Constraint Programming Solutions to LogicQL Program Verification and System Resilience Problems

by

Heng Liu

B.Sc. Honours, St. Francis Xavier University, 2013

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

Approval

Name:	**Heng Liu**
Degree:	**Master of Science (Science)**
Title:	***Constraint Programming Solutions to LogicQL Program Verification and System Resilience Problems***
Examining Committee:	**Chair:** Dr. Andrei Bulatov Associate Professor

Dr. Eugenia Ternovska
Senior Supervisor
Associate Professor

Dr. David Mitchell
Supervisor
Associate Professor

Dr. Arvind Gupta
Examiner
Professor
UBC

Date Defended: 14 December 2015

Abstract

Constraint Programming (CP) is the study of solving problems by stating constraints on the solution to be found. Many computer science problems can be viewed as a special case of the constraint problem. In this thesis, we focus on using CP solvers for solving two distinct problems. First, we develop and implement a framework for an automatic generation of models that satisfy a program in LogicQL, a high-level database query language. We show that our system gives immediate feedback to the user and can be used for incremental development of LogicQL programs. Second, we consider a system resilience problem that is important in many application domains. We present the design and implementation of SR-solver, a novel integrated tool for evaluating system resilience. The SR-solver supports a graphical representation of the system, thus making the evaluation of system resilience accessible to general users.

Keywords: Constraint Programming; LogicQL; Program Verification; System Resilience

Acknowledgements

I cannot express enough thanks to my research supervisors for their continued support and encouragement: Dr. Eugenia Ternovska and Dr. David Mitchell. I offer my sincere appreciation for useful critiques of this research work offered by Dr. Arvind Gupta.

I am also indebted to the members of the INOUE Laboratory at National Institute of Informatics with whom I have interacted during the course of my NII internship. Particularly, I would like to acknowledge Dr. Katsumi Inoue, Dr. Nicolas Schwind, Dr. Morgan Magnin, Dr. Tony Ribeiro, Maxime Clement and Jacopo Panerati for the many valuable discussions that helped me understand the system resilience problem better.

Finally, I wish to thank my parents, family, and friends for their support in reaching this milestone.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Constraint Programming (CP) is the study of solving problems by stating constraints such that any solution of the problem must satisfy all of the stated constraints. For example, if we consider two arithmetic constraints $X + Y = 4$ and $X - Y = 2$ together, then only one solution is accepted: $X = 3$ and $Y = 1$. CP is a form of declarative programming, as the problems are specified using a set of variables with a set of constraints between them, without specifying how the solution is be computed. The use of CP can greatly reduce the effort required to develop problem-specific algorithms and implementations. As stated by Eugene C. Freuder, "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [5]

A problem specified in such a way is called a *Constraint Satisfaction Problem* (CSP). If the variables in a CSP can only take values from a finite universe, then this CSP is called a *finite domain* constraint satisfaction problem. A constraint programming framework takes a CSP and tries to find an assignment to variables in the CSP that satisfies all of the constraints. Solving CSPs is NP-hard in general, as the Propositional Satisfiability Problem (SAT) is one of the most fundamental CSP.

Formally, a Constraint Satisfaction Problem is defined as follows [28]:

Definition 1 (Constraint Satisfaction Problem (CSP))
A CSP P is a triple $P = \langle V, D, C \rangle$ where

- V is an n-tuple of variables V $= \langle v_1, v_2, ...v_n \rangle$ with $n \geq 0$. Each variable v_i has a domain D_i associated to it.
- D is a corresponding n-tuple of domain D $= \langle D_1, D_2, ...D_n \rangle$. Each domain D_i defines a set of admissible values for the corresponding variable v_i. A domain is not limited to the numerical domains. Some domains contain strings, sets, and so on.

- C is a n-tuple of constraints C = $\langle C_1, C_2, ...C_n \rangle$. A constraint C_i is a pair $\langle R_{S_i}, S_i \rangle$ where $S_i \subseteq V$ and R_{S_i} is a relation on the variables in S_i. In other words, R_i is a subset of the Cartesian products of the domains of the variables in S_i. Intuitively, a constraint C_i imposes a relation that must hold among the variables in S_i.

A solution to the CSP P is an n-tuple of A = $\langle a_1, a_2, ..., a_n \rangle$ where each $a_i \in D_i$ and for each $C_i = \langle R_{S_i}, S_i \rangle \in C$, the projection of A onto S_i is an element in R_{S_i}. That is to say, all the constraints in C must be satisfied. We denote $sol(P)$ as the set of all solutions for CSP P. If $sol(P)$ is empty, then the CSP is unsatisfiable. For some CSPs, we may want to find an optimal, or least costly solution, given some cost function defined in terms of variables V.

Constraint Programming is well suited for many real-life problems as it provides an intuitive way of expressing the problems. In [3], CP has been applied to the areas of assignment problems, operational research problems (such as scheduling problems), business options trading, and bioinformatics. Further application areas include bounded program verification for conventional programming languages, like Java or C [12], as well as product configuration [29]. The success of applying CP to real-life problems leads us on the path to solving two distinct problems using CP solvers. The first one is an automatic debugger for the LogicQL programming language, a declarative, rule-based database language. Due to the declarative semantics, tracing why a LogicQL program has unintentional behaviors quickly gets tedious. We present a CP solver-based implementation for verifying the LogicQL program which we called LogicQL Debugger. For the second problem, we consider a system resilience problem that is important in many application domains. We formulate the problem of evaluating resilience as a CSP and provide a full GUI prototype implementation for resilience analysis using a CP system.

Contributions

As stated in the previous section, in this thesis, we use CP for two distinct problems. Therefore, the main contributions of this thesis are divided into two parts. With regards to the LogicQL program Verifier part, the main contribution is to formally define and simulate a LogicQL program based on a constraint satisfaction solver, specifically:

1. We formalize the task of the LogicQL program verification in the context of model expansion.

2. We design and implement the LogicQL Debugger, a fully automatic, GUI-based, generator of satisfying LogicQL models that gives immediate feedback to the user.

3. We define and implement the translation of the LogicQL language to the language of background solver.

4. We demonstrate that the debugger can be used to support the incremental development of LogicQL programs.

With regards to the system resilience project, the two main contributions are:

- We present the computational complexity of evaluating each of the system resilience properties.

- We define and evaluate system resilience based on a parallel constraint optimization solver.

More specifically:

1. We prove that evaluating resistance, recoverability and functionality can be done in polynomial time, whereas evaluating stabilizability is NP-Complete.

2. Given the fact that we are solving an NP-Complete problem, we formalize the system resilience problem as a CSP.

3. We implement a fully graphical user interface, based on JUNG, a Java graph framework, for users to define SR-models interactively.

4. We define and implement a translation of an SR-model to the language of underlining solver.

5. We show that our implementation can scale to systems of a useful size.

Chapter 2

LogicQL program Verification using the IDP system

Computers have been an essential tool for modern society. The dominance of computers is not just because of their speed of processing, but; more importantly, it is because they are free from making mistakes. Human beings, on the other hand (inevitably) make mistakes from time to time. In the software engineering community, some estimates suggest that over 50% of large software projects are never put into services [18]. Why does this happen? The main reason is due to the method used in software development. The traditional way of developing software can be summarized as "write, test, and fix". Theoretically speaking, this method does work if we are able to test every possible case. However, in practice, it is an impossible job since some errors may not show up for a very long time. One famous example is the Millennium bug, also called the Y2K problem. Many software developers now choose a rigorous approach to design an "error-free" system by writing down the specification of the system in some logic-based language, then using the computer to help them guarantee the behavior of the given computer system. This approach is quite feasible once the specification is correct. But due to the nature of human, we cannot guarantee that the specifications are correct. The problem of detecting whether a specification is correct is undecidable. What we can do is to implicitly check the specification in a finite universe. There is a hypothesis, called small scope hypothesis [17], argues that a high proportion of errors can be spotted by testing the program for all possible inputs within a small bound. A counterexample generator built on this hypothesis, called the Alloy Analyzer [16], has enjoyed considerable success recently. Inspired Alloy's success, our goal in this chapter is to provide a better tool which enables us to debug LogicQL programs [13]. Our tool produces a (usually exhaustive) set of models that satisfy the defined LogicQL program.

The rest of this chapter is organized as follows. Section 2.1 provides a brief review of underlying concepts including first-order logic (FO), Model Expansion (MX), FO with inductive definitions (FO(ID)), and the IDP system. In Section 2.2, we provide a description of the LogicQL language followed by the formalization of LogicQL program verification as a model expansion task. Section 2.3 describes the TXL source to source translator and the implementation of the LogicQL Debugger using TXL as a LogicQL to IDP translator and the IDP system as backend solver. Section 2.4 presents an example of using the LogicQL Debugger as a tool for the incremental development of the LogicQL program. Section 2.5 reviews some work related to LogicQL program verification. Lastly, Section 2.6 concludes this chapter with potential future work.

2.1 Background

2.1.1 First Order Logic (FO)

Definition 2 (Vocabulary and Structure)
A vocabulary σ is a set of relation and function symbols. Each symbol is associated with an arity which is the number of its arguments. Constant symbols are zero-ary function symbols.

A finite structure Å for vocabulary σ, or, a σ-structure, is a tuple containing a finite universe A, and a relation (function) for each relation (function) symbol of σ defined as follows:

- each n-ary relation symbol $R_i \in \sigma$ is mapped to a n-ary relation $R_i^{Å}$ over finite universe A, i.e, $R_i^{Å} \subset A^n$,
- each n-ary function symbol $f_i \in \sigma$ is mapped to a n-ary function $f_i^{Å}$ over finite universe A, i.e, $f_i^{Å} \subset A^n \to A$,
- each constant symbol $c_i \in \sigma$ is mapped to an element $c_i^{Å} \in A$.

In general, we write $Å = (A; R_1^{Å}, \ldots, R_n^{Å}, c_1^{Å}, \ldots, c_n^{Å}, f_1^{Å}, \ldots, f_n^{Å})$ where R_i, c_i, and f_i are relation symbols, constant symbols, and function symbols respectively. Relations are also called *predicates*. Throughout the thesis, we use these two names interchangeably.

For example, if σ has constant symbols $0, \ldots, 9$ and a binary function symbol $+$, then, assuming we are given a finite universe of discourse A= $0, \ldots, 9$, one possible finite structure for σ is $Å = (A; 0^{Å}, \ldots, 9^{Å}, +^{Å})$ where $0^{Å} = 0, \ldots, 9^{Å} = 9$ and $+^{Å}$ has its standard meaning of one digit addition on elements of universe A.

Assuming we have an infinite set of variables, we inductively define terms and formulas of first-order logic over vocabulary σ as follows:

Definition 3 (FO Terms and Formulas)

- Every variable is a term.
- Each constant symbol is a term.
- If f is a n-ary function symbol and $t_1, t_2, \ldots, t_n$ are terms, then f$(t_1, t_2, \ldots, t_n)$ is also a term.
- If R is a n-ary relation symbol and $t_1, t_2, \ldots, t_n$ are terms, then R$(t_1, t_2, \ldots, t_n)$ is a well defined atomic formula.
- If ϕ_1 and ϕ_2 are two formulas, then $\neg\phi_1$, $\phi_1 \vee \phi_2$ and $\phi_1 \wedge \phi_2$ are also formulas.
- If ϕ is a formula, then $\forall x\ \phi$ and $\exists x\ \phi$ are formulas

$\neg$, $\vee$, and $\wedge$ are *boolean connectives* and $\forall$ and $\exists$ are *quantifiers*. Formulas that do not contain any boolean connectives or quantifiers are also called *atoms*.

A variable occurrence x in a formula ϕ is *bound* if it is paired with a quantifier in formula ϕ and *free* otherwise.

A *sentence* is a formula that does not have any free variables. For example: $\forall x \exists y$ P(x,y) is a sentence whereas $\exists y$ P(x,y) is not. We denote that the set of free variables of formula ϕ by $\phi(\bar{x})$. We use $\forall\vec{x}R(\vec{x})$ as an abbreviation for $\forall x_1 \ldots \forall x_n$ R$(x_1, \ldots, x_n)$, similarly, $\exists\vec{x}R(\vec{x})$ for $\exists x_1 \ldots \exists x_n$ R$(x_1, \ldots, x_n)$.

A *valuation* σ is a function mapping each variable to an element of the Universe A. We denote $\sigma(\alpha/x)$ as the object assignment such that

$$\sigma(\alpha/x)(y) = \begin{cases} \alpha & y = x \\ \sigma(y) & y \neq x \end{cases}$$

The value of first order term t, with respect to structure Å and valuation σ, written $t^{\text{Å}}[\sigma]$, can be defined recursively by:

- If t is a constant symbol c, then $t^{\text{Å}}[\sigma] = c^{\text{Å}}$.
- If t is a variable x, then $t^{\text{Å}}[\sigma] = \sigma(x)$.
- If t is of the form $f(t_1, \ldots, t_n)$, then $t^{\text{Å}}[\sigma] = f(t_1^{\text{Å}}[\sigma], \ldots, t_n^{\text{Å}}[\sigma])$.

