatures. Unlike other signatures, subset signatures do not introduce a new type, so instead they have an ofType table. Subset signatures also differ in that it is possible to define a union of signature types. Union types cannot be expressed using a foreign key constraint on a single table, so a different table is created to store atoms of each type.

Listing 2.6: Translation of Subset Signature

```
-- Occupied OfType Tables --
CREATE TABLE Occupied.ofType_Gate(
  key INTEGER PRIMARY KEY
      REFERENCES Gate.type(key) ON DELETE CASCADE)

CREATE TABLE Occupied.ofType_Runway(
  key INTEGER PRIMARY KEY
      REFERENCES Runway.type(key) ON DELETE CASCADE)

-- Occupied Relation View --
CREATE VIEW Occupied.Occupied (col1)
  AS SELECT * FROM Occupied.ofType_Gate UNION
     SELECT * FROM Occupied.ofType_Runway
```

Listing 2.6 depicts the translation of the subset signature `Occupied`. `Occupied` is the union of two signatures `Gate` and `Runway`. An ofType table is defined for each parent. The relation view is the union of the ofType tables. If a subset signature has n parents, n ofType tables will be created and the union of all tables is the relation view.

2.4 Field Tables

A field declaration in Alloy is of the form `sig A {f: e}`, where `A` is the name of the signature, `f` is the name of the field and `e` is a bound expression. If `e` has arity n, the field will have arity $n+1$. The type of a field is determined by its bound expression. The type of an Alloy expression is a union of products.[2] The type of the `connections` field in `Taxiway`

[2] A detailed description of the Alloy type system is found in [10, p. 107-117]

is $Taxiway \rightarrow Gate + Taxiway \rightarrow Runway$, where $\rightarrow$ is the product operator and $+$ is the union operator. This type is used to translate the field to SQL.

Listing 2.7: Translation of Field

```
-- connections Type Tables --
CREATE TABLE Taxiway.connections_Gate (
  key1 INTEGER PRIMARY KEY
       REFERENCES Taxiway.type(key) ON DELETE CASCADE,
  key2 INTEGER PRIMARY KEY
       REFERENCES Gate.type(key) ON DELETE CASCADE)

CREATE TABLE Taxiway.connections_Runway (
  key1 INTEGER PRIMARY KEY
       REFERENCES Taxiway.type(key) ON DELETE CASCADE,
  key2 INTEGER PRIMARY KEY
       REFERENCES Runway.type(key) ON DELETE CASCADE)

-- connections Relation View --
CREATE VIEW Taxiway.connections (col1, col2)
  AS SELECT * FROM Taxiway.connections_Gate UNION
     SELECT * FROM Taxiway.connections_Runway
```

It is not possible to express unions types using foreign key constraints on a single table. Instead multiple type tables are created. The name of the type table is the field name with the type appended to the end. Listing 2.7 illustrates the translation of `connections` in `Taxiway`. The types $Taxiway \rightarrow Gate$ and $Taxiway \rightarrow Runway$ are translated to two tables each with two columns. Each column references the respective type table.

The relation view of the field is defined below the type tables. The number of columns of the view is equal to the arity of the field. It is expressed as the union of all type tables for the field. In Listing 2.7, the connections with type $Taxiway \rightarrow Gate$ are stored in `Taxiway.connections_Gate` while the connections with type $Taxiway \rightarrow Runway$ are stored in `Taxiway.connections_Runway`. The relation view is the union of both type tables, creating a single relation.

Chapter 3

Translation of Alloy Expressions

In this chapter we present a methodology for translating an arbitrary Alloy 4 expression to a valid SQL expression. We first define the restrictions that are placed on an Alloy expression. Then we present a procedure that rewrites an Alloy expression to Ormolu's Intermediate Language (OIL)[1]. Finally we describe how an OIL expression is translated to a SQL expression.

3.1 Restrictions on Alloy Expressions

Alloy supports several operators taken primarily from predicate and relational calculus. Most of these operators are easily expressed in SQL. However there are some that pose a significant challenge and are not supported. If an unsupported expression is encountered, the Ormolu compiler will reject the model.

3.1.1 Integer Expressions

There are two separate notions of integers in Alloy. An integer value is an integer in the traditional mathematical sense. Each integer value is associated with an integer atom. Integer atoms are regular atoms in Alloy, and can appear in sets and relations. The built-in `Int` signature contains all integer atoms. The set of integers is then bounded during analysis. [2]

It is possible to translate simple integer values to SQL, but sets of integers pose a more difficult challenge. If the `Int` signature is treated as other signatures then a table must be created containing all the integer atoms. This will result in a table with an entry for every possible integer. This approach would be inefficient thus requiring special treatment of the `Int` signature. To prevent complicating the translation process integer expressions and operations on integers are not supported. This includes the functions and predicates defined in the util/integer module.

3.1.2 Closures and Other Restrictions

Transitive and reflexive-transitive closures are used to express reachability conditions. There are two possible ways of translating a closure to SQL. One option is to create a user-defined

[1] Not to be confused with OIL, the Web-based ontology language.

[2] This is true for Alloy 4.1.10. A future release of Alloy will eliminate the distinction between integer values and sets of integers. All integers will be treated as sets of integers.

expr	::=	**let** letDecl,+ blockOrBar
	|	quant decl,+ blockOrBar
	|	unOp expr
	|	expr binOp expr
	|	expr arrowOp expr
	|	expr [**not**]? compareOp expr
	|	expr **implies** expr **else** expr
	|	expr [expr,*]
	|	number | − number
	|	**none** | **iden** | **univ** | **Int** | **seq/Int**
	|	name
	|	block
	|	**{** decl,+ blockOrBar **}**
decl	::=	[**disj**] name,+ **:** [**disj**] expr
letDecl	::=	name **=** expr
quant	::=	**all** | **no** | **some** | **lone** | **one** | **sum**
binOp	::=	**or** | **and** | **iff** | **implies**
	|	& | + | −
	|	++ | <: | :> | **.**
	|	<< | >> | <<<
arrowOp	::=	[**some** | **one** | **lone** | **set**]? → [**some** | **one** | **lone** | **set**]?
compareOp	::=	**=** | **in** | < | > | <= | >=
unOp	::=	**not** | **no** | **some** | **lone** | **one**
	|	**set** | **seq** | # | ∼ | ***** | ∧
block	::=	**{** expr* **}**
blockOrBar	::=	block
blockOrBar	::=	| expr
name	::=	[**this** | ID] [/ ID]*

Figure 3-1: Grammar of Alloy 4 Core Elements

function that recursively computes the closure and returns the final result set. Another option is to express the closure as a recursive query. The standard method for expressing recursive queries is using Common Table Expressions (CTE). CTE were introduced in SQL:1999 and are supported by most major database vendors. However, CTE are not allowed in subqueries. To avoid complicating the translation process, closures are not supported in Ormolu.

There are some other miscellaneous restrictions placed on Alloy expressions in Ormolu. If-else expressions that yield a relation or integer expression are not supported. It is possible to translate these expressions to SQL case expressions, however this would complicate the flattening procedure. The functions or predicates defined in the util/ordering module are also not supported. The reason for this is explained in the next chapter.

expr	::=	number | − number
	|	**Int** | **seq/Int**
quant	::=	**sum**
binOp	::=	<< | >> | <<<
compareOp	::=	< | > | <= | >=
unOp	::=	**seq** | # | * | ∧

Figure 3-2: Restricted Alloy 4 Grammar Elements

3.2 Translation to OIL

OIL is an intermediate language derived from a subset of Alloy that is easily expressible in SQL. Figure 3-3 defines the syntax of OIL. There are two kinds of expressions, relations and formulas. Relations are translated to queries in SQL, while formulas are translated to boolean value expressions. In this section we describe how an Alloy expression is translated to an OIL expression.