The intuition behind the valuation of terms of the form $f(t_1, ..., t_n)$ is straightforward. To determine the value of $f(t_1, \ldots, t_n)$, we first figure out the value of each argument $t_1, \ldots, t_n$, then we determine what function f denotes with respect to structure Å. At last, we apply the function with the values of arguments to figure out the value of $f(t_1, \ldots, t_n)$

Definition 4 (Semantics of FO Formula)
A first order formula ϕ with respect to structure Å and valuation σ is true, written Å $\vDash \phi[\sigma]$, is determined as follows:

- If ϕ is $(t_1 = t_2)$, then $\mathring{A} \models \phi[\sigma]$ iff $t_1^{\mathring{A}}[\sigma] = t_2^{\mathring{A}}[\sigma]$.
- If ϕ is $R(t_1, \ldots, t_n)$, then $\mathring{A} \models \phi[\sigma]$ iff $(t_1^{\mathring{A}}[\sigma], \ldots, t_n^{\mathring{A}}[\sigma]) \in R^{\mathring{A}}$.
- If ϕ is $\neg\psi$, then $\mathring{A} \models \phi[\sigma]$ iff $\mathring{A} \not\models \psi[\sigma]$.
- If ϕ is $\psi_1 \vee \psi_2$, then $\mathring{A} \models \phi[\sigma]$ iff $\mathring{A} \models \psi_1[\sigma]$ or $\mathring{A} \models \psi_2[\sigma]$.
- If ϕ is $\psi_1 \wedge \psi_2$, then $\mathring{A} \models \phi[\sigma]$ iff $\mathring{A} \models \psi_1[\sigma]$ and $\mathring{A} \models \psi_2[\sigma]$.
- If ϕ is $\forall x\psi$, then $\mathring{A} \models \phi[\sigma]$ iff for every α in $\mathring{A}$, $\mathring{A} \models \psi[\sigma(\alpha/x)]$.
- If ϕ is $\exists x\psi$, then $\mathring{A} \models \phi[\sigma]$ iff for some α in $\mathring{A}$, $\mathring{A} \models \psi[\sigma(\alpha/x)]$.

Note that if ϕ is a sentence, then the substitution σ has no effect on the truth value of ϕ. Thus we simply write $\mathring{A} \models \phi$, or $\mathring{A} \not\models \phi$. In the case that $\mathring{A} \models \phi$, we say that $\mathring{A}$ is a *model* for ϕ.

2.1.2 Model Expansion

In [25], the authors introduced a declarative problem solving paradigm formalized as the task of the *model expansion* (MX). In the following sections, we introduce the mathematical background of MX and review the IDP system, an MX solver for FO(ID), an extension of classical logic.

For a formula ϕ, we write *vocab*(ϕ) for the collection of exactly those relation, constant and function symbols which occur in ϕ. Let σ ,υ, and τ be vocabularies such that $\sigma \cup \upsilon = \tau$ and $\sigma \cap \upsilon = \emptyset$. Let $\mathring{A} = (A; \sigma^{\mathring{A}})$ be a σ-structure. We say a τ-structure $\mathring{U} = (U; \tau^{\mathring{U}})$ is an *expansion* of $\mathring{A}$ if their domains are the same (i.e., A = U), and $\tau^{\mathring{U}} = \sigma^{\mathring{A}} \cup \upsilon^{\mathring{U}}$.

The MX problem is that of finding models of a given formula that expand a given finite structure. We require that the user first axiomatizes the problem in some logic L, which has to have a standard model theory. The axiomatization creates a relation from an instance of the problem to its solutions. Logic L can be an extension of first-order logic, or the language of Answer Set Programming, or a Datalog-like logic programming language.

Definition 5 (Model Expansion (MX))

Formally, the MX search problem for an arbitrary logic L, denoted MX(L), is:

Given : 1. An L-formula ϕ with vocabulary $\tau = \sigma \cup \upsilon$

2. A σ -structure $\mathring{A}$

Search: a τ-structure $\mathring{U}$, which is an expansion of $\mathring{A}$ and satisfies ϕ

Thus, we expand the structure $\mathring{A}$ to τ such that it satisfies ϕ. If we can find a τ-structure $\mathring{U}$ that expands $\mathring{A}$, then we will say σ-structure has a ϕ expansion.

2.1.3 FO(ID)

The basis of FO(ID) lies in classic first-order logic (FO) defined in the Section 2.1.1. In [25], Mitchell and Ternovska proved that for any formula ϕ of first-order logic with vocabulary $\tau = \sigma \cup \upsilon$ and vocabulary σ, the problem of deciding whether an σ-structure has a ϕ expansion is in NP. And, more importantly, every problem in NP can be reduced in polytime to the model extension problem MX(FO).

We illustrate MX(FO) with graph 3-coloring problem:

Example 1

A graph 3-coloring problem instance is a structure with vocabulary σ = {Vtx, Edge(Vtx, Vtx), Color, R, G, B}. The solution is a function mapping from vertices to one of the three colors so that neighbouring vertices have a different color. We represent the expansion vocabulary υ = Coloring(Vtx), a function from vertices to colours. The FO formula for this problem is:

$\phi = \forall v_1 \forall v_2$ (Edge$(v_1, v_2) \implies$ Coloring$(v_1) \neq$Coloring(v_2))

More formally, the task is to find an interpretation for function Coloring(Vtx) such that:

(Vtx, Edge(Vtx, Vtx), Color, R, G, B, Coloring(Vtx)) $\models \phi$

A (inductive) definition $\triangle$ is a set of rules of the form $\forall \vec{x}(X(\vec{t}) \leftarrow \phi)$, where $\vec{x}$ is a tuple of variables, X is a predicate symbol, ϕ is a first-order formula, and $\vec{t}$ is a tuple of terms such that any free variable in $\vec{t}$ is among the variables in $\vec{x}$. $X(\vec{t})$ is called the head of the rule and ϕ the body. The "$\leftarrow$" symbol is called definitional implication and it has a different semantic meaning from material implication "$\rightarrow$". Intuitively, the definitional implication should be read as "if", such as "$X(\vec{t})$ is true if ϕ is".

The FO(ID) formulas are constructed from boolean combinations of definitions $\triangle$ and FO formulas. The semantics of FO(ID) extends FO with the well-founded semantics of logic programming. More details can be found in [21] and [11].

In the context of finite structures, MX(FO(ID)) does not have more expressive power than MX(FO). However, properties like transitive closure in a graph and rules in LogicQL are not easily expressible in MX(FO). We shall see that supporting inductive definitions makes expressing such properties natural and trivial.

To demonstrate the usefulness of FO(ID), consider the following program that computes the transitive closure of a graph as well as the complement of the transitive closure `tc`. The graph is given as input via a binary relation `e` containing its edges.

Example 2

```
∀x ∃y   n(x) <- e(x,y).
∀x ∀y   tc(x,y) <- e(x,y).
```

```
∀x ∀y ∀z   tc(x,y) <- e(x,z) ∧ tc(z,y).
∀x ∀y ∀z   ntc(x,y) <- n(x) ∧ n(y) ∧ !tc(x,y).
```

2.1.4 The IDP system

The IDP system [36] is an MX solver. Its specification language is based on FO(ID). More concretely, the logic L for the IDP system is a full first-order logic with an order-sorted type system, inductive definitions, partial functions, arithmetic, existential quantifiers with numerical bounds and aggregates [36]. Though these extensions do not increase the class of problems that can be modeled in MX(FO), they do considerably simplify the modeling of the LogicQL language. For readability, we will use standard logical connective notations throughout the thesis. For the ASCII notation of the connectives in IDP's input language, we refer to Table 2.1.

In this section, we describe the IDP Program. The basic overall structure of an IDP Program consists of 4 sections:

```
vocabulary V {...}
theory T:V {...}
structure S:V {...}
procedure main() {...}
```

Figure 2.1: IDP Program Structure

Vocabulary has declarations of all types, predicate and functions symbols. Vocabulary declares all the variables of the problem. Take the vocabulary part for the 3-coloring problem as an example:

```
vocabulary V {
type Vtx
type Color subtype of int
type Edge(Vtx,Vtx)
Coloring(Vtx):Color
}
```

The vocabulary is named V and introduces two types: "Vtx" and "Color". Note that the second type is a subtype of a predefined type: int. It also declares a binary predicate symbol Edges, a relation between two vertices and a unary function symbol "Coloring" mapping vertex to its color. All IDP vocabularies are well-typed as cyclic dependencies are forbidden and will result in a compiler error. For example, "type A subtype of B and type A supertype of B" are not allowed as A depends on B.

Theory has all FO sentences and inductive definitions. As an example, Figure 2.2 gives the theory part for the 3-colouring problem:

```
theory T: V {
!v1[Vtx] v2[Vtx] : Edge(v1,v2) => Coloring(v1) ~= Coloring(v2).
}
```

Figure 2.2: 3-Coloring Problem using IDP

Logical Connective	$\forall$	$\exists$	$\wedge$	$\vee$	$\neg$	$\rightarrow$	$\leftarrow$
IDP ASCII Representation	!	?	&	|	~	=>	<-

Table 2.1: IDP ASCII equivalents for Logical Connectives

The meaning of this formula has been explained in Section 2.1.3. The theory part consists of ASCII representations of FO(ID) formulas. The mapping from logical connectives to ASCII symbols is provided in Table 2.1.

Structure defines a partial instance for the problem. One possible instance of the 3-colouring problem is:

```
structure T: V{
     Vtx = {  V1 ; V2 ; V3 }
     Color = { 1..3}
     Edge= {  V1,V2;
              V2,V3;
              V3,V1;}
    }
```

In the box above, "Vtx = {V1 ;V2 ;V3 }" specifies elements of Vtx to be V1, V2 and V3. Similarly, elements of Color are 1, 2 and 3. "Edge={V1,V2; V2,V3; V3,V1;}" specifies tuples of relation Edge to be {(V1,V2),(V2,V3),(V3,V1)}. There is no specification for function "Colouring" as this is what we want to find.

The last part of an IDP program is **Procedures**. The procedure defines what kind of inference we want to use. In the case of the 3-colouring problem, we want to perform model expansion, since we want to search for an interpretation of function "Colouring" that satisfies theory T. The procedure is the following:

```
stdoptions.nbmodels = 2
          printmodels(modelexpand(T,S))
```

By default, IDP returns one model of the theory. By setting stdoptions.nbmodels to 2, IDP will return 2 models (if there is more than 1). When set to 0, IDP will print all models of the theory.

2.2 LogicQL

LogiQL is a declarative logic programming language developed by LogicBlox, Inc[13], to harness the power of Datalog with first-order logic constraints to support building databases. It has been developed based on Datalog.

Prolog is a popular logic programming language initially implemented in France in 1972. Both Prolog and LogiQL are built on first-order logic and provide elegant support for deductive inferences, including recursive rules. Unlike Prolog, LogiQL programs are guaranteed to terminate. This useful property is achieved by placing further syntactic restrictions on the kinds of rules that can be formulated.

Datalog was also developed in the 1970s with a specific emphasis on providing access to deductive databases. Unlike Datalog, LogiQL has been designed to handle large quantities of data stored in industrial-strength databases. Hence, it can be thought of as a query language for such databases. In fact, the name "LogiQL" combines "logic" with "QL", which is shorthand for "query language."

From a database point of view, LogiQL has a few of differences with SQL:

	SQL	LogiQL
model	tables	predicates
data	relations	sets of tuples
logic	queries views	rules queries

There are two major differences between a predicate and a relational table:
(i) A predicate contains a set of tuples, whereas a table contains a bag of tuples. That is, a table may contain many duplicate copies of the same tuple; a predicate only contains the same tuple once. *(ii)* A predicate does not contain NULLs.

From the FO(ID) point of view, on the other hand, LogicQL is essentially a fragment of FO(ID) with different syntax.

Logical Connective	$\wedge$	$\vee$	$\neg$	$\rightarrow$	$\leftarrow$
LogicQL ASCII Representation	,	;	!	->	<-

Table 2.2: LogicQL ASCII equivalents for Logical Connectives

2.2.1 LogicQL Language

The basic building blocks of the LogicQL language are the LogicQL formulas. LogicQL formulas can be viewed as a proper subset of FO formulas in terms of their semantic meaning. Specifically, the LogicQL formulas are defined inductively as follows:

- Every variable is a term.
- Each constant symbol is a term.
- If R is a n-ary predicate symbol and $t_1, t_2, ..., t_n$ are terms, then R$(t_1, t_2, ..., t_n)$ is an atomic formula.
- If R is a n-ary functional predicate symbol and $t_1, t_2, ..., t_n$ are terms, then R$[t_1, t_2, ..., t_{n-1}] = t_n$ is an atomic formula.
- If ϕ_1 and ϕ_2 are two formulas, then $!\phi_1$, $\phi_1;\phi_2$ and ϕ_1, ϕ_2 are also formulas. "!", ";" and "," are boolean connectives negation, disjunction and conjunction respectively.

Compared with FO formulas defined in Section 2.1.1, the LogicQL formulas have no functions symbols but include functional predicates. The inclusion of functional predicates does not increase the expressive power of the language as each functional predicate is equivalent to an FO predicate with a functional constraint that makes sure no two tuples in the predicate share the same sequence of first n-1 values.

Every variable in a LogicQL formula is implicitly bound to a quantifier. The variable whose name consists of a single underscore character "_" is an *anonymous* variable that is bound to an existential quantifier. By calling it anonymous, we mean that the name of this variable does not matter and will not be used in other parts of the formula. All other variables are bound with universal quantifiers. Therefore, every LogicQL formula can be viewed as an FO sentence.

Now, we are ready to formally define the LogicQL programs:

Definition 6 (LogicQL program)
A LogicQL program D is a set clauses $D = F \cup R \cup C$ where:

- F is a set of facts. Each fact consists of an atomic formula, e.g. "+House("Windsor")."

- R is a set of rules, that is, formulas connected by a left-hand arrow. Rules provide instructions for deriving new facts from existing ones and can be translated to an equivalent inductive definition that has the same semantic meaning. For instance **`person(x)<-male(x)`** says all males are persons. This rule is equivalent to: ∀**`x (person(x)<-male(x))`**
- C is a set of constraints. A *constraint* expresses an invariant property of the data contained in the program's workspace. It is expressed using a rightward facing arrow (**`->`**). Constraints constrain predicate facts, e.g. with types or other restrictions. The interpretation of a constraint, just like the implication of two FO formulas, is that whenever the formula on the left-hand-side of the arrow holds true, the formula on the right-hand side of the arrow must hold true, as well.

  There are three forms of constraints:

  - formula -> formula.
    For example: hasGenderCode(:gc) -> gc = "M" ; gc = "F".
    means Possible gender codes are M and F.
  - -> formula. This is equivalent to TRUE -> formula
    For example: ->Person(_).
    means Person predicate is not empty.
  - ! formula. This is equivalent to TRUE -> !formula
    For example: !isParentOf(p, p).
    means no person p is a parent of him or her self.

An *Intensional DataBase (IDB)* predicate is a predicate occurring in the head of rules in D. All other predicates are *Extensional DataBase (EDB)* predicates. The extensional (database) vocabulary, denoted vocab(EDB), consists of the set of all EDB predicate symbols; whereas the intensional vocabulary vocab(IDB) has all the IDB predicate symbols. Hence, the vocabulary for the program D, vocab(D), is the union of vocab(EDB) and vocab(IDB).

We say that a constraint $c \in C$ is an EDB constraint if $vocab(c) \subset vocab(EDB)$, i.e., if it contains only extensional predicate symbols. The remaining constraints are said to be IDB constraints.

When running the LogicQL program D, the LogicQL engine handles the program D in the following fashion:

- An EDB instance is created by all facts $F \subset D$.
- Check if the EDB instance satisfies all EDB constraints in D.
- If it is satisfiable, run all rules $R \subset D$ to derive the instance of IDB.
- Check if the IDB instance violates the IDB constraints or not
- If inconsistency found, abort the program and rollback to the initial state.

It is worth noting that the complexity of checking constraints and deriving new IDB facts are both polynomial time. Hence, it is impossible to directly use the LogicQL engine to solve MX tasks (unless P=NP).

2.2.2 LogicQL Program Verification as a Model Expansion Task

To verify a LogicQL program, we embrace the idea of small scope hypothesis [17]: exhausting the entire space of executions within some finite bounds, in which a program is analyzed for all acceptable models. That is, we take a LogicQL program and a bound on the size of the finite universe, the correctness of the program is checked against all potentiality satisfying models. This explicit analysis is not only able to reveal bugs in the program but also errors in the intended specification themselves.

Formally, given a LogicQL program D and a finite vocab(EDB)-structure Å, the task of model expansion for a LogicQL program verification is:

Definition 7
Given:

1. A LogicQL program D with vocabulary vocab(D)=vocab(IDB) ∪ vocab(EDB)
2. A finite vocab(EDB)-structure Å.

Search:

A vocab(D)-structure Ů, which is an expansion of Å and satisfies D.

2.3 LogicQL Debugger

In this section we present the LogicQL Debugger, an integrated tool for programming and debugging the LogicQL programs based on the MX formulation defined in the previous section. Rather than using trial and error method for finding satisfying models of the LogicQL programs on the cumbersome platform-dependent LogicQL evaluation engine, our LogicQL Debugger takes a LogicQL program D as input and asks the users to specify the finite domain for D, and then translates program D, together with the domain, to an IDP problem specifications and then it generates the satisfying models using the IDP solver. In addition to the translation program, we provide a cross-platform Graphical User Interface (GUI) for the general users to edit LogicQL programs and view the results from the IDP solver. User knowledge of the IDP system is not required as every IDP-related task is done automatically underneath the GUI.

The LogicQL Debugger has three components. i) the GUI-based editor and GUI-based Model Scope editor, ii) the LogicQL to IDP translator and iii) the IDP solver. Figure 2.3 shows the architecture of the LogicQL Debugger.

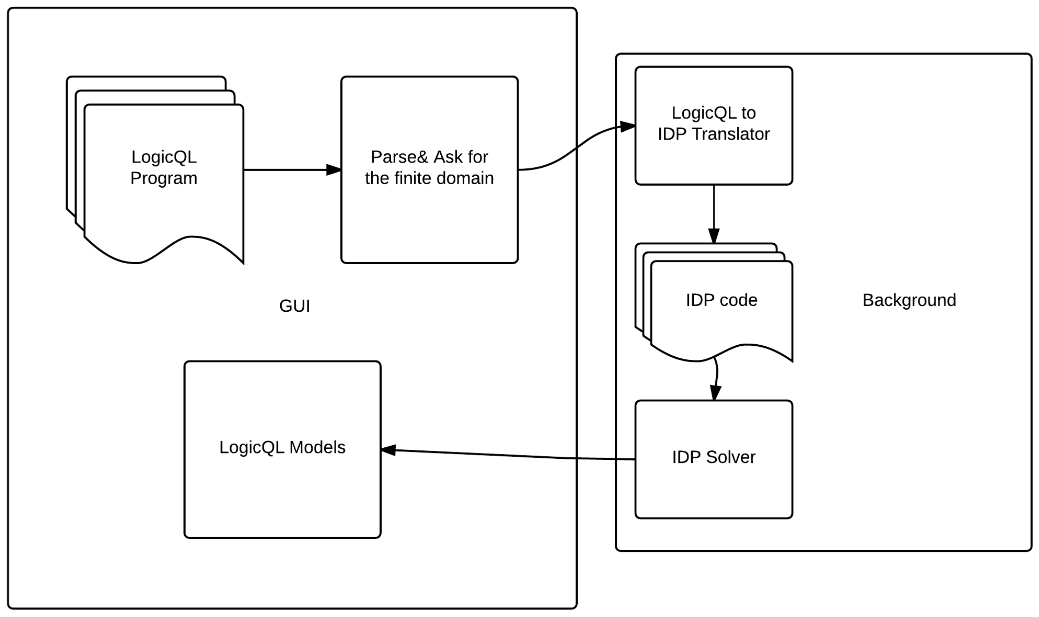


Figure 2.3: The High-level Architecture of the Debugger

2.3.1 LogicQL Debugger GUI

Figure 2.4 presents the main GUI window of the LogicQL debugger. The GUI has a panel for the LogicQL code area and another panel for the console messages. The users can either load LogicQL code from an external file or just type the code in the code area. Once the user decides to check the LogicQL code, he can then press the "Run" button in the lower right. One of the following cases will occur:

Case 1: Syntax errors found in the code

Suppose the user entered the following code:

```
ServierName(n)->string(n).
Date(expDate)-> int(expDate)
```

Note that the user forgot to end the second constraint with the "dot". After the "run" button is pressed, the debugger reports the syntax errors, as can be seen in Figure 2.5.

Case 2: No Syntax errors in the code

If the user had fixed the syntax error in case 1, a domain selection window would have popped up, asking for the finite domain for this LogicQL program.

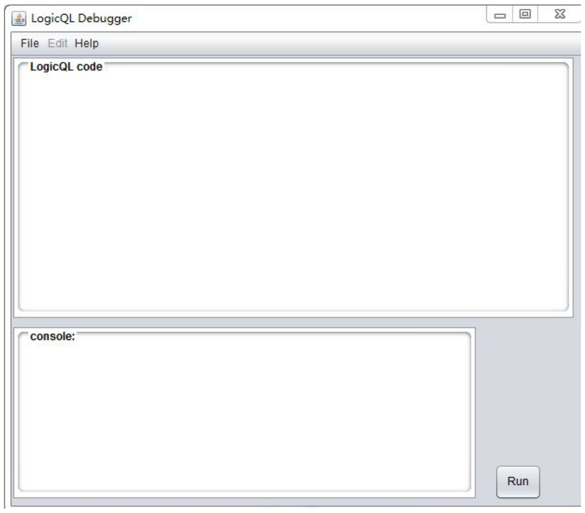
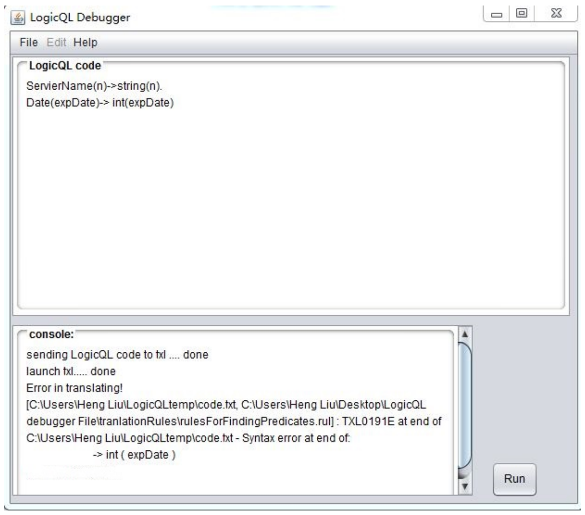


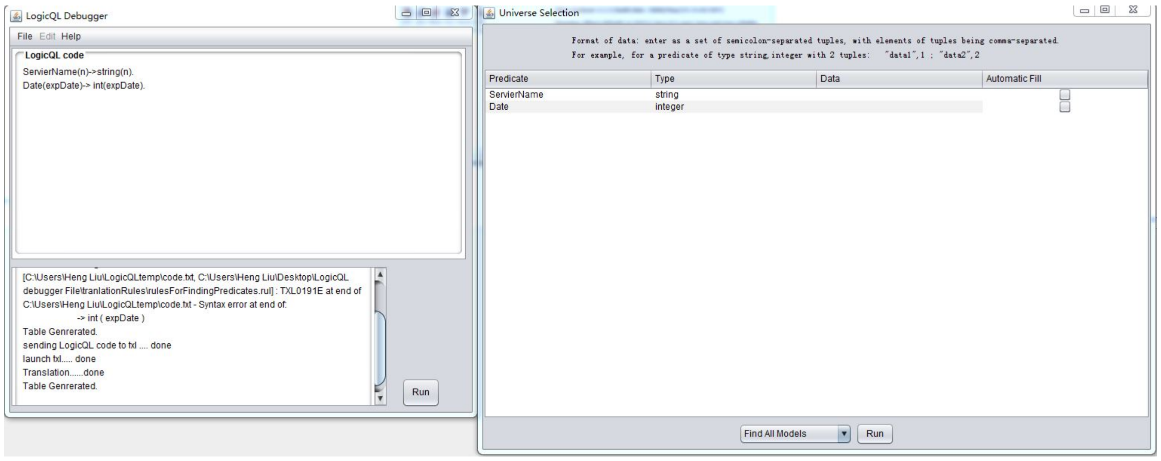
Figure 2.4: Main GUI of LogicQL Debugger



```
ServierName(n)->string(n).
Date(expDate)-> int(expDate)
```

```
sending LogicQL code to txl .... done
launch txl..... done
Error in translating!
[C:\Users\Heng Liu\LogicQLtemp\code.txt, C:\Users\Heng Liu\Desktop\LogicQL
debugger File\tranlationRules\rulesForFindingPredicates.rul] : TXL0191E at end of
C:\Users\Heng Liu\LogicQLtemp\code.txt - Syntax error at end of:
              -> int ( expDate )
```