As a working example we will consider the body of the `DoubleOccupancies` function presented in Chapter 3. The relevant portion of the model is shown below.

```
pred HasDoubleOccupancy[a1: one Aircraft, a2: one Aircraft] {
  a1 != a2
  (a1 + a2).location in Occupied
  a1.location = a2.location
}

fun DoubleOccupancies[]: Aircraft->Aircraft {
  {a, b: Aircraft | HasDoubleOccupancy[a, b]}
}
```

3.2.1 In-line Process

A let expression introduces a variable x that is bound to an expression e. Whenever the variable x appears in the body, it is replaced by expression e. This is accomplished by utilizing methods provided by the Alloy 4 compiler API to in-line expressions. Function calls are rewritten in a similar manner. The arguments of a function call are bound to

expr	::=	relation | formula
relation	::=	**none** | **iden** | **univ**
	|	name
	|	relation [& | + | −] relation
	|	relation [<: | :> | **.**] relation
	|	relation ++ relation
	|	relation → relation
	|	∼ relation
	|	**{** decl,+ | formula **}**
formula	::=	formula [**and** | **or**] formula
	|	**not** formula
	|	[**some** | **lone** | **one**] relation
	|	formula **=** formula
decl	::=	varName **:** relation
name	::=	sigName | fieldName | varName

Figure 3-3: Grammar of Ormolu's Intermediate Language

the parameters of the function or predicate. The parameter variables are replaced with their respective bound expressions in the function body. The body is then in-lined in the expression.

In our example, the function `DoubleOccupancies` invokes `HasDoubleOccupancy` so in-lining occurs. The variable `a` replaces `a1` and `b` replaces `a2` in the body of `HasDoubleOccupancy`. The body is then in-lined in place of the call `HasDoubleOccupancy[a, b]`

Listing 3.1: Example of In-line Process

```
{a, b: Aircraft {
  a != b
  (a + b).location in Occupied
  a.location = b.location}
}
```

3.2.2 Relation Expressions

The majority of relation expressions have direct analogs in OIL. There are a few exceptions. When an arrow operator is translated to Ormolu, the multiplicity constraints are dropped. Disjointness constraints are also dropped from declarations. Multiplicity and disjointness constraints are interpreted as facts by Ormolu. One of the design decisions of Ormolu is not to translate facts. This is discussed in greater detail in Chapter 6. The constants **none**, **univ**, and **iden** have analogous elements in OIL. The names of signatures and fields are translated to sigName and fieldName elements respectively.

3.2.3 Formula Expressions

Unlike relation expressions, the majority of formula expressions do not have a direct analog in OIL. The exceptions are **not**, **or**, **and**, **some**, **lone**, and **one**. Implication and conditional implication are translated to conjunctive normal form. The subset operator is expressed

using the **some** and set difference operator. The relation `left` is in `right` if and only if every element in `left` is in `right`. This means the set difference of `left` and `right` should be the empty set. Quantified formulas are translated to comprehensions with quantifiers applied.

One area of potential confusion is the equals (=) operator. The equals operator is present in OIL, but it has a different semantic meaning. In Alloy the operator returns true when two relations contain the exact same element and false otherwise, while in OIL it tests if two formulas evaluate to the same value. The equals operator in OIL is really the **iff** operator in Alloy. The equals operator in Alloy is rewritten as a predicate that returns true if the left relation is a subset of the right relation and vice versa. Listing 3.2 depicts the example after the translation to OIL.

Figure 3-4: Translation from Alloy to OIL Formula Expressions

not expr	⇒	**not** expr
left **or** right	⇒	left **or** right
left **and** right	⇒	left **and** right
left **iff** right	⇒	left = right
left **implies** right	⇒	**not** left **or** right
cond **implies** left **else** right	⇒	(cond **and** left) **or** (**not** cond **and** right)
left [**not**]? = right	⇒	[**not**]? (left **in** right **and** right **in** left)
left [**not**]? **in** right	⇒	[**not**]? **not some** left − right
no expr	⇒	**not some** expr
some expr	⇒	**some** expr
lone expr	⇒	**lone** expr
one expr	⇒	**one** expr
all decl,+ | expr	⇒	**no** { decl,+ | **not** expr }
no decl,+ | expr	⇒	**not some** { decl,+ | expr }
some decl,+ | expr	⇒	**some** { decl,+ | expr }
lone decl,+ | expr	⇒	**lone** { decl,+ | expr }
one decl,+ | expr	⇒	**one** { decl,+ | expr }
{ expr* **}**	⇒	expr [**and** expr]*

Listing 3.2: Example of Alloy Translated to OIL

```
{a, b: Aircraft |
  not ( not some a - b and not some b - a )
  and
  not some (a + b).location - Occupied
  and
  ( not some a.location - b.location and
    not some b.location - a.location )
}
```

3.3 Translation to SQL

Once an Alloy expression is translated to OIL, the OIL expression is translated to a SQL expression. As mentioned in the previous section, relation expressions are translated to

query expressions, while formulas are translated to boolean expressions. Translation from OIL to SQL is performed by the function `toSql`. This will convert the OIL expression to a SQL string. The target database is HyperSQL DB, but the translation method is easily expressible in other SQL dialects.

3.3.1 Boolean Expressions

Listing 3.3: Example of Boolean Expressions Translated to SQL

```
{a, b: Aircraft |
  NOT ( NOT EXISTS toSql(a - b) AND NOT EXISTS toSql(b - a))
  AND
  NOT EXISTS toSql((a + b).location - Occupied)
  AND
  ( NOT EXISTS toSql(a.location - b.location) AND
    NOT EXISTS toSql(b.location - a.location) )
}
```

Listing 3.3 is the example after the `toSql` method is applied to boolean expressions. The definition of the `toSql` method for formula expressions is presented in Figure 3-5. The basic structure of the translation is to call `toSql` on non-leaf elements while translating the operator to an equivalent operator in SQL. For instance the **some** operator returns true if and only if the relation is not empty. In SQL this is expressed by using the EXISTS operator. Given a subquery, it returns true if the subquery returns some set of tuples and false otherwise. The translation of **lone** and **one** use a special method `count` that returns the number of tuples in the result set of a query. The details of the `count` method are explained in the next subsection.

`toSql`(left **and** right)	⇒	`toSql`(left) AND `toSql`(right)
`toSql`(left **or** right)	⇒	`toSql`(left) OR `toSql`(right)
`toSql`(**not** formula)	⇒	NOT `toSql`(formula)
`toSql`(**some** relation)	⇒	EXISTS (`toSql`(relation))
`toSql`(**lone** relation)	⇒	`toSql`(`count`(relation)) >= 1
`toSql`(**one** relation)	⇒	`toSql`(`count`(relation)) = 1
`toSql`(left = right)	⇒	`toSql`(left) = `toSql`(right)

Figure 3-5: Definition of toSql for Boolean Expressions

3.3.2 Query Generation

Boolean expressions are translated by recursively calling the `toSql` method on non-leaf elements. A similar approach applied to relation expressions would produce deeply nested queries. The majority of database vendors perform query optimizations that automatically denest or flatten queries. However, several database systems place explicit constraints on subqueries, especially correlated subqueries and will reject queries if correlated values are referenced too deeply in a query. Given these factors it is the goal of the translator to flatten queries when it can do so easily.