Figure 2.5: LogicQL Debugger GUI Showing Syntax Errors



```
ServierName(n)->string(n).
Date(expDate)-> int(expDate).
```

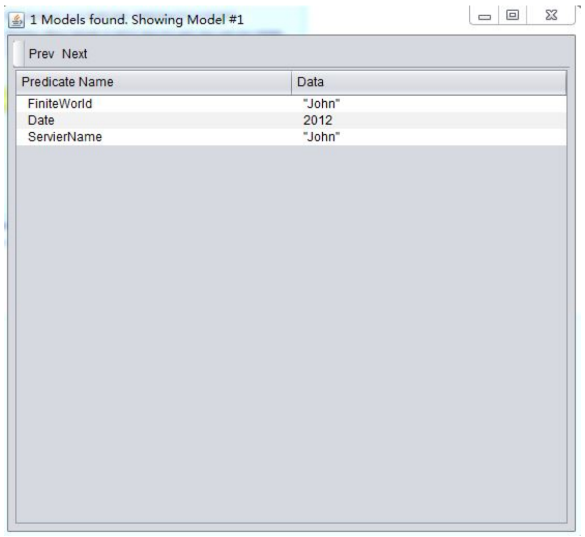
```
[C:\Users\Heng Liu\LogicQLtemp\code.txt, C:\Users\Heng Liu\Desktop\LogicQL
debugger File\tranlationRules\rulesForFindingPredicates.rul] : TXL0191E at end of
C:\Users\Heng Liu\LogicQLtemp\code.txt - Syntax error at end of:
              -> int ( expDate )
Table Genrerated.
sending LogicQL code to txl .... done
launch txl..... done
Translation......done
Table Genrerated.
```

Format of data: enter as a set of semicolon-separated tuples, with elements of tuples being comma-separated.
For example, for a predicate of type string,integer with 2 tuples: "data1",1 ; "data2",2

Predicate	Type	Data	Automatic Fill
ServierName	string		☐
Date	integer		☐

Figure 2.6: LogicQL Debugger GUI for Defining Finite Domain

Suppose the user adds "John" to *ServerName* and 2012 to *Date* and then presses the run button with "Find all models" option selected. The IDP solver tries to find all satisfying models and the final result is shown in Figure 2.7.



Predicate Name	Data
FiniteWorld	"John"
Date	2012
ServierName	"John"

Figure 2.7: LogicQL Debugger GUI for showing Results

Unsurprisingly, given the fact that the user defined everything in the domain selection window and there are no rules in this LogicQL program, only one model can be found. And this model is exactly what the user has entered.

2.3.2 The Translator

In this section, we provide the technical details on how the logicQL program is translated to its equivalent IDP specifications. We use TXL [9], a hybrid functional and rule-based language for performing source-to-source transformations. The TXL transformation process consists of three parts:

- 1. Parse Phase. The TXL tokenizes the entire LogicQL program, and then parses it according to the context-free grammar for LogicQL language. The grammar is described in BNF-like notation. A portion of the LogicQL grammar is provided in Figure 2.8.
- 2. Transformation Parse. It applies a set of transformation rules that takes the parse tree of the LogicQL input and transforms it into a new tree that corresponds to the equivalent IDP output.
- 3. Unparse Phase. It unparses the tree produced in step 2 and produces the IDP output based on the context-free grammar for IDP language.

```
define LogicQL_program
         [repeat clause]
end define

define clause
     [constraint]
        | [logicQLrule]
        | [comment]
        | [fact]
end define

define constraint
      [formula] -> [formula] [end_formula]
        |  -> [formula] [end_formula]
        |  '! [formula] [end_formula]
end define
```

Figure 2.8: A Fragment of LogicQL Grammar

The TXL transformation approach brings two benefits to the LogicQL Debugger. First, the initial parse phase acts as a LogicQL syntax checker. Should the program contain any syntax errors, TXL will detect and report these errors. Thus, no pre-processing is needed to guarantee that the LogicQL program is syntactically correct. Second, we can dynamically generate the finite domain selection table, since the parsed tree enables us to extract the vocabulary of the input LogicQL program. Note that LogicQL program is strongly typed, so the predicate-type binding tables can be extracted at the same time.

The transforming paradigm involves three steps: (1) build a typed symbol table for both predicates and variables in the LogicQL program; (2) integrate the logicQL and IDP grammars to form an intermediate translation grammar; (3) for each grammar non-terminal, apply a set of independent translation rules to it.

Build symbol tables

The symbol tables serve as a database for the transformation process. The primary content of the symbol tables are the predicate or variable symbols and data type information for each clause in the LogicQL program. This information is gathered by a special kind of transformation rule in TXL called a constructor. A constructor is often used to build intermediate subtrees for use later in the transformation processes. In our implementation, a table entry looks like:

```
define typedVariableEntry
[number]  [repeat identifier]
  end define

  function main
          export  table_typedVarlist [repeat typedVariableEntry]
                  -
       ...
    }
```

Each table entry maintains an entry for each predicate or variable in the following format:

```
 Clause Number, symbol, type 1, ... ,type n
```

We allow multiple types for the same symbol due to the fact that we need to have a symbol-type pair for variables as well as predicates. It is usually the case that the predicate is a non-unary predicate. Therefore we need store one type for each of the arguments for that predicate. For example, if the symbol table has to store information about the following LogicQL program:

```
Vtx(v)  -> String(v).
Edge(v1,v2) -> Vtx(v1), Vtx(v2).
```

Then the content of symbol table is:

```
1 v   String
2 v1   Vtx
2 v2   Vtx

1 Vtx String
2 Edge Vtx    Vtx
```

This symbol table is stored as a global variable that can be accessed any time during the transformation process. Whenever a symbol needs to be searched in a symbol table, it is searched by matching the clause number and the symbol altogether and thus guarantees we looked up the intended symbol. In our implementation, we build a one table for predicates and another for variables. The vocabulary of the input logicQL program is the collection of all symbols in predicate symbol table.

Integrate source and target grammars

Beside the already defined logicQL grammar and IDP grammar (please refer to the appendix for the full grammar), we build an integrated translation grammar as the media of exchange. A program in the integrated grammar can either be a LogicQL program or an IDP program. We also relax the definition of IDP sentences and formulas so that both the untranslated and translated code are accepted in every context, allowing us to work independently on each of the components. The integrated grammar is not used during the parse phase and unparse phase and thus guarantees the input LogicQL program and the output IDP specification are free of syntax errors.

```
include "logicQL.Grm"
include "IDP.Grm"

define program
         [LogicQL_program]
        | [IDP_program]
end define

redefine IDP_sentence
        ...
        |[formula]
end redefine
redefine formula
         ...
        | [formula] '& [formula]
        | [formula] '| [formula]
        |  '~ [formula]
end redefine
```

Translate to IDP

As stated in Section 2.1.4, the output IDP program has four (independent) sections, namely, vocabulary, theory, structure, and procedure. Here, we present in detail how we build the IDP program section by section. There are four independent sections in an IDP program, We present the building procedure of these individual sections

Vocabulary is the section that declares of all types, predicate and functions symbols. Formally, an IDP vocabulary section is defined as:

```
  vocabulary [id] {  [repeat IDP_symbol_declaration] }
```

where vocabulary is the keyword, [id] is the name of the vocabulary and it can be any string beginning with a letter. We use V as the name of vocabulary. Each symbol declaration is of the form:

```
    [id] ( [list id]  )
```

The following procedure illustrates the building procedure:

1. the section begins with "vocabulary [id] { "
2. For each table entry of the form: [Predicate] [Type 1] ... [Type n]
   add [Predicate] ([Type 1], ..., [Type n]) to the section.
3. Close the section with "}"

Theory consists of all FO sentences and inductive definitions(rules). Recall that a constraint LogicQL program is a set of clauses and each clause is either a fact, a constraint or a rule. To help the transformation process, we employ three kinds of tables, one assigns a unique clause number to each LogicQL clause and the other two is the predicate/ variable symbol tables mentioned in the previous section. We now ready to present how each kind of LogicQL clauses is transformed into IDP clauses:

1. Replace Every LogicQL logic connective with its equivalent IDP connective according to Table 2.2 and Table 2.1.
2. Replace every functional predicate of the form:

   ```
           [Predicate symbol] [ [Argument list ] ] = [Argument]
   ```

   with

   ```
          [Predicate symbol] ([Argument list], [Argument] )
   ```

   That is to say, we treat each functional predicate as regular predicate. To preserve the semantic meaning of the functional predicate, we include a functional constraint for each functional predicate in the program. This is to ensure that the model generated by the IDP solver is indeed an instance acceptable by the logicQL engine.
   For example, for a functional predicate person_has_age(String):int, we will add the following constraint to the IDP theory:

   ```
   !x1[String] !y1[int] !y2[int]
   :  person_has_age[x1] = y1 & person_has_age[x1] = y2 =>  y1=y2
   ```

3. For every variable symbol of the form [variable] [type] we found by matching the clause number in universal variable table, we append ![variable] [[type]] to the beginning resulting sentence.

4. Similarly, for each symbol from existential variable table,
   we append ?[variable] [[type]].

5. For each form of logicQL clause, transform to its sentimentally equivalent form in IDP language:

   - **Clauses of the form: [formula] -> [formula] [end formula]**
     We replace the original clause of form: [formula] -> [formula] [end formula] with :[formula] => [formula] and the final result is of the form:

     ```
      ! Variable1 [Type 1] !Variable2 [Type 2] ... ?VariableN [Type N]
                   :[formula] => [formula]
     ```

     For example, the 3-colouring specification in LogicQL language is:

     ```
        Edge(v1,v2) ->  Coloring(v1) , !  Coloring(v2)
     ```

     and will be translated to:

     ```
     !v1[Vtx] !v2[Vtx] : Edge(v1,v2) =>  Coloring(v1) & ~Coloring(v2)
     ```

   - **Clauses of the form: -> [formula] [end formula]**
     is replaced with :
     $[formula]$

   - **Clauses of the form: ! [formula] [end formula]**
     is replaced with :
     $\sim[formula]$

   - **Clauses of the form: [formula] <- [formula] [end formula]**
     is replaced with :
     $[formula] <= [formula]$

Structure contains a (partial) instance description for the IDP specification. An instance description contains the domain of types and may contain interpretations of predicates. Interpretations of predicates are given as a set of semicolon spaced tuples, where each element of tuples being comma-separated. There are two sources of the instance description:

1. For each fact in the LogicQL program of the form "+ PredicateName(Data).", we add "Data" to the set of semicolon spaced tuples for PredicateName.

2. Alternatively, the user may want to populate the instance description from GUI. Anything entered in GUI is already in IDP format and thus no further transformation is needed.

Procedure defines what inference we want to use. In our case it is model expansion. In the current implementation, the procedure section is fixed as shown in the following:

```
procedure main(){
 stdoptions.nbmodels = n
        printmodels(modelexpand(T,S))
}
```

The number "n" is the number selected by the user from the domain selection GUI, this number represent the number of models the user wants to find by the IDP solver. By default, the IDP solver returns all satisfying models.

2.4 Case study: Incremental Development of Automated Software Distribution for Airplanes

In this section, we demonstrate how the debugger can be effectively used in finding out the impact of changes in rules or finding bugs in the current implementation.

With the advances in electronic communication, we see an increasing need for electronic collaboration. The applications areas include software distribution, healthcare, and many others [15]. Many of these applications comprise confidential information and thus making trust management between collaborated parties highly critical. Policies are described to define conditions under which an action is approved or forbidden. The collaborators usually only have a rough idea of what policies should be set and, as a consequence, policies are typically described informally. Take the following statement as an example: "only the airline-approved partner can perform a software update on airplanes". This statement is a policy as it governs who may perform the software update on the airplane, based on the airline-approved partner list. But it is not clear if this policy asks for the software update on the planes or only forbids anyone who is not on the list from doing an update. If the developer of electronic collaboration system misinterprets the ambiguous policy, the collaborated parties may suffer lost revenue and breached security.

Our debugger produces a (usually exhaustive) set of models that satisfies the defined policies. Whether a user is interested in the impact of changes in policies or finding bugs in the current implementation, the user can check satisfying models to concretize policies' behavior. By doing so, our tool provides an integrated environment for policy-based trust management system based on LogicQL. In particular, the use case we focus on in this section is automated electronic software distribution for airplanes. Given the immediacy of the satisfying models the debugger provides, one can refine the (possibly unambiguous) policies in a series of small modifications and additions.

2.4.1 The problem

Airplanes are maintained by service providers who are contracted by the airline. An airline may have several contracted service providers, responsible for performing software maintenance of different airplane types or on different locations. The service providers have to authenticate themselves to the airplane to perform maintenance, such as a short-term authentication token must be obtained from the airline. The question is to decide for which airplane types and which part of airplane service providers are authorized to install the update. More details on the problem of Automated Software Distribution for airplanes can be found in [19].

We assume that software updates were provided by trusted suppliers (i.e., Boeing) and the airplane trusts all the information its airline provides. We also assume that the authorization policy that state the condition under which a service provider can perform software on which type of aircraft is specified as follows:

1. Airline keeps a list of which parts needs to be loaded into which tail.
2. Airline keeps a list of contracts with service providers that specify the service providers name, expire date of the contact and the airplane type.
3. No work is assigned to a partner whose contract has expired.
4. The airline currently only working on loading parts to "tail123" on Boeing 787.

2.4.2 LogicQL formulation

We begin out with LogicQL formulation of the problem with declaring *entity types*. An entity is a concrete object that may contain one or more values. Also, the collection of entity declaration defines the structure of this LogicQL program. Note that throughout this section, we conventionally start with an upper case letter for entity types and lower case letters for all other non-entity predicates. Here is the LogicQL code for declaring entity types:

```
ServierName(n)  -> string(n).
AirplaneType(t)  -> string(t).
PartName(p)  -> string(p).
TailName(tail)  -> string(tail).
Date(expDate)  -> int(expDate).
```

This code introduces five entity types(or simply types) – Servier Name, Airplane Type, Part Name, Tail Name and Date – each representing a set of objects. For example, the ServierName entity type would contain the set of Servier Name used in the LogicQL program. The code contains neither constraints nor rules, so there's no need to start debugging yet.

Next, we add predicates to model the first two policies addressed in the previous section. The following code declares two constraints that the first two lines of code restrict *servicer* predicate to be a subset of $ServierNameXDateXAirplaneType$ while the last two lines of code restrict *approvedPart* predicate to be a subset of $PartNameXTailName$:

```
servicer(n,expDate,t) − > string(n),int(expDate),string(t).
servicer(n,expDate,t) − > ServierName(n),Date(expDate),AirplaneType(t).
approvedPart(p,tail) − > string(p),string(tail).
approvedPart(p,tail) − > PartName(p),TailName(tail).
```

Figure 2.9: A (buggy) LogicQL Program for the First Two Policies

We then give a scope that bounds the size each of the types: in our case, to two object in each type, except for AirplaneType, which is limited to one object. Since, for now, we only want to debug the first policy, the approvedPart predicates are also fixed as shown in Figure 2.10. Running the LogicQL Debugger finds 16 different models. Each of the models can be shown in either textual or graphical, here the textual view of the first model is presented in figure 2.11.

```
ServierName = "A";"B";
AirplaneType = "777"
PartName = "partA";"partB"
TailName = "tail123";"tail456"
Date = 2017;2015
approvedPart="partA","tail123"
```

Figure 2.10: Finite Domain for the Debugger

```
Date = 2015; 2017
AirplaneType = 777
Name = "A"; "B"
PartName = "partA"; "partB"
TailName = "tail123"; "tail456"
approvedPart = "partA","tail123"
servicer = "A",2015,777; "A",2017,777; "B",2015,777; "B",2017,777
```

Figure 2.11: The First Model from the Debugger

We immediately see that our model allows two distinct contracts for the same service provider and airplane type pair with the only difference been in the "expDate". This result is not surprising since we did not add any constraints that restricted the servicer predicate to be a functional predicate (i.e. each service provider and airplane type should map to only one expire date). We would like to make sure that this model will not happen again, so we add a new constraint on *servicer* predicate:

```
servicer(n,expDate,t) − > string(n),int(expDate),string(t).
servicer(n,expDate,t) − > ServierName(n),Date(expDate),AirplaneType(t).
approvedPart(p,tail) − > string(p),string(tail).
approvedPart(p,tail) − > PartName(p),TailName(tail).
servicer(n,expDateA,t),servicer(n,expDateB,t) − > expDateA = expDateB.
```

Figure 2.12: Refined LogicQL Program for the First Two Policies

Running the debugger with the same scope described in figure 2.10 again gives nine models, and we can check that all of the models are consistent with the policy. These little simulations are useful because, with minimal effort on writing test cases, one can confirm that the implementation of the policies not only considered obvious cases but also other cases that might not have been considered at all. Next, we continue with the implementation of the last two policies that specify which service provider is approved to load which part onto an airplane:

```
load(n,p) < − servicer(n,expDate,t), approvedPart(p,tail),
expDate > 2014
t= "787",tail="tail123" .
```

As service providers are contracted by airlines to perform software updates for one particular software package, we express the policies as a LogicQL rule that automatically derives and authenticates service providers to airplanes. As shown in the box above, a servicer n is allowed to load package p into the airplane only if n's contract has not expired (expDate $>$ 2014) , type of airplane is Boeing 787 (t= "787") and p is supposed to be loaded into the plane with tail number "tail123" (tail="tail123").

Following the same strategy we used for debugging the LogicQL constraints, we firstly give a scope of the domain. This time, we are limited to two objects in each type, except for service provider that is limited to three objects as shown in figure 2.13. We choose to fix the servicer predicate since exhausting all possible combinations of servicer predicate gives more than 512 models. In theory, though, by not fixing some predicates, would give us complete coverage of possible models than have hand generated test cases. Most flaws can be illustrated by small scopes since those flaws arise from the cases being mistakenly handled or simply forgotten to take into account whatever the size of scope. This observation is

```
Name = "A";"B";"C"
AirplaneType = "777";"787"
PartName = "partA";"partB"
TailName = "tail123";"tail456"
Date = 2013;2015
servicer = "A",2015,"787" ; "B",2013,"787" ; "C",2015,"787" ; "C",2013,"777"
```

Figure 2.13: New Finite Domain for the Debugger

called small scope hypothesis [16]. Therefore, in practice, we tend to fix some "bug-free" predicate to limit the scope of models to a reasonable number (usually less than 50).

Running the debugger again gives us another 16 different models. The first five models are shown in figure 2.15. By going through all the models, we notice that the load predicate in first four models is empty which is acceptable since non-qualified service provider will never be selected. But when we move to the model 5, we notice that two service providers are authorized to load the same software part on the same plane. While this model does satisfy all the policies, it also reveals a scenario that we forgot to handle – what if there is more than one service provider who is qualified in performing software updates? Without introducing new entity types, we add one more constraint specifying that there is only one service provider selected to load each part into a plane:

```
load(n1,p),load(n2,p) − > n1 =n2.
load(n,p) < − servicer(n,expDate,t), approvedPart(p,tail),
, expDate > 2014
t= "787",tail="tail123" .
```

Figure 2.14: Refined LogicQL Program for the Last Two Policies

By running the debugger again, we conform that we've eliminated the redundant authorization problem. The final version of the LogicQL code discussed in this section is shown in figure 2.16.

```
Common Entity Types
Date = 2013; 2015
AirplaneType = 777; 787
Name = "A"; "B"; "C"
TailName = "tail123"; "tail456"
PartName = "partA"; "partB"

Model 1
servicer = "A",2015,787; "B",2013,787; "C",2013,777; "C",2015,787
load =
approvedPart = "partA","tail456"; "partB","tail456"

Model 2
approvedPart = "partA","tail456"
load =
servicer = "A",2015,787; "B",2013,787; "C",2013,777; "C",2015,787

Model 3

approvedPart = "partB","tail456"
load =
servicer = "A",2015,787; "B",2013,787; "C",2013,777; "C",2015,787

Model 4
approvedPart =
load =
servicer = "A",2015,787; "B",2013,787; "C",2013,777; "C",2015,787

Model 5
approvedPart = "partA","tail456"; "partB","tail123"; "partB","tail456"
load = "A","partB"; "C","partB"
servicer = "A",2015,787; "B",2013,787; "C",2013,777; "C",2015,787
```

Figure 2.15: First Five Models from the debugger

```
ServierName(n) − > string(n).
AirplaneType(t) − > string(t).
PartName(p) − > string(p).
TailName(tail) − > string(tail).
Date(expDate) − > int(expDate).
servicer(n,expDate,t) − > string(n),int(expDate),string(t).
servicer(n,expDate,t) − > ServierName(n),Date(expDate),AirplaneType(t).
approvedPart(p,tail) − > string(p),string(tail).
approvedPart(p,tail) − > PartName(p),TailName(tail).
servicer(n,expDateA,t),servicer(n,expDateB,t) − > expDateA = expDateB.
load(n1,p),load(n2,p) − > n1 =n2.
load(n,p) < − servicer(n,expDate,t), approvedPart(p,tail),
, expDate > 2014
t= "787",tail="tail123" .
```

Figure 2.16: Final Version of LogicQL Program for the Automated Software Distribution for Airplanes

2.5 Related work

The LogicQL Debugger is, to our knowledge, the first program verifier for the LogicQL language - a programming language that supports FO constraints and inductive definitions. In this section, we review related work on the declarative program verification techniques on related but usually less expressive languages.

Datalog is the foundation of the LogicQL language. Datalog can be viewed as a subset of LogicQL language that contains only LogicQL rules. The DES system [7], a debugger for Datalog, uses query debugging based on the principles of algorithmic debugging. A Datalog query for a given Datalog Program is a single Datalog rule of interest. The result of applying a query is the derived IDB on the basis of the Datalog Program. The DES system works by requiring the users answer questions about the validity of the partial results obtained for some sub-queries. If a buggy answer is found by the user, its associated portion of the program is pointed out as buggy. Another approach, more closely related to ours, tries to follow the computation model to find bugs. Existing proposals [20][35] are mainly based on a variant of proof trees inspired by SLD resolution. SLD resolution is a sound and complete procedure for Horn clauses, and it is the main computation procedure used in Prolog. In our setting, we deal with a MX task on a language with almost full first-order logic and inductive definitions. Thus, the use of SLD resolution is not a feasible approach. Besides the mentioned approaches which rely on computation model of the programs, authors in [24] use a translation from Answer-set programming (ASP) to natural language. Answer-set programming (ASP) is a well-known declarative programming paradigm based on stable model semantics. This approach eases the reading of an ASP program and thereby helps program verification.

In the context of the first-order world, the Alloy Analyzer [16], a bounded testing tool for first-order logic with relational calculus, has enjoyed considerable success lately [6]. The Alloy Analyzer translates Alloy specifications to an intermediate language KodKod [34] first, then the KodKod model finder solves the Allot specifications by reducing it to an SAT problem. The Alloy Analyzer works as an automatic counter-example generator, when the user simulates a partial program, the analyzer returns examples immediately that suggest a new improvement to be made. Alloy's success inspired us to develop the LogicQL Debugger. Our debugger translates the LogicQL programs to an intermediate IDP language first, and then the IDP system solves the LogicQL programs by applying its grounder and its solver MiniSAT(ID).

2.6 Conclusions and Future work

We have presented a tool for debugging LogicQL programs. The key idea is to refine the LogicQL program by exploring small satisfying models for the program under the small scope hypothesis [16]. Our tool provides a fully automatic simulation of possible models that gives immediate feedback. Moreover, a preliminary GUI is provided so that it would require no specialized knowledge of the IDP system, thus making the debugger accessible to all LogicQL users.

We have illustrated the incremental development of the LogicQL code by a process of developing LogicQL programs for automatic software distribution. The process demonstrated in this chapter is closely correlated to development for a large-scale real-world example. The prototypical LogicQL debugging system has been implemented, while we have presented a tool for LogicQL only, it is not difficult to extend the debugger to support other Datalog-like languages.

In future work, we plan to extend the debugger to support a richer set of LogicQL languages, including but not limited to Aggregations and float number supports. Additionally, further extension with the support of comparative debugging, which can automatically track an error between executions of two similar programs, would also be useful.

// Chapter 3

System Resilience Problem

Many systems may fail in an unexpected way after some "unpredictable" events that are not only rare but also have a significant impact on the system. These kind of events are usually called X-events [8]. For example, the 3.11 earthquake of Japan in 2011 brought a tsunami 14 meters high, 8 meters over the anticipated height. The consequence was devastating: roughly twenty-thousands people died and more than two hundred thousand people lost their homes due to the three nuclear reactor meltdowns caused by the tsunami. We recognize that X-events do happen, but it is unrealistic to design a system that claims to be unbreakable no matter what happens. We can, however, assess the system's ability to withstand large perturbations and what is the cost of recovering from the damage caused by these perturbations. We denote these properties as the resilience of the system [22].