Denesting of correlated subqueries is an active area of research with various techniques including rewriting correlated subqueries using various aggregation functions. [2, 13, 16,

Table 3.1: Properties of a SQL Query for Alloy Expression

Property	Description
`Project`(relation)	List of projected columns of a SELECT clause
`Tables`(relation)	List of tables or subqueries that appear in the FROM clause
`Filter`(relation)	List of boolean value expressions that appear in the WHERE clause

17, 18, 19]. Ormolu uses a simpler technique inspired by a SPARQL-to-SQL translator presented in [8]. We model a query as a series of properties. Each relation expression defines the value of these properties. Table 3.1 describes these properties. The `toSQL` method of a relation expression is defined using these properties as

SELECT DISTINCT `Project`(relation)
FROM `Tables`(relation)
WHERE `Filter`(relation)

The properties `Project`, `Tables`, and `Filter` all return list of strings. The lists are similar to lists in Lisp, consisting of an empty list (`Nil`), and a list constructor (::). Table 3.2 defines the properties and methods of list.

Table 3.2: List Properties and Methods

Property	Description
`Nil`	The empty list
head :: tail	Constructs a new list from element head and list tail
`head`(list)	Returns the first element of the list
`last`(list)	Returns the last element of the list
`tail`(list)	Returns a list containing all but the first element of the list
`int`(list)	Returns a list containing all but the last element of the list
`reverse`(list)	Reverses the elements of a list
lst1 ::: lst2	Appends lst2 to the end of lst1

The Count Method

Now that the query model has been described, the `count` method introduced earlier can be specified. When `count` is applied to a relation it overrides the Project property of that relation. The list of strings is replaced with the string `COUNT(*)`. This is an aggregate function that returns the number of tuples in a result set.

Constants

The properties of the constants **none**, **univ**, and **iden** are shown in Figure 3-6. **none** projects a single column from a dummy table. The dummy table is given a correlation name *alias* that is generated uniquely for each expression. The `Filter` property is set to false to ensure the query returns an empty result set. **univ** is defined in a similar way, except

the relation view of **univ** is selected and the `Filter` is set to true. The **iden** constant is derived from the properties of **univ**, and projects the column of **univ** twice.

Project(none)	⇒	*alias*.col1 :: Nil
Tables(none)	⇒	(VALUES(-1)) AS *alias*(col1) :: Nil
Filter(none)	⇒	FALSE :: Nil
Project(univ)	⇒	*alias*.col1 :: Nil
Tables(univ)	⇒	univ.univ AS *alias*(col1)
Filter(univ)	⇒	TRUE :: Nil
Project(iden)	⇒	Project(univ) ::: Project(univ)
Tables(iden)	⇒	Tables(univ)
Filter(iden)	⇒	Filter(univ)

Figure 3-6: Properties of Constants

Names

All three name expressions in OIL result in different translations. For a sigName a single column is projected from the relation view of the referenced signature. For a fieldName all n columns are projected from the relation view of the referenced field where n is the arity of the field. The varName is the correlation name of an outer query. It projects all n columns of the query referenced by the varName and selects from a dummy table.

Project(sigName)	=	*alias*.col1 :: Nil
Tables(sigName)	=	sigName.signame AS *alias*(col1) :: Nil
Filter(sigName)	=	TRUE :: Nil
Project(fieldName)	=	*alias*.col1 :: *alias*.col2 :: ... :: *alias*.col$n$ :: Nil
Tables(fieldName)	=	sigName.fieldName AS *alias*(col1, col2, ..., col$n$) :: Nil
Filter(fieldName)	=	TRUE :: Nil
Project(varName)	=	varName.col1, varName.col2, ..., varName.col$n$ :: Nil
Tables(varName)	=	(VALUES(-1)) :: Nil
Filter(varName)	=	TRUE :: Nil

Figure 3-7: Properties of Name Expressions

Consider the expression (a+b).location - Occupied in the example. The table below shows how the properties would be set by the translator. There are two varNames, `a` and `b`. They are each bound to the Aircraft signature so a single column is projected, while selecting from a dummy table. There is one fieldName, `location`. The location field has an arity of 2, so the alias `rel1` declares two columns. There is one sigName, `Occupied`. It has a different alias assigned to it by the translator than the location field.

Expression	Project	Tables	Filter
a	a.col1 :: Nil	(VALUES(-1)) :: Nil	TRUE :: Nil
b	b.col1 :: Nil	(VALUES(-1)) :: Nil	TRUE :: Nil
location	rel1.col1 :: rel1.col2 :: Nil	Aircraft.location AS rel1(col1, col2) :: Nil	TRUE :: Nil
Occupied	rel2.col1 :: Nil	Occupied.Occupied AS rel2(col1) :: Nil	TRUE :: Nil

Figure 3-8: Properties of Example Name Expressions

Set Operators

The set operators +, −, and & are translated to the SQL operators UNION, EXCEPT, and INTERSECT respectively. All three set operators invoke the `toSql` method on their left and right subexpressions. As a result a nested query is created. If subexpressions contain references to correlation names of an outer query, translation will fail. This occurs in the translation of the example in Listing 3.4. The `toSql` function is invoked in the translation of (a+b).location - Occupied. The `toSql` call contains references to varNames, which are correlation names. There is a chance this query will be rejected, depending on the database engine.

Project($x + y$)	=	$alias$.col1 :: ... :: $alias$.col$n$ :: Nil
Tables($x + y$)	=	(`toSql`(x) `UNION` `toSql`(y)) AS $alias$(col1, ..., col$n$) :: Nil
Filter($x + y$)	=	TRUE :: Nil
Project($x - y$)	=	$alias$.col1 :: ... :: $alias$.col$n$ :: Nil
Tables($x - y$)	=	(`toSql`(x) `EXCEPT` `toSql`(y)) AS $alias$(col1, col2, ..., col$n$) :: Nil
Filter($x - y$)	=	TRUE :: Nil
Project($x\&y$)	=	$alias$.col1 :: ... :: $alias$.col$n$ :: Nil
Tables($x\&y$)	=	(`toSql`(x) `INTERSECT` `toSql`(y)) AS $alias$(col1, col2, ..., col$n$) :: Nil
Filter($x\&y$)	=	TRUE :: Nil

Figure 3-9: Properties of Set Operators

The best way to work around this problem is to distribute operations over set operators if possible. For example, the join operator in the expression `(a+b).location` should be distributed over the union operator and written as `a.location + b.location` instead. The first expression will nest the union inside the join operator when translated, while the second expression will not cause any additional nesting.

Listing 3.4: Example of Set Operators Translated to SQL

```
//a + b
SELECT rel3.col1
FROM (
  SELECT DISTINCT a.col1 FROM (VALUES(-1))
  UNION
  SELECT DISTINCT b.col1 FROM (VALUES(-1))
  ) AS rel3(col1)
WHERE TRUE

//(a+b).location - Occupied
SELECT rel4.col1
FROM (
  toSql( (a+b).location )
  UNION
  SELECT DISTINCT rel2.col1 FROM Occupied.Occupied AS rel2(col1)
  ) AS rel4(col1)
WHERE TRUE
```

Non-Set Operators

The advantages of the model based approach to generating SQL queries is seen in Figure 3-10. The properties of the operators in the table are defined in terms of the properties of other relations. No invocation of the `toSql` method is needed, resulting in a flat query.