The concept of resilience was first examined in the early 1970s [23] and its conceptual meanings has been defined in numerous ways. The common overlapping definition of resilience is the ability to recover from X-events. In the context of man-made systems, resilience is defined as "the ability to maintain its core purpose and integrity in the face of dramatically changed circumstances" [30]. In [30], the authors formalize the resilience problem based on the SR-model, a novel model for modeling systems that can change dynamically over time. Then, four SR-model properties that are believed to be central to the idea of resilience are defined:

- Resistance: The ability to maintain some underlying costs under a certain threshold.
- Recoverability: The ability to recover to a baseline of acceptable quality as quickly and inexpensively as possible.
- Functionality: The ability to provide a guaranteed average degree of quality for a period of time.
- Stabilizability: The ability to avoid undergoing changes that are associated with high transitional costs.

The next step in the development of the SR-model framework is the problem of evaluating system resilience for the SR-model. In this chapter, we take the initial step towards the problem, namely the algorithms for evaluating each of the four resilience properties. Given the transdisciplinary nature of this topic, we present a fully GUI-based software implementation for solving SR resilience problem. The software does not require any specialized expertise from the system user, thus making the technology for solving system resilience problems accessible to general users.

This chapter continues with a formal definition of the SR-model, which is addressed in this work. Section 3.2 will then unveil the computational complexity of evaluating system resilience properties. Given the fact that we are potentially trying to solve an NP search problem, we present our implementation based on Gecode, which provides a parallel constraints optimization solver with state-of-the-art performance, in Section 3.3.

3.1 SR-model

SR-model is a dynamic system model based on constrained systems, which are similar to the systems in Constraint Optimization Problems (COP) [10]. A *system* must include variables: objects or items that can take on a variety of values. The set of possible values for a given variable is called its domain. For example, for a system modeling transportation logistics, we may choose to see the methods of transportation as our variables, each with the same domain, which is the number of cargoes to be shipped. A *configuration* is an assignment that sets a value for each of the variables in a system. There is a *cost* associated with each configuration, which acts as an additional constraint inherent to the system.

In the formal definition of a system, we are given a countable set of variables $\mathcal{X} = \{x_1,x_2,...\}$ and a countable set D. These fixed sets form the language on which any system is defined.

Definition 8 (System)
A system is a tuple S= $<$ X, dom, c$>$ where:

- X $\subseteq \mathcal{X}$ is a set of variables,
- dom is a domain function which maps each variable $x_i \in$ X with a finite set dom(x_i) $\subset$ D,
- c is a cost function mapping from Ω(S) to R^+, where Ω(S) is the set of all assignments associating each $x_i \in$ X with dom(x_i) .

An element of Ω(S), i.e., an assignment to all variables in S, is called a configuration of S and is denoted by α. $\alpha(x_i)$ denote the value of x_i with respect to domain dom(x_i). The pair (S,α) is called a system state.

For example, we consider a university of three campuses. It is necessary to assign support staff to each of the campuses to ensure the campuses function properly. The cost of each configuration depends on two parameters: the cost of the support staff, and the level of (positive) impact of staff. The assignment of the support staff is governed by the system's controller whose goal is to minimize the cost per impact.

Example 3

The university under consideration can be modelled as the system S= < X, dom, c> where:

- X = {B, S, V} where B, S, and V associate the number of staffs assigned to campus B, campus S, and campus V respectively.
- dom(B) = dom(S) = dom(V) = {L, M, H}. Each of these values represents the level of staff assigned to each campus.
- $$c_{cost}(\alpha(V)) = \begin{cases} 2 & \text{if } \alpha(V)\text{=L} \\ 4 & \text{if } \alpha(V)\text{=M} \\ 6 & \text{if } \alpha(V)\text{=H} \end{cases}$$

$$c_{cost}(\alpha(B)) = c_{cost}(\alpha(S)) = \begin{cases} 1 & \text{if } \alpha(B) \text{ or } \alpha(S)\text{=L} \\ 2 & \text{if } \alpha(B) \text{ or } \alpha(S)\text{=M} \\ 3 & \text{if } \alpha(B) \text{ or } \alpha(S)\text{=H} \end{cases}$$

$$c_{impact}(\alpha(V)) = \begin{cases} 2 & \text{if } \alpha(V)\text{=L} \\ 3 & \text{if } \alpha(V)\text{=M} \\ 3 & \text{if } \alpha(V)\text{=H} \end{cases}$$

$$c_{cost}(\alpha(V)) = \begin{cases} 2 & \text{if } \alpha(V)\text{=L} \\ 4 & \text{if } \alpha(V)\text{=M} \\ 6 & \text{if } \alpha(V)\text{=H} \end{cases}$$

$$c = \frac{\sum_{x \in B,S,V} \quad c_{cost}(\alpha(x))}{\sum_{x \in B,S,V} \quad c_{impact}(\alpha(x))}$$

Since we have 3 variables in our system and each variable has a 3-value domain. There are 27 elements in the set of all configurations: Ω(S). For instance, $\alpha = \{\alpha(B) = H,\ \alpha(V) = M,\ \alpha(S) = L\}$ represents the high level of staff which is assigned to campus B, medium level to campus V and low level to campus S respectively. The cost of this configuration is:

$$c = \frac{3+2+2}{3+2+2} = 1$$

In the SR model, we consider a special type of dynamic system called Discrete Event System (DES) [27]. Our systems are subject to change with respect to a discrete representation of time. That is to say, this system can change only at discrete instants of time and not between two-time steps. We assume that the period between any pair of consecutive snapshots remains the same. For example, in Example 3, we may assume one-time step represents one day. By making this assumption, we can model the system's change from one-time step to the next one that reflects the actions performed by the system's controller and the consequences of an exogenous event.

Now we are ready to formally define *dynamic systems.* This graph-like structure allows us to represent the evolution for a system. For each system in the dynamic system, there is a set of actions that can be performed by the system's controller, but the consequence of the chosen action may contain non-deterministic effects that depend on the exogenous event.

Definition 9 (Dynamic System)
A Dynamic System (DS) : is a tuple DS= <S_0, $\mathcal{S}$, $\mathcal{A}$, poss, Φ_A> where:

- S_0, where $S_0 \in \mathcal{S}$, is the "initial" system,
- $\mathcal{S}$ denotes the set of all possible systems,
- $\mathcal{A}$ is a non-empty set of all possible actions,
- poss gives a (non-empty) set of possible moves for each system ,
- Φ_A is a partial function from $\mathcal{S}$ X $\mathcal{A}$ to $2^{\mathcal{S}}$, Φ_A specifies how a given system may change in response to some actions.

Note that the consequences of the exogenous events are explicitly given. A dynamic system can be represented as a graph where each node represents a system, and each edge represents the (potential) consequence of some move that would transform the current system into another one. Thus, we sometimes use *dynamic system graph* to refer to a Dynamic System (DS).

Example 4

Figure 3.1 represents a simple dynamic system with $\mathcal{S} = \{S_0, A, B, C, D\}$ and $\mathcal{A} = \{noop, a_0, a_1, a_2, a_4, a_5\}$. It's worth noting that we do not require that every action be executed on every system in the DS. Suppose the controller chooses an action a_1 at the initial system S_0, in the next time step, we may fall into either system A or system D depending on exogenous events.

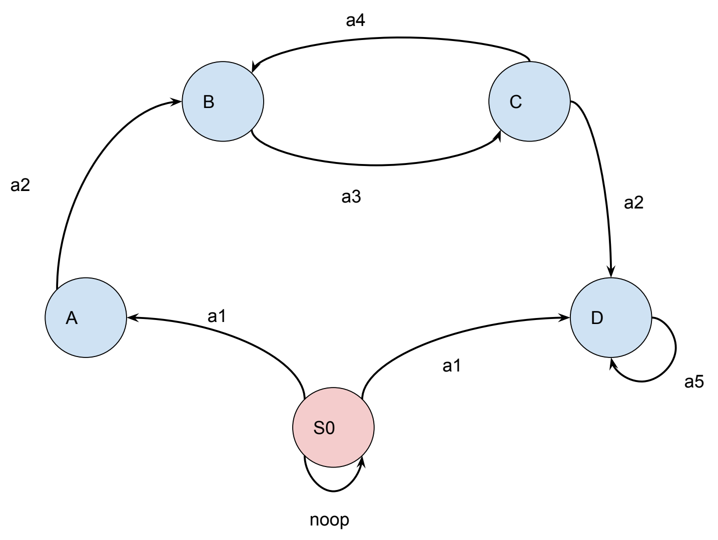


Figure 3.1: Sample Dynamic System Graph

3.1.1 Resilience of the SR-model

In the case of resilience properties for SR-models which we are about to define, those properties evaluate the behavior of the system that is specified by *system trajectories*. A system trajectory is simply the list of systems that occurs during a particular dynamic system path.

Definition 10 (System Trajectory)
A system trajectory ST is a (possibly infinite) sequence of systems (S_0, …, S_n, …).

Each $S_i \in ST$ represents the system (as defined in Definition 8) at time step i whereas the first element for every ST is S_0, the initial system defined in the SR-model. Adjacent systems in the ST are not required to satisfy any relationship.

Resilience properties are not defined based on system trajectories but on the *state trajectory*. Intuitively, a state trajectory is a system trajectory ST together with a configuration for each system in ST. Once again, adjacent states are not required to satisfy any relationship.

Definition 11 (State Trajectory)
A state trajectory SST is a (possibly infinite) sequence of system states { (S_0, α_0), (S_1, α_1), …}.

Example 5

Let us consider a ST with two systems: S_0 and S_1 where S_0 is our university example defined in Example 3. S_1 is similar to S_0, but

$$c_{1_{impact}}(\alpha(B)) = \frac{2}{3} \quad c_{0_{impact}}(\alpha(B)) = \frac{2}{3} \quad c_{impact}(\alpha(B))$$

This is to model the situation that with new buildings on the B campus opening at time step 1, the impact per support staff is reduced. Then, an example of a state trajectory is { (S_0,$\{\alpha(B) = H, \alpha(V) = M, \alpha(S) = L\}$), ($S_1$,$\{\alpha(B) = H, \alpha(V) = M, \alpha(S) = L\}$)}. That is while the configuration between two systems remains unchanged, the cost to the second system state was raised to:

$$c_1 = \frac{3+2+2}{\mathbf{2}+2+2} = 1.17$$

compared with c_0=1 in Example 3

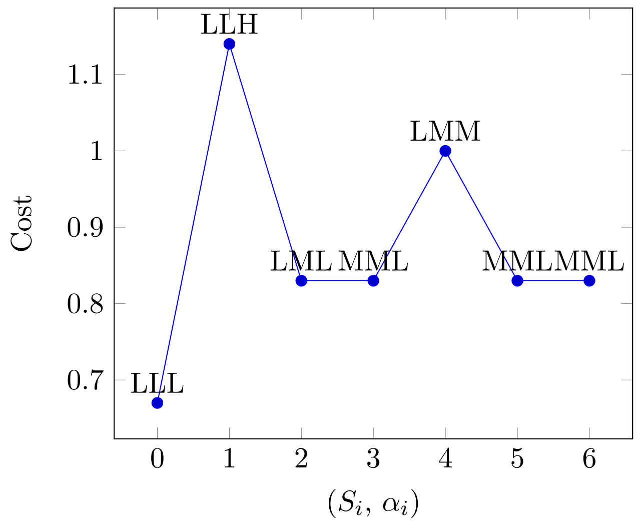
Now we are ready to formally define the resilience proprieties:

Resistance

Definition 12 (Resistance)
Resistance: given a state trajectory SST{ (S_1, α_1), (S_2, α_2), ...} and a non-negative number L, SST is L-resistant if for each state (S_i, α_i) $\in$ SST, $c_i(a_i) \leq L$

A state trajectory is L-resistant if the cost of each system state is always below the threshold L. That is to say, we guarantee the system never exceeded a certain degree of quality. If there is a system state where the cost is greater than L, then we say this SST is not L-resistant.

Example 6
For simplicity, we denote $\alpha = MML$ as an abbreviation for $\alpha = \{\alpha(B) = H, \alpha(V) = M, \alpha(S) = L\}$. The following figure illustrates an SST for the university staff allocation Dynamic System DS where there is only one system S_0, as defined in Example 8, in the DS. The dynamic system graph for this DS is simply a node with a self-loop.



This SST is 1-resistant but not 2-resistant.

Recoverability

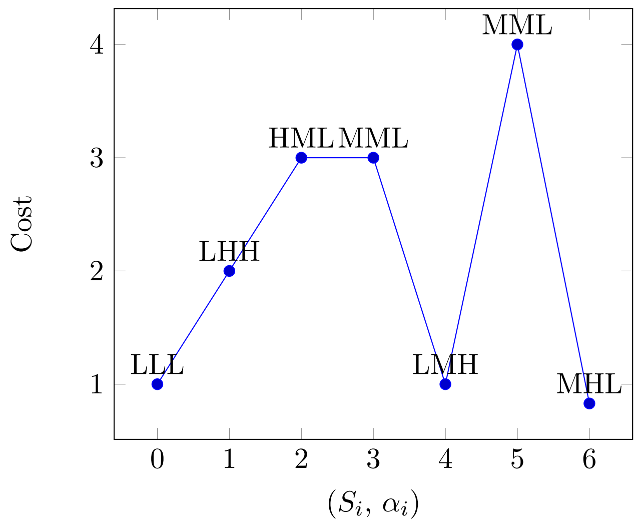
Definition 13 (Recoverability)
Recoverability: Given a SST { (S_1,α_1), (S_2,α_2), $\ldots$} and two non-negative numbers P and Q, SST is $< P, Q > -$recoverable if for any subsequence $\{(S_a,\alpha_a)$, $\ldots$, $(S_b,\alpha_b)\}$of SST with $\forall i \in [a, b]$, $c_i(a_i) > P$, the following conditions are satisfied:
(i) $\sum_{i=a}^{b}(c_i(a_i) - P) \leq Q$, and
(ii) The system state (S_{b+1},α_{b+1}) after (S_b,α_b) has cost $c_{b+1}(a_{b+1}) \leq P$.

Recoverability is a quantitative formulation of the ability to spring back to a former position or shape. Similar to Resistance, this two-parameter property exploits the cost of each system state in the given SST. The first parameter P acts as the threshold for "normal" cost. If the cost is always kept under P, then this SST is vacuously recoverable. On the other hand, a system state with the cost greater than P does not make this SST "unacceptable". The second parameter Q captures the maximum amount of extra cost that is necessary for the system state to get back to a "normal" state. There are two cases when we say a SST is not $< P, Q > -$recoverable:

- The system never comes back to a state with a cost no greater than P.
- The system does come back to a "normal" state, but the cumulative extra cost is greater than Q.

We illustrate the property using our university example:

Example 7



Suppose we want to check if this SST is $< 1, 5 >$ −recoverable. The first step is to find all sub-SST where every element in the subset has a cost greater than 1. In this example, there are two subsets satisfying this condition:

1. $\{(S_1, \alpha_1), (S_2, \alpha_2), (S_3, \alpha_3)\}$ with corresponding costs {2,3,3}.
   This sub-SST has a cumulative extra cost of 5.
2. $\{(S_4, \alpha_4)\}$ with corresponding cost {4}. This sub-SST has a cumulative extra cost of 3.

Since both sub-SSTs have a subsequent "normal" state in the SST and the extra cost never exceeded 5. We say this SST is indeed $< 1, 5 >$ −recoverable. However, this SST is not $< 0.5, 5 >$ −recoverable as there is no system state with the cost below 0.5.

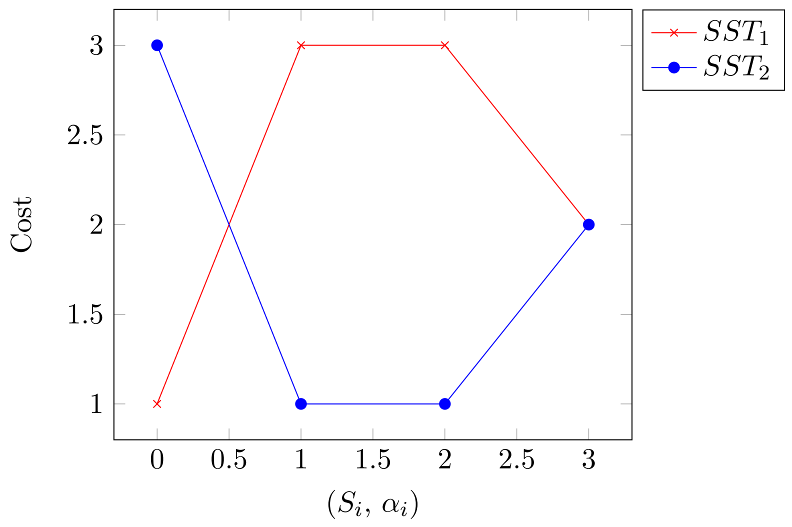
Functionality

Definition 14 (Functionality)
Functionality: Let avg(SST) denote the average cost of the given SST, then the SST is said F-functional if avg(SST) ≤ S, where S is a non-negative number.

$$avg(SST) = \begin{cases} \sum_{0}^{k-1} \dfrac{C_i(\alpha_i)}{k-1} & \text{if SST has a finite size of k} \\ \lim\limits_{k \to +\infty} \sum_{i=0}^{k} \dfrac{C_i(\alpha_i)}{k} & \text{otherwise} \end{cases}$$

This property requires for a state trajectory SST that the average cost be kept under some threshold f. This property provides a guaranteed average degree of quality in the long run while recovering from the X-events that may destabilize the system. We illustrate the distinction between functionality and resilience in the following example.

Example 8



Both SST_1 and SST_2 are 3-resistant. However, we have $avg(SST_1) = \frac{1+3+3+2}{4} = 2.25$, and $avg(SST_2) = \frac{3+1+1+2}{4} = 1.75$. Therefore, SST_2 is 2-functional whereas SST_1 is not.

Stabilizability

Definition 15 (Stabilizability)
Stabilizability: Given a distance function d((S_i,α_i), (S_j,α_j)) that calculates the distance (the penalty of changes) between system states. A SST is said K-stabilizable if for every $i \in \{2, 3, ...\}$ d((S_{i-1},α_{i-1}),(S_i,α_i)) $\leq$ K.

Stabilizability evaluates the ability of a system to avoid changes in the configuration while maintaining the integrity of the system. The quantitative evaluation is achieved by introducing a distance (penalty) between two system states. The distance is calculated based on the system states and the distance function provided by the user. We do not impose any constraint on the distance functions as long as they can be calculated efficiently. Let us demonstrate this property within our university example:

Example 9
Suppose the distance function for our university example is defined as

$$d((S_i, \alpha_i), (S_j, \alpha_j)) = \sum_{x_i \in X} d_{x_i}(\alpha_i(x_i), \alpha_j(x_i))$$

where for every $x_i \in X = B, V, S$

$$\begin{cases} d_{x_i}(\alpha_i(x_i), \alpha_j(x_i)) = 0 & if \alpha_i(x_i) = \alpha_j(x_i) \\ d_{x_i}(\alpha_i(x_i), \alpha_j(x_i)) = 1 & if (\alpha_i(x_i), \alpha_j(x_i)) \in \{LM, ML, MH, HM\} \\ d_{x_i}(\alpha_i(x_i), \alpha_j(x_i)) = 3 & if (\alpha_i(x_i), \alpha_j(x_i)) \in \{LH, HL\} \end{cases}$$

Then for system states (S_1,MML) and (S_2,LHH), the distance is:

$$d((S_1, MML), (S_2, LHH)) = 1 + 1 + 3 = 5$$

3.1.2 Determine the Resilience of a Dynamic System

Before we extend the resilience properties defined in the previous section for system state trajectories to dynamic systems, we define the set of state trajectories that are realizable in a given Dynamic System.

Definition 16 (Realizable System States Trajectories)
Given a dynamic system DS = $<S_0, \mathcal{S}, \mathcal{A}$, poss, $\Phi_A>$, a state trajectory SST = { (S_0,α_0), (S_1,α_1), ... } is said to be *realizable* in DS if for every $(S_i, \alpha_i) \in SST$ with $i \geq 1$, there is an action $a \in poss(S_{i-1})$ such that $S_i \in \Phi_A(S_{i-1}, a)$ and α_i is a proper configuration for the system S_i

To address the issue of control of the dynamic system, we introduced the notion of *strategy*. A strategy is the choice made by system's controller, which lies in two level of controls in a dynamic system:

- 1. The first level consists in tuning the system's configuration. That is to say, for a given system S, the configuration of S is determined by the controller.
- 2. The second level decides which action can be performed on a system to cope with system's changes over time. For example, the cost function may change over time.

What is not under the control of the system's controller is the non-determinism consequence of the each chosen action. A strategy consists of choosing a specific configuration and specifying an action for every system in DS. Formally, a strategy of a DS is:

Definition 17 (Strategy of a Dynamic System)
A strategy is a set of (action,configuration) pairs of the form
$\{(action_0, configuration_0), \ldots, (action_n, configuration_n), \ldots\}$
that satisfies the following condition:

- For every system $S_i \in DS$, we have one (action,configuration) pair $(action_i, configuration_i)$ for S_i.

- $action_i$ must be a valid choice of action for S_i,and $configuration_i$ is a proper configuration for S_i.

From this definition, we are now ready to define the Resilience of a Dynamic System:

Definition 18 (Resilience of Dynamic System)
Let DS be a dynamic system and STR be the set of all possible strategies for this DS. Then we say

- DS is x-resistant if x= min$\{$max$\{L \in R \mid$ L is the resistance of SST realized by strategy st $\}|st \in STR\}$
- Given a number P, DS is $< P, x >$ $-$recoverable if x= min$\{$max$\{Q \in R \mid$ <P,Q> is the recoverability of SST realized by strategy st $\}|st \in STR\}$
- DS is x-functional if x= min$\{$max$\{F \in R \mid$ F is the functionality of SST realized by strategy st $\}|st \in STR\}$
- DS is x-stabilizable if x= min$\{$max$\{S \in R \mid$ S is the Stabilizability of SST realized by strategy st $\}|st \in STR\}$

Intuitively, given a specific strategy st, we use the aggregation function max to find the worst case, and we can provide a guarantee for the resilience properties by setting the value of x to capture this worst case scenario for the strategy st. Then, the use of aggregation function min that allows us to figure out the best possible case one could get by adopting the appropriate strategy. In sum, we want to determine an optimal strategy such that we get the lowest threshold value for a resilience property in any possible uncontrollable scenario.

3.2 On the Complexity of Evaluating Resilience Properties

In this section, we unveil the complexity of evaluating resilience properties defined in the previous section by presenting algorithms for each of the resilience properties. Each of the algorithms takes a dynamic system and solves one of the resilience problems defined in Definition 18.

Proposition 1
For Resistance, Functionality and Recoverability, to obtain the lowest threshold value in any possible uncontrollable scenario, it is always right to choose the configuration with the lowest cost for each system.

Proof. (For Resistance) Suppose one of the system S_i is not configured with the lowest cost, say, $c_i > c_{i_{min}}$ and an optimum strategy is determined that gives us the lowest threshold value x. Then there is a set of system state trajectories (SSTs) realized by the strategy such that every $sst \in SSTs$ is x-resistant. There are two cases for system S_1:

1. S_i does not appear every $sst \in SSTs$. Hence, changing the configuration of S_i to a lower cost one can result in an equally good answer (threshold value). Since the value of resistance is proportional to the cost of systems among the sst, the cost of S_i will not affect the resistance of the sst without S_i.

2. S_i is in every $sst \in SSTs$. Then changing the configuration of S_i to a lower cost one would result in a better answer(threshold value), which contradicts the assumption of the optimum strategy that gets the optimal threshold value x.

Therefore, it is always right to choose the configuration with the lowest cost for the case of resistance.

We can apply a similar argument for the Functionality and Recoverability since the threshold value is also proportional to the cost of the system along the realizable SSTs. □

Given Proposition 1, the task of evaluating resilience properties can be reduced to determine an action to make for each system $s \in S$. And thus, the notation of Dynamic systems can be simplified to:

A Dynamic Systems DS is a tuple $< S_0, G >$, where:

- S_0 is the "initial" system state (node),
- $G = (S, E)$ denotes the *dynamic system graph* where each vertex $s \in S$ represents a system state and each edge $e \in E$ represents the consequence of an action that would transform the current system (node) into another one. Additionally, each $e \in E$ is labeled with an action and a cost of changes between system states.

3.2.1 A Polynomial Algorithm for Resistance

Recall that L-Resistance of DS is defined as for any realisable system state sequence(SST){ (S_1, α_1), (S_2,α_2), ...}, the DS is L-resistant if for each state (S_i, α_i) ∈ SST, $c_i(a_i) \leq L$.

The problem of deciding the resistant of the dynamic system DS, denoted as is L-Resistance of DS, is the following:

Instance: A dynamic system graph $G = (S, E)$ and S_0
Question: What is the best choice of action for each system so that we achieve the lowest threshold value for the resistance property in any possible uncontrollable scenario?