$\text{Project}(x \to y)$	=	$\text{Project}(x) ::: \text{Project}(y)$
$\text{Tables}(x \to y)$	=	$\text{Tables}(x) ::: \text{Tables}(y)$
$\text{Filter}(x \to y)$	=	$\text{Filter}(x) ::: \text{Filter}(y)$
$\text{Project}(x.y)$	=	$\text{init}(\text{Project}(x)) ::: \text{tail}(\text{Project}(y))$
$\text{Tables}(x.y)$	=	$\text{Tables}(x) ::: \text{Tables}(y)$
$\text{Filter}(x.y)$	=	$\text{last}(\text{Project}(x)) = \text{head}(\text{Project}(y)) :: \text{Filter}(x) ::: \text{Filter}(y)$
$\text{Project}(x <: y)$	=	$\text{Project}(y)$
$\text{Tables}(x <: y)$	=	$\text{Tables}(x) :: \text{Tables}(y)$
$\text{Filter}(x <: y)$	=	$\text{head}(\text{Project}(x)) = \text{head}(\text{Project}(y)) :: \text{Filter}(x) ::: \text{Filter}(y)$
$\text{Project}(x :> y)$	=	$\text{Project}(x)$
$\text{Tables}(x :> y)$	=	$\text{Tables}(x) ::: \text{Tables}(y)$
$\text{Filter}(x :> y)$	=	$\text{tail}(\text{Project}(x)) = \text{head}(\text{Project}(y)) :: \text{Filter}(x) ::: \text{Filter}(y)$
$\text{Project}(\sim x)$	=	$\text{reverse}(\text{Project}(x))$
$\text{Tables}(\sim x)$	=	$\text{Tables}(x)$
$\text{Filter}(\sim x)$	=	$\text{Filter}(x)$

Figure 3-10: Properties of Arrow, Join, and Transpose Operators

This is seen in the listing below. The first query, depicts the translation of `(a + b).location` if `toSql` was called on the left and right subexpressions of the join operator. The left subexpression — `a + b` — is wrapped in a subquery. However the translator

knows the projected columns, selected tables, and filter of the subexpressions. It can use this information to flatten the query.

For a join expression the tables of both subexpressions are needed, so the Tables lists are concatenated together to form a single list. The filters of the subexpressions must also be applied so the Filter list are concatenated. The last column of the left subexpression must match the first column of the right subexpression, so this constraint is added to the Filter list as well. Finally the matching columns are dropped and the remaining columns are joined together. This means the init, all but the last element of a list, of the left subexpression's Project list is concatenated with the tail, all but the first element of a list, of the right subexpression's Project list.

Listing 3.5: Example Denesting of a Query by the Translator

```
//(a+b).location With Nesting
SELECT rel6.col2
FROM (
  SELECT rel3.col1
  FROM (
    SELECT DISTINCT a.col1 FROM (VALUES(-1))
    UNION
    SELECT DISTINCT b.col1 FROM (VALUES(-1))
    ) AS rel3(col1)
  WHERE TRUE
  ) AS rel5(col1),
  Aircraft.location AS rel6(col1, col2)
WHERE
  rel5.col1 = rel6.col1 AND TRUE AND TRUE

//(a+b).location Without Nesting
SELECT rel5.col2
FROM (
  SELECT DISTINCT a.col1 FROM (VALUES(-1))
  UNION
  SELECT DISTINCT b.col1 FROM (VALUES(-1))
  ) AS rel3(col1),
  Aircraft.location AS rel5(col1, col2)
WHERE
  rel3.col1 = rel5.col1
```

The same reasoning is applies to the remaining operators. For instance, the transpose operator reverses the columns of a binary relation. It leaves the Tables and Filter properties unchanged in the subexpression, but reverses the Project list of the subexpression. For domain and range restriction, the right or left subexpression is projected, but an additional filter is applied requiring the first or last column of the relation to match the first column of the other subexpression. In order to apply this filter, the tables of both subexpressions are needed, so the Tables list are concatenated. The arrow operator simply concatenates the Project list of the two subexpressions together, while doing the same for Tables and Filters.

Override

The override operator creates a helper relation that selects all the tuples from the left relation that do not match tuples in the right relation. The union of the helper relation and

Project(x ++ y)	=	Project((x help y) + y)
Tables(x ++ y)	=	Tables((x help y) + y)
Filter(x ++ y)	=	Filter((x help y) + y)
Project(x help y)	=	Project(x)
Tables(x help y)	=	Tables(x) ::: Tables(y)
Filter(x help y)	=	head(Project)(x) <> head(Projection(x)) :: Filter(x) ::: Filter(y)

Figure 3-11: Properties of Override

the right relation is used to define the properties of the operator. Since a union operator is used, the override operator will produce a nested query.

Comprehensions

A comprehension consists of a series of declaration statements followed by a formula. Each declaration consists of a variable name and a relation. The relation is converted to a subquery. The variable name is used as the correlation name of the subquery. The formula is used as the filter. Figure 3-12 defines the properties for a comprehension with a single declaration, but it is easily generalized to multiple declarations.

Project(**{** var:rel | form **}**)	=	var.col1 :: ... :: var.col$n$ :: Nil
Tables(**{** var:rel | form **}**)	=	`toSql`(rel) AS var(col1, ..., col$n$) :: Nil
Filter(**{** var:rel | form **}**)	=	form :: Nil

Figure 3-12: Properties of Comprehension

The listing below is the complete translation of the body of the `DoubleOccupancies` function. The body contains a single comprehension statement. The Aircraft table is chosen twice in the from clause. The tables are given correlation names that map to the variable names in the declaration. This "binds" the Aircraft signature to the variables. The formula is translated to SQL and placed in the where clause. Finally it projects the columns of `a` and `b` in the select clause.

Listing 3.6: Complete Translation of Example to SQL

```
SELECT DISTINCT a.col1, b.col1
FROM
  Aircraft.Aircraft AS a(col1),  Aircraft.Aircraft AS b(col1)
WHERE
  NOT (
    NOT EXISTS (
      SELECT DISTINCT a.col1 FROM (VALUES(-1))
      EXCEPT
      SELECT DISTINCT b.col1 FROM (VALUES(-1))
      )
    AND
    NOT EXISTS (
      SELECT DISTINCT b.col1 FROM (VALUES(-1))
      EXCEPT
      SELECT DISTINCT a.col1 FROM (VALUES(-1))
      )
    )
  AND
  NOT EXISTS (
    SELECT rel2.col2
    FROM
      (
      SELECT DISTINCT a.col1 FROM (VALUES(-1))
      UNION
      SELECT DISTINCT b.col1 FROM (VALUES(-1))
      ) AS rel1(col1),
      Aircraft.location AS rel2(col1, col2)
    WHERE
      rel1.col1 = rel2.col1
    EXCEPT
    SELECT DISTINCT rel3.col1 FROM Occupied.Occupied AS rel3(col1)
    )
  AND
  (
  NOT EXISTS (
    SELECT DISTINCT rel4.col2
    FROM (VALUES(-1)), Aircraft.location AS rel4(col1, col2)
    WHERE a.col1 = rel4.col1
    EXCEPT
    SELECT DISTINCT rel5.col2
    FROM (VALUES(-1)), Aircraft.location AS rel5(col1, col2)
    WHERE b.col1 = rel5.col1
    )
  AND
  NOT EXISTS (
    SELECT DISTINCT rel4.col2
    FROM (VALUES(-1)), Aircraft.location AS rel6(col1, col2)
    WHERE a.col1 = rel6.col1
    EXCEPT
    SELECT DISTINCT rel5.col2
    FROM (VALUES(-1)), Aircraft.location AS rel7(col1, col2)
    WHERE b.col1 = rel7.col1 )  )
```

Chapter 4

Time and State in Ormolu

This chapter presents the state library provided by Ormolu. We begin by discussing how time and state are modeled in Alloy and the difficulties of translating this to SQL. This is followed by the definition of the state library. We conclude by demonstrating how the state library is used in an example.

4.1 Time and Ordering

The natural way of modeling a system is describing how it evolves over time. Time is so important in describing a system that many modeling languages add some notion of time or state to their logic. Alloy is not one of those languages. There is no notion of time or state in Alloy. Excluding time keeps the logic of Alloy simple and flexible, allowed several idioms of time to be supported.

Many of these idioms rely on the ordering library provided by Alloy to model stateful systems. Consider the challenge of specifying an aircraft taking off. There are several ways of modeling this. Listing 4.3 depicts a very simple model. There are two locations `Ground` and `Air`. There is also a `Time` signature that is ordered and an `Aircraft` signature. There is a single `location` field that associates an aircraft with a location at every point in time. Taking off is then specified as changing location from the ground to the air.

Listing 4.1: Takeoff Event Modeled Using Explicit Notion of Time

```
open util/ordering[Time]
sig Time{}
abstract sig Location{}
one sig Ground, Air extends Location{}
sig Aircraft {
  location: Location one -> Time
}

pred TakeOff[a: Aircraft, t: Time] {
  let before = t.prev, after = t {
    before[a.location] = Ground
    after[a.location] = Air
  }
}
```

Despite how simple this model is, the Ormolu translator will reject it. This is because

the ordering library is used and as stated in the previous chapter, the ordering library is not supported. Instead Ormolu provides a state library that supports a similar modeling idiom but without exposing the ordering library directly. The natural ordering of Ormolu instances is exploited to provide a simple method for translating the state library to SQL.

4.2 Instances in Alloy and Ormolu

An Alloy model defines a set of possible instances. An *instance* of an Alloy model is an assignment of values to its variables. Given a constraint and a scope for the variables of the model, the Alloy analyzer will find an instance that satisfies the constraint. An Alloy instance is created automatically by the Alloy Analyzer when executing a command. There is no syntax for specifying an instance in Alloy. An Alloy model defines a set of instances through the constraints placed on a model. An instance is immutable. Atoms cannot be added, deleted, or rearranged in an instance.

This differs greatly with instances in Ormolu. An Ormolu instance is derived from observations of the running system. These observations are supplied by the user and the system has no knowledge of what the final set of values are for a signature or field. Observations describe changes or modifications to the current instance, and do not describe a complete new instance. When a new observation is received the current instance is overwritten and tuples are inserted into or deleted from the analysis database. An Ormolu instance is dynamic and unbounded with unpredictable inputs.

4.3 The State Library

The state library is a series of modules that are used to model stateful fields in Alloy. A different module is used for each degree of field. Listing 4.2 is the unary state module. It accepts a Value and Version as parameters. The binary state module accepts Value1, Value2 and Version as parameters, and so on for higher arities[1].

State is modeled as a series of versions. The `state` field is a binary relation mapping a set of values to a version. There is a maximum version, `maxVer`. The quantified formula in the appended fact forces a transition between versions of states. It states that for all versions less than or equal to the maximum version, the set of values does not equal the set of values of the previous version. For instance, if the first version of a state is empty, the next version cannot be empty as well. It must map to a different set of values. If the version is greater than the maximum version it has the same set of values as the maximum version.

It is important to note that the user of the library does not have access to the state field directly because it is marked private. Instead views over the versions of state are exposed. There are three views provided: current, previous and past values. The current view is the set of values associated with the maximum, or current, version. The previous view is the set of values associated with the previous version before the maximum version. The past view is the union of the set of values of all previous versions of state.

These views of states are easy to update as observations are received. An observation updates the current view if and only if it adds or removes values from that view. If it does

[1]Currently only the UnaryState and BinaryState modules are created because higher arities are rarely needed.

Listing 4.2: Unary State Library

```
module state[Value, Version]
open util/ordering[Version]

sig UnaryState {
  private state: Value set -> Version,
  private maxVer: one Version,

  current: set Value,
  previous: set Value,
  past: set Value,
}{
    all version: Version - first |
      lte[version, maxVer] =>
        version[state] != version.prev[state]
      else
        version[state] = maxVer[state]

    current = maxVer[state]
    previous = maxVer.prev[state]
    past = maxVer.prevs[state]
}

sig OneUnaryState in UnaryState{}{ state in Value one-> Version }
sig LoneUnaryState in UnaryState{}{ state in Value lone-> Version }
sig SomeUnaryState in UnaryState{}{ state in Value some-> Version }
```

cause an update the old set of values is mapped to the previous view. Finally the union of the past values and the old set of values is taken and set as the past view.

The OneUnaryState, LoneUnaryState, and SomeUnaryState signatures define multiplicity constraints over the state field. For example OneUnaryState adds an appended fact that says the state field must have exactly one value for each version. The same is done for LoneUnaryState and SomeUnaryState.

Using the state library is very simple. To have a stateful view over a signature A, the user opens the state/unary module passing in A and another signature to serve as the version[2]. The UnaryState signature with the correct multiplicity is used in declaration for fields.

Listing 4.3 demonstrates how the original takeoff specification is expressed using the state library. A stateful view is opened over the Location signature. The unary state replaces the Location→Time bound expression for the location field. The takeoff predicate does not pass in a time, only a single aircraft. The previous value of the location state is then constrained to be on the ground and the current value in the air.

Looking at the use of the state library, a user may believe it is possible to traverse the versions of a state and write statements such as `previous[previous[a.location]]` or `next[previous[a.location]]`. These are not valid statements and will lead to type errors

[2]The reason a Version signature is passed into the module is to prevent multiple orderings from being created. A large number of orderings can slow down analysis of a model. If multiple stateful fields are opened, the same Version signature can be passed in for each, resulting in only a single ordering being defined over the Version.

Listing 4.3: Takeoff Event Modeled Using Explicit Notion of Time

```
open state/unary[Location, Version] as location

abstract sig Location{}
one sig Ground extends Location{}
one sig Air extends Location{}
private sig Version{}
sig Aircraft {
  location: location/OneUnaryState
}

pred TakeOff[a: Aircraft] {
  previous[a.location] = Ground
  current[a.location] = Air
}
```

or warnings when executed. Exposed fields in the state library are views and return only the set of values associated with a specific version or set of versions. The actual versioning is hidden from the user so traversal between versions is not possible.

4.4 Translation of Stateful Fields

The state library is designed to eliminate the need for directly using the ordering library in some cases. The state library adds a level of indirection and standardizes the use of the ordering library. When an ordering is opened on a signature, a special constraint is placed on the signature to enforce the ordering. Listing 4.4 contains the definition of the main component of the ordering library provided by Alloy, the Ord signature. The First field is the first atom in the ordering, and the Next field contains the mapping from one atom to another in the ordering. The actual ordering constraint is handled by the totalOrder predicate. This predicate is treated specially by the Alloy compiler and utilizes symmetry breaking to implement ordering efficiently.

Listing 4.4: Ord in util/ordering

```
private one sig Ord {
   First: set elem,
   Next: elem -> elem
} {
   pred/totalOrder[elem,First,Next]
}
```

To translate the ordering constraint to SQL, two things must be supplied. An atom must serve as the first atom, and the next mapping must be supplied. The reason why the ordering library is not supported is that the value of the first atom and the next mapping cannot be determined in general.

One case which the ordering can be determined automatically is the order of Ormolu instances. Instances build upon each other and evolve as observations are made. For example, imagine a single aircraft whose current location is Ground. An observation updates the current location to be the Air. The ordering is now clear that the location's first value

was Ground and its successor is Air. If another observation states the aircraft's location is Air, the current location is not updated, and the previous location is still Ground. Finally the aircraft will land and the location will change to Ground. The final ordering of the aircraft location is Ground then Air then Ground.

The sequence of states for the location field could be arbitrarily long. In the case of TFDM, only the end of this ordering is needed for modeling the system. Because of this the state library limits how far back into the ordering of states the model can reference. The three views are presented in Table 4.1 with the corresponding values for the example.

Table 4.1: Views over the State Sequence

View	Description	Example
current	The current (last) value of the stateful field	Ground
previous	The previous value of the stateful field	Air
past	The union of all previous values of the stateful field	Ground, Air

The key insight is that Ormolu knows how to maintain this relationship between the views. The signatures and fields specified in the state library act exactly the same during analysis. They are only treated specially when the field is updated. This is discussed in greater detail in the following chapter.

Chapter 5

Architecture of Ormolu

In this chapter we describe the architecture of Ormolu. We begin by discussing the analysis database that contains the structure of the model and stores the actual data supplied to the system. This is followed by discussing the observer that listens for observation from the system being monitored. The updater is then described that translates the observations received from the observer to updates to the analysis database. Finally we present the analyzer that executes queries over the analysis database and outputs the results of analysis.

5.1 Analysis Database

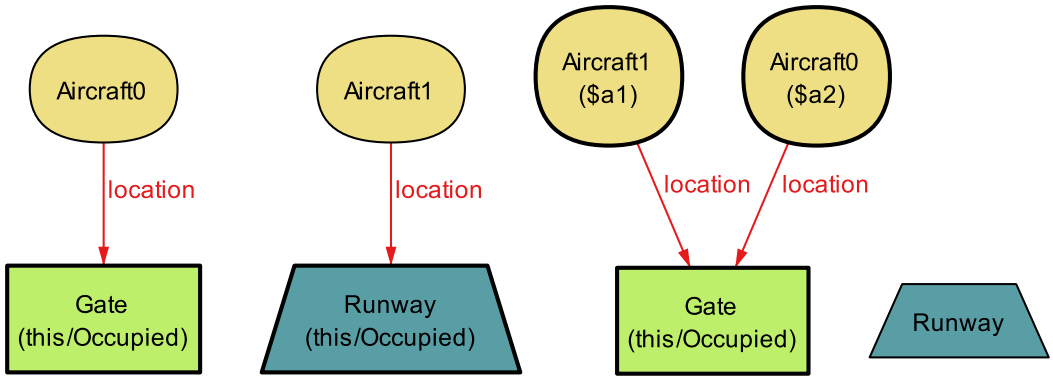
The schema of the analysis database is generated during startup of Ormolu. The Alloy source file is passed through the Alloy 4 compiler. The compiler type checks the source file to ensure there are no errors in the model. It also generates an abstract syntax tree. The abstract syntax tree is passed to the Ormolu translator. The translator creates the necessary schemas, tables, and views as specified in Chapter 2.

The HyperSQL Database (HSQLDB) is the relational database engine used in Ormolu. HSQLDB is written in Java and has several attractive features. The most important of these features is the ability to run in memory-only mode. By running in memory-only mode, expensive writes and reads from disk are avoided, vastly improving the performance of updates to the analysis database and query speed. If performance is not the primary concern, HSQLDB also has a file mode to decrease memory usage at the cost of query time.

5.2 Observer

The analysis database initially contains no tuples. Tuples are added as observations are made. Figure 5-1 depicts two instances. There are two aircrafts, Aircraft0 and Aircraft1, and two locations, a Gate and a Runway. The current instance has Aircraft0 located at the Gate, while Aircraft1 is on the Runway. This instance transitions to a new instance where Aircraft1 moves from the Runway to the Gate. An observation is the description of this transition.

Observations are sent by the running system to Ormolu as JSON strings. JavaScript Object Notation (JSON) is a popular format used to transmit semi-structured data across networks. There are two structures in JSON, objects and arrays. Objects are key-value pairs while arrays are list of elements. JSON is simple, lightweight, and easy for humans to



(a) Current Instance (b) New Instance

Figure 5-1: Instances of the Aircraft Location Model

read and understand. The observation that describes the transition of Figure 5-1 is depicted in Lisingt 5.1. This section specifies the JSON structure of an Ormolu observation.

Listing 5.1: Observation Describing Transition from Current to New Instance

```
{
  ``sigs'':[{
    ``name'' : ``Aircraft'',
    ``type'' : ``Aircraft'',
    ``id'' : ``Aircraft1'',
    ``remove'' : false,
    ``fields'' : [{
      ``name'' : ``location'',
      ``arity'' : 1,
      ``replace'' : [{
          ``value'' : [{
              ``name'' : ``Gate''
              ``id'' : ``Gate''}]
              }]
          }]
}
```

5.2.1 Sig Object

An observation has a single key *sigs*. The sigs key maps to an array of Sig objects. The name key of the Sig object is the name of the signature in the model that is being updated. The type key is the basic type of the signature atom. The id key is the external identifier for an atom. It uniquely identifies an instance of the signature. The remove key is a boolean indicating whether the signature is being inserted or removed from the analysis database. The remove key is optional, and if it is not included it defaults to false. The fields key maps to an array of Field objects. The example in Listing 5.1 contains a single Sig object. It states that the atom Aircraft1 is a member of the signature Aircraft with the type Aircraft.

Listing 5.2: Observation and Sig JSON Specification

```
``Observation'':{
  ``sigs'':[Sig]
}

``Sig'': {
  ``name'': string,
  ``type'': string,
  ``id'': string,
  ``remove'': boolean,
  ``fields'':[Field]
}
```

5.2.2 Field Object

A Field object specifies an update to a field of a signature. The name key maps to the name of the field to update in a signature. The arity key is an integer that denotes the arity of the field's bound expression. If the field is bound to a unary relation, the arity key should be one. If it is bound to a binary field the arity should be two and so on. The add, remove, and replace keys are all optional. Each maps to an array of Value objects. The add key denotes tuples that are inserted to the field, while remove denotes tuples that should be deleted from the field. If the replace key is specified the add and remove field are ignored. All tuples currently belonging to the field are deleted and the tuples specified are inserted. The example in Listing 5.1 contains a single Field object. It states that the relation Aircraft1.location should be replaced with the contents of the replace key and all other tuples in Aircraft1.location should be removed.

Listing 5.3: Field JSON Specification

```
``Field'':{
  ``name'': string,
  ``arity'': integer,
  ``add'': [Value],
  ``remove'': [Value],
  ``replace'':[Value]
}
```

5.2.3 Value Object

A Value object specifies a tuple. It consist of an array of SigRef objects. The number of SigRef objects in a Value object must equal the arity of the enclosing field. For instance the arity of Aircraft1.location is one, so the Value object should contain only a single SigRef. The name, type, and id keys are analogous to the name, type, and id keys of a Sig object. The example in Listing 5.1 contains a single Value object. It states that the atom Gate in signature Gate should replace the current value of the Aircraft1.location field.

Listing 5.4: Value JSON Specification

```
``Value'':{
  ``value'': [SigRef]
}

``SigRef'':{
  ``name'': string,
  ``type'': string,
  ``id'': string
}
```

5.3 Updater

When Ormolu receives an observation, it parses the JSON string and updates the current instance. Updating the analysis database is a non-trivial task. Recall in Chapter 2 that foreign key constraints are utilized to simulate Alloy's type system. Foreign key constraints are an example of integrity constraints on a database. If an update will result in a violation in an integrity constraint, the database manager will reject the operation and the update will fail. Ormolu generates a number of update procedures for every table to prevent integrity constraints from being violated. We explain the concept behind each of the procedures in this section.

5.3.1 Signature Insertion Procedure

Each signature has one or more insertion procedures associated with it. The name of the insertion procedure is of the form *sig*.add, where *sig* is the name of the signature. Since subset signatures can have multiple type tables, they can potentially have multiple insertion procedures as well. To distinguish the insertion procedures of a subset signature apart, the type is appended to the end of the name. This produces a name of the form *subsig*.add_*type*, where *subsig* is the name of the subset signature and *type* is the type table the atom is added to.

A signature insertion procedure accepts as input an atom's string representation and returns an atom's integer representation. Recall an atom's string representation is the external identifier of an atom, and the integer representation is the internal identifier. The procedure will insert the atom into the appropriate type table if it is not a member. If the atom is already a member of the signature, no insertion occurs, but the internal identifier is still returned. The insertion procedure is an idempotent operation.

Inserting into a signature's type table is a non-trivial task. In order to successfully insert a value into a signature, several other insertions are often necessary. Consider the Ormolu instance in Figure 5-2. Before a value is inserted into the Occupied.ofType_Runway table, it must be added to the Runway table. Before it can be added to the Runway table, it must be added to the Location table, and before the Location table the univ table.

Each insertion procedure invokes its parent's add procedure before inserting into its respective type table. Figure 5-3 illustrates this process. A series of invocations is made until the top signature, univ, is reached. A new internal identifier is generated and passed down the chain. Each of the respective signatures will add the internal identifier to its type table before returning the identifier to the child signature. This ensures a foreign key constraint is never violated and the insertions are performed in the correct order.

univ.type	
col1	atom
0	Aircraft0
1	Aircraft1
2	Gate
3	Runway

Aircraft.type
col1
0
1

Gate.type
col1
2

Occupied.ofType_Gate
col1
2

Location.type
col1
2
3

Runway.type
col1
3

Occupied.ofType_Runway
col1
3

Aircraft.location_Location	
col1	col2
0	2
1	3

Figure 5-2: Instance of Aircraft Location Model in Ormolu

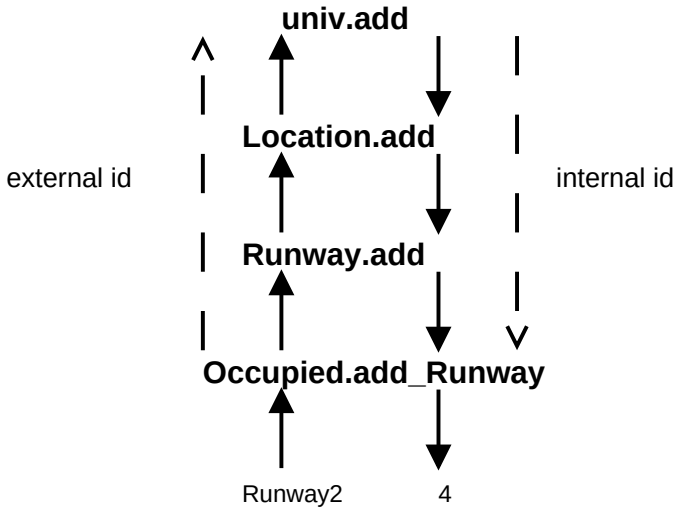


Figure 5-3: When Occupied.add_Runway is invoked, the external identifier Runway2 is passed up the invocation chain until the call to univ.add is made. The univ.add procedure will add Runway2 to its type table and generate the internal identifier 4 for the atom. The internal identifier is passed down the invocation chain and inserted into the respective type tables, ultimately being returned by the original call to Occupied.add_Runway.

5.3.2 Signature Deletion Procedure

The deletion procedure of a signature is similar to the insertion procedure. It's name is the same with del swapped for add. The deletion procedure accepts as input the internal identifier and removes it from the given signature. The chain of invocations for deletion occurs in the opposite direction. When an atom is removed from a signature it is removed from all children signatures *and* fields that reference the atom.

Several databases support a `ON DELETE CASCADE` trigger defined on a foreign key constraint. This instructs the database manager to remove the foreign key from the table when the key is removed from the referenced table. For example, if the atom Gate is removed from the univ type table, the deletion will cascade down to the Location, then Occupied and finally the Aircraft.location table as well, removing the internal identifier 2 from all tables in the analysis database.

5.3.3 Field Update Procedures

The update procedure for a field is more intricate than that of a signature. Each type table of a field defines an insertion and deletion procedures of the form $sig.field$_add_$type$ and $sig.field$_del_$type$ respectively, where sig is the name of the enclosing signature, $field$ is the name of the field, and $type$ is the type of the bounding expression of the field. Both procedures receive as input n internal identifiers. The insertion and deletion procedure of a field does not immediately make changes to the type table. The insertion and deletions are logged in temporary tables.

To push the update to the type tables, a call to the commit procedure of a field is required. The commit procedure takes as input a boolean value. If it is true, the update is pushed to the field as a replace operation. There are several steps to a replace operation.

1. Generate a list of atoms whose field is being updated. This is done by selecting col1 from the temporary tables.

2. Select the current tuples in the field of the updated atoms and add them to the temporary deletion table.

3. Remove from the type tables any tuple that appears in the temporary deletion table.

4. Insert into the type tables any tuple that appears in the temporary insertion table.

A replace operation essentially clears the field of any atom that is being updated and replaces it with the contents of the temporary insertion table. If false is passed to the commit procedure the first two steps of the replace operation are skipped. Only tuples Ormolu is explicitly instructed to delete are removed from the type table.

5.3.4 Updating Stateful Fields

Updating stateful fields is exactly the same as updating fields without state. The only difference is that the commit procedure is modified. If the field of an atom is being updated, the current view of that field is copied over to the previous view and added to the past view. After this is done, updating precedes just like any other field on the current view. This shuffling of values from current to previous and past is the only overhead required for using the state library.

5.4 Analyzer

In Alloy analysis is performed by writing `run` and `check` commands. Recall the model of aircraft location portrayed in Listing 2.1. There are two commands. The constraint states there are no two aircrafts that share the same occupied location. While both commands have the same body, when executed they will produce different instances. A `run` command will search for an instance that satisfies the constraint, while the `check` command searches for a counterexample that violates the constraint.

Ormolu uses analysis functions to perform analysis. These functions are run when ever an Ormolu instance is updated. Analysis functions are special functions in an Alloy model. There are three conditions that a function must adhere to in order for it to be considered an analysis function.

1. The function is not marked `private`
2. The function does not take any arguments
3. The function must appear in the main module of the Alloy source file

These requirements are somewhat arbitrary, but decreases the likelihood that an analysis function is declared by mistake.

Listing 2.1 defines a single analysis function `DoubleOccupancies`. It is a function that returns all pairs of aircrafts in a instance that occupy the same gate or runway. The analysis function is translated to a SQL query. This query is run whenever the Ormolu instance is updated. The execution strategy for analysis could be improved by only running an analysis function if one of the relations it uses is updated.

Chapter 6

Conclusion

Ormolu is a unique approach to runtime monitoring. It uses an Alloy model to generate an embedded database that is analyzed using queries. We conclude by presenting a limited evaluation of Ormolu on a real dataset. This is followed by exploring the design decisions and limitations of this unique approach. We end with a brief discussion of future works for the project.

6.1 Evaluation

To test the feasibility of Ormolu in practice, we monitored a simulation of the TFDM system. The simulation consist of almost four hours of actual data recorded from a local Massachusetts airport with TFDM installed in a tower. The data is a dump from a database between 11am and 2pm on January 4th, 2011. The database was ordered by time and transformed to a single file containing a sequence of observations encoded in JSON. In total there were over 23000 observations stored in a 4MB text file.

Listing 6.1 is the relevant portion of the TFDM Alloy model used in the test. It looks for examples of bad arrivals to the airport. There are four main signatures in the model, Location, Intent, Aircraft and TFDM. Location models the location of an aircraft, and Intent is an enumertaion that describes the general action an aircraft is taking. Aircraft contains two stateful fields for, location and intent. TFDM is the model of the information known by the TFDM system and the flightData field is the set of aircrafts that it knows about.

CreateArrivalFlightData determines if the flight data of an arrival was processed correctly. It considers only aircrafts that have been recently added to the flight data (not in the previous, but now in the current flight data). Inside the negation it describes the condition for a correct arrival. The aircraft's previous location was outside the air space of the airport and the intent of the aircraft is to arrive at the airport or simply to fly over it. By negating this condition, it finds all cases that don't meet this specification and reports these aircraft.

The test was run on a laptop running Windows 7 with 4GB of RAM and a 2GHz dual core processor. The data was read from the JSON text file and feed to Ormolu. After each observation the analyzer ran the analysis query for a total of 23052 times. The whole analysis was completed in less than 10 minutes.

While the test was not thorough, some conclusions can be made. The average execution time for each query was 26ms. The rate at which observations were made was roughly two

Listing 6.1: Relevant Components of the TFDM Alloy Model

```
open state/unary[Aircraft, Version] as aircraft
open state/unary[Location, Version] as location
open state/unary[Intent, Version] as intent
private sig Version{}

abstract sig Location {}
abstract sig Airborne, Grounded extends Location {}
sig OutsideTowerSpace, InsideTowerSpace extends Airborne {}
abstract sig ActiveRegion, SurfaceDestination extends Grounded {}
sig Gate extends SurfaceDestination {}
sig Hangar extends SurfaceDestination {}
sig Taxiway, Ramp, HoldingPen extends ActiveRegion {}
sig Runway extends ActiveRegion {}

abstract sig Intent {}
one sig Arrive, Depart, Overflight, TransitionTaxi extends Intent {}

abstract sig Object {}
sig Aircraft extends Object {
                location: one location/OneUnaryState,
                intent: one intent/OneUnaryState,
}
one sig TFDM extends Object {
                flightData: one aircraft/UnaryState,
}

fun BadArrival[]: Aircraft {
  {aircraft: Aircraft | CreateArrivalFlightData[aircraft]}
}

pred CreateArrivalFlightData [aircraft: Aircraft] {
  aircraft not in previous[TFDM.flightData]
  aircraft in current[TFDM.flightData]
  !{
    previous[aircraft.location] in OutsideTowerSpace
    previous[aircraft.intent] in Arrive + Overflight
      }
}
```

observations per second. If the observation rate is accurate then query time would need to be below 500ms for Ormolu to process observations fast enough to prevent observations from being dropped. 26ms falls an order of magnitude below this mark implying that Ormolu is feasible given a slow enough observation rate.

One informal observation is the query time increased as more observations were made. This is an expected result. A more thorough evaluation is necessary to determine how quickly query performance degrades. We favored simplicity over performance in the translator, so it is possible more optimized queries can be produced and indexes defined that can speed up performance even more. Another possible solution for dealing with degrading performance is to purge the analysis database of stale data periodically.

6.2 Design Decisions and Limitations

Ormolu is able to replicate the semantics of Alloy to some degree. There are cases where the translation falls short. Many of these supposed limitations are a result of the design decision for the translator to ignore facts. This decision was made because facts are interpreted as constraints that are inherently true in the system. If the constraint of a fact needs to be checked by Ormolu it should be written as a predicate and invoked when needed.

Listing 6.2: Explicit Mutual Disjointness

```
sig A{}
sig B, C, D extends A{}

predicate bcdDisjoint[]{
  no B & C
  no B & D
  no C & D
}
```

Constraint paragraphs marked explicitly with the `fact` keyword are not the only constraints Ormolu considers facts. There are several cases where a fact or constraint is implied in Alloy. One example is for extension signatures. By definition extension signatures are mutually disjoint and cannot contain the same atoms. The tables of extension signatures do not prevent this from occurring. If the user wishes to enforce this constraint in Ormolu then the mutual disjointness must be written out explicitly. Listing 6.2 illustrates one way of expressing mutual disjointness explicitly. The predicate `bcdDisjoint` states there is no intersection between any pair of extensions to `A`. In general, a signature with n extensions would have $\binom{n}{2}$ possible pairs to write out explicitly.

Whether a constraint is explicitly marked as a fact or not, the Ormolu translator will ignore it. All constraints must be explicit and appear in a predicate or function. Here are known cases where this approach should be utilized if the constraints of a fact are to be checked by Ormolu. This is not an exhaustive list and the Alloy specification should be read to determine what constraints are implied in a model.

- Set multiplicity constraints, especially in field declaration
- Relation multiplicity constraints, especially in field declaration
- Disjoint or partitioned declarations

- Appended facts
- Disjointness of extension signatures
- Declaration constraints in fields
- Abstract signatures having no elements except those belonging to its extensions

Rewriting these constraints explicitly is a rather straightforward task. A future version of the Ormolu translator could perform the rewriting automatically for the user if desired. Since the user may or may not want to check these facts, possibly for performance reasons, the rewriting should be configurable.

6.3 Future Work

Ormolu generates a runtime monitor from an Alloy source file. The current implementation shows that the approach is feasible. There are several areas where Ormolu might be improved and areas of future research.

Full Support of Alloy Logic The Ormolu translator supports the most commonly used elements of Alloy's logic. There are also some restrictions placed on the model. There are a handful of operators that are not supported. The translator should support the complete logic of Alloy. One possible way of supporting operators that do not map well to database operators is creating user-defined functions that implement the operation. If the database allows user-defined functions to be specified in a language such as Java, procedural code can be used to implement the operations. For instance closures, which have limited support through Common Table Expressions, could be implemented as a function in Java that constructs the proper result set.

Optimization of Analysis Database Efficiency was not one of the design goals of Ormolu. Since the queries that will run are known beforehand there is great potential for optimizing the analysis database for these queries. This can include specializing the translation of signatures and fields to use fewer or more tables, automatically creating indices over commonly referenced columns, executing analysis functions in parallel, etc. Before optimizing Ormolu, a more thorough evaluation of its performance and detailed profiling to identify the bottle necks of the system is needed. Since the current performance with no optimization is feasible, further refinement in this area will only increase the attractiveness of this approach.

Time Library The state library is great for constraining how a single field changes over time. However there is no way of expressing the relationship between changes in different fields. For instance, after an aircraft takes off it should then lift its landing gear. The ordering of events is important, after the ground to air transition, the landing gear should be lifted, but not before. If the landing gears are retracted before take off that can indicate a serious problem occurred. There is no way of sequencing these events currently in Ormolu. A specially recognized timing library could be implemented in a similar way to the state library to support this use case. The other alternative is to discover a way to support the ordering library so these specialized libraries would not be necessary.

Bibliography

[1] M. Auguston. Software architecture built from behavior models. *ACM SIGSOFT Software Engineering Notes*, 34(5):1–15, 2009.

[2] F. Bry. Logical rewritings for improving the evaluation of quantified queries. *MFDBS 89*, pages 100–116, 1989.

[3] E. Clarke. Model checking. In *Foundations of software technology and theoretical computer science*, pages 54–56. Springer, 1997.

[4] E.F. Codd. Derivability, redundancy, and consistency of relations stored in large data banks. *IBM Research Report RJ*, 599, 1969.

[5] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[6] C. de la Riva, M.J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for sql queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 67–74. ACM, 2010.

[7] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30(12):859–872, 2004.

[8] B. Elliott, E. Cheng, C. Thomas-Ogbuji, and Z.M. Ozsoyoglu. A complete translation from sparql into efficient sql. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 31–42. ACM, 2009.

[9] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[10] D. Jackson. *Software Abstractions: logic, language and analysis*. The MIT Press, 2006.

[11] D. Jackson and C.A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *Software Engineering, IEEE Transactions on*, 22(7):484–495, 1996.

[12] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[13] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 239–250. IEEE, 2004.

[14] J.P. Near, M. Hermenegildo, and T. Schaub. From relational specifications to logic programs. In *Technical Communications of the 26th International Conference on Logic Programming*, volume 7, pages 144–153. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[15] B. Plattner and J. Nievergelt. Monitoring program execution: A survey. *Computer*, 14(1):76–93, 1981.

[16] J. Rao and K.A. Ross. Reusing invariants: A new strategy for correlated queries. In *ACM SIGMOD Record*, volume 27, pages 37–48. ACM, 1998.

[17] P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Complex query decorrelation. In *icde*, page 450. Published by the IEEE Computer Society, 1996.

[18] H. Steenhagen, P. Apers, and H. Blanken. Optimization of nested queries in a complex object model. *Advances in Database TechnologyEDBT'94*, pages 337–350, 1994.

[19] C. Zuzarte, H. Pirahesh, W. Ma, Q. Cheng, L. Liu, and K. Wong. Winmagic: subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 652–656. ACM, 2003.