We present an algorithm which, given a dynamic system graph, output the choice of action for every system state in the graph. There are three kinds of labels in the algorithm:

- Visited label, this label determines whether a system is already been visited,
- L,a-Resistant label denotes that if this system chooses action a, then every SST start from this system is L-Resistant

- L-Resistant label denotes that the action to choose for this system has been determined and every SST start from this system is L-Resistant

Observe that the threshold value L is dependent on the system state(node) with the highest cost, the algorithm works by incrementally mark the highest cost node that a (system, action) pair can reach. Whenever a system s gets a k,a-Resistant label for every action in s, the algorithm selects the action that results in the lowest value of L.

The algorithm proceeds as follows:

1. Initially, every system in S is unvisited with no label assigned to it.

2. Select an unvisited system $s \in S$ with the highest cost c, mark s as a visited. If there is no l-Resistant label assigned to s, then label s as c-Resistant.

3. For each edge (s_i, s) with an action a, If there is no l,a-Resistant label on s_i, mark s_i as c,a-Resistant. If there is l,a-Resistant label for every possible action on s_i, then select the action that has the lowest value of l (say l_{min}), mark s_i as l_{min}-Resistant and change the cost of s_i to l_{min}

4. repeat step 2 and 3 until all systems are labeled "Visited".

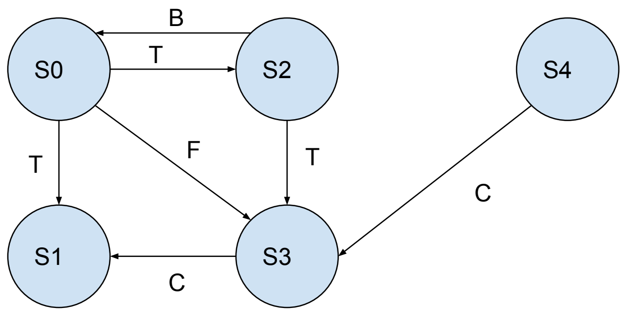
This algorithm not only determines the action each system must take in order to achieve the lowest resistance, but also reports L-Resistant for each system. Every node is visited exactly once in the main loop and at every iteration of the main loop, the algorithm traverses an entire graph in the worst case. Therefore, this algorithm runs in O$(|S|(|S| + |E|))$ time.

3.2.2 A Polynomial Algorithm for Recoverability

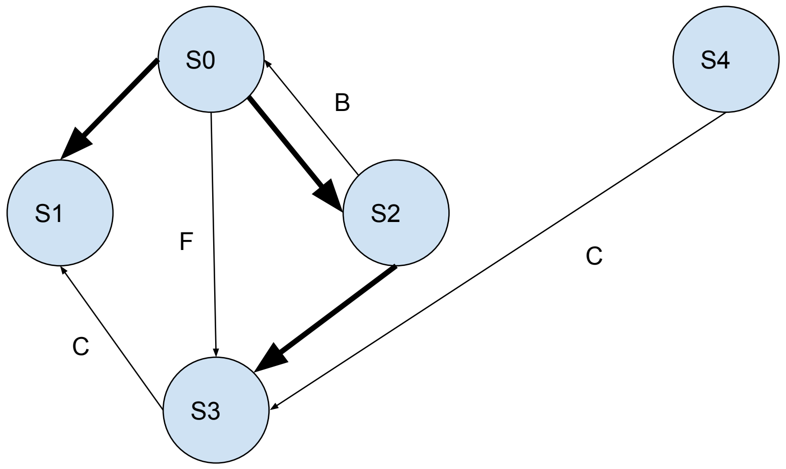
Recoverability is defined as: for any realisable system state sequence(SST)$\{(S_1, \alpha_1), (S_2, \alpha_2), \ldots\}$, the DS is $< P, Q > -$recoverable if for any subsequence $(S_a, \alpha_a), \ldots, (S_b, \alpha_b)\}$of SST with $\forall i \in [a, b]$, $c_i(a_i) > P$, the following conditions are satisfied:
(i) $\sum_{i=a}^{b} (c_i(a_i) - P) \leq Q$, and
(ii) there is a system state (S_t, α_t) after (S_b, α_b) such that $t > b$ and $c_t(a_t \leq P)$.

Assume that P is a given number and we have a new dynamic system graph $\hat{G}$ with the cost of each system updated to ((the Original cost) - P). The problem of deciding the recoverability of a dynamic system DS, denoted as is $< P, Q >$-recoverable of DS, is the following:

Instance: A dynamic system graph $\hat{G} = (S, E)$ and S_0
Question: What is the best choice of action for each system so that we achieve the lowest threshold value for Q in any possible uncontrollable scenario?



(a)



(b)

Figure 3.2: (a) An example of a graph with exactly one back edge (S2,S1), one forward edge (S0,S3), one cross edge (S3,S1) connecting a node to another node in the same depth-first tree and one cross edge (S4,S3) connecting a node to another node in a different depth-first tree. (b) A redraw of part (a) in a depth-first tree form with thicker tree edges.

We use three kinds of labels in this algorithm:

- system color (WHITE, GRAY or BLACK). WHITE denotes the system has not been discovered; GRAY denotes the system is in the process of exploring edges; BLACK denotes that we have finished exploring all edges of the system.
- $< P, Q >$,a-recoverable denotes that if this system chooses action a then every SST start from this system is $< P, Q >$-recoverable.
- $< P, Q >$-recoverable label denotes that the action to choose for this system has been determined and every SST start from this system is $< P, Q >$-recoverable. Every system that is labeled BLACK has this $< P, Q >$-recoverable label as we have explored all outgoing edges of the system.

To evaluate the recoverability efficiently, we make use of an important property of depth-first search (DFS) on graphs that can be used to classify edges of the input dynamic graph. The idea is that when we explore an edge (s,v) using depth-first search, the color of system(node) v can be used to indicate four different kinds of edge types:

1. If v is WHITE, then (s,v) is a tree edge. That is to say, system(node) v is first discovered by edge (s,v).
2. If v is GRAY, then (s,v) is a back edge. For back edge (s,v), system(node) s is connected to an ancestor system v in a depth-first tree, which means we found a cycle $s - v \rightsquigarrow s$.
3. If v is BLACK, the (s,v) is either a forward or cross edge. A forward edge is an edge (s,v) connecting a system s to a descendant v in a depth-first tree. A cross edge (s,v) are edges that cross between nodes in either the same depth-first tree or different depth-first tree as long as one is not an ancestor of the other one.

Figure 3.2 provides an example of different edges labels of a graph. Edges label T, B, F, C indicates tree edge, back edges, forward edge and cross edge respectively. Figure 3.2(b) shows a redrawing of Figure 3.2(a) in a depth-first tree form.

The Depth-first search-based labeling algorithm for recoverability, DFS-LABEL-R($\hat{G}$), works by recursively labeling the threshold value Q for each (system,choice) pair from the DFS and converging to a conclusion of the best choice of action for system s after every outgoing edge of s has been explored. When an edge (s,v) is explored, the threshold value Q is determined based on the type of edge (s,v). If this edge is a back edge, that is, this edge leads to a cycle, then a sst of infinite length can be formed and the value Q is calculated on this SST using the definition of recoverability stated above. Note that if every system in the cycle has a cost greater than 0, then Q $= \infty$ as Q equals the sum of the cost of an infinite subsequence of SST. If this edge is either a forward or cross edge, then this edge lead to a system v where the best choice has already determined. The algorithm then tries to propagate a SST using the choice of action for system v and calculates the value of Q accordingly.

Specifically, DFS-LABEL-R($\hat{G}$) is defined as follows:

DFS-LABEL-R($\hat{G}$)

1. Initially, every system $s \in S$ is WHITE.
2. For each system $s \in S$, if s is WHITE then call DFS-LABEL-VISIT-R($\hat{G}$,s)

DFS-LABEL-VISIT-R($\hat{G}$,s)

1. Change the color of s to GRAY.
2. For each edge $(s, v) \in E$ with action a
    - If v is WHITE, then call DFS-LABEL-VISIT-R($\hat{G}$,v).
    - If v is GRAY, then (s, v) is a back edge and we have found a cycle $v \rightsquigarrow s - v$.
        - If there is no system with cost less or equal to 0 in the cycle $v \rightsquigarrow s - v$, then for every edge ((s_i, s_k)) with action a_i, label s_k as $< P, \infty >, a_i$-recoverable
        - Else If there is a system with cost less or equal to 0 in the cycle $v \rightsquigarrow s - v$, then calculate the max possible value of Q in this cycle. After that for every edge ((s_i, s_k)) with action a_i, label s_k as $< P, max(Q) >, a_i$-recoverable
        - Note that if there already exists a label $< P, Q_1 >, a_i$-recoverable on s_k when we want label s_k as $< P, Q_2 >, a_i$-recoverable, the label $< P, Q_1 >, a_i$-recoverable is replaced with $< P, Q_2 >, a_i$-recoverable if and only if $Q_2 > Q_1$
    - If v is BLACK, then form a path s-v-v_1...-v_n such that every system between s and v_n is BLACK and v_j is the successor state of v_i in the path only if
      i: v_i chooses action a_i
      ii: there is an edge (v_i,v_j) with label a_i
      iii: both systems are $< P, Q >$-recoverable
      We continue propagating the path until either one of the following condition is satisfied:
        - v_n is GRAY, i.e., there is a cycle $s - v \rightsquigarrow s$. We perform the same computation as in Step 2.2 when v is GRAY.
        - v_n is BLACK and v_n is already been visited in the path, then there is a cycle $v_n \rightsquigarrow v_n$ in this path. We compute the max value of Q for SST $s \rightsquigarrow v_n \rightsquigarrow v_n$ and label s as as $< P, max(Q) >, a$-recoverable

3. Among all $< p_i, q_i > a_i$-recoverable labels on system s, choose the action a_j that has the lowest value of Q (say Q_{min}), mark s as $< P, Q_{min} >$-recoverable and change the color of s to BLACK.

At each step of DFS, DFS-LABEL-R($\hat{G}$) traverse an entire graph in the worst case. Since DFS takes time O($(|E| + |S|)$)), this algorithms runs in O($(|E| + |S|)^2$)) time.

3.2.3 A Polynomial Algorithm for Functionality

Functionality of DS is defined as for any realisable system state sequence(SST)$\{(S_1,\alpha_1), (S_2,\alpha_2), \ldots\}$, the DS is f-functional if avg(SST) $\leq$ S. For infinite SST, avg(SST) is equal to the average cost of all systems in a cycle.

The problem of deciding the Functionality of a dynamic system DS, denoted as is f-functional of DS, is the following:

Instance: A dynamic system graph $G = (S, E)$ and S_0
Question: What is the best choice of action for each system so that we achieve the lowest threshold value f in any possible uncontrollable scenario?

The algorithm proposed in this section can be viewed as a simplified version of the Depth-first search-like labeling algorithm DFS-LABEL-R($\hat{G}$) for Recoverability. We no longer check the condition of costs on the SSTs and the value of f is calculated using the average cost of STTs.

Once Again, we have three kinds of labels:

- system color (WHITE, GRAY or BLACK). WHITE denotes the system has not been discovered; GRAY denotes the system is in the process of exploring edges; BLACK denotes that we have finished exploring all edges of the system.
- f,a-functional denotes that if this system chooses action a then every SST start from this system is f-functional
- f-functional label denotes that the action to choose for this system has been determined and every SST start from this system is f-functional

Depth-first search-like labeling algorithm works as follows:
DFS-LABEL-F(G): same as DFS-LABEL-R(G).
DFS-LABEL-VISIT-F(G,s)

1. Change the color of s to GRAY.
2. For each edge $(s, v) \in E$ with action a
    - If v is WHITE, then call DFS-LABEL-VISIT(G,v).

- If v is GRAY, then (s, v) is a back edge and we have found a cycle $v \rightsquigarrow s - v$.
    - Calculate average cost (say f_i) of the cycle, and then for every edge ((s_i, s_k)) in the cycle with action a_i, label s_k as f_i-functional.
    - Same as the recoverability case, if there already exists a label f_1, a_i-functional on s_k when we want label s_k as f_2, a_i-functional, the label is replaced with f_2, a_i-functional if and only if $f_2 > f_1$
- If v is BLACK, then form a path s-v-v_1...-v_n such that every system between s and v_n is BLACK and v_j is the successor state of v_i in the path only if
  i: v_i chooses action a_i
  ii: there is an edge (v_i,v_j) with label a_i
  iii: both systems are f_i-functional
  We continue propagating the path until either one of the following condition is satisfied:
    - v_n is GRAY, i.e., there is a cycle $s - v \rightsquigarrow s$. We perform the same computation as in Step 2.2 when v is GRAY.
    - v_n is BLACK and v_n is already been visited in the path, then there is a cycle $v_n \rightsquigarrow v_n$ in this path. Suppose v_n is f_i-functional, we label s as as f_i, a-functional

3. Among all f_i, a_i-functional labels on system s, choose the action a_j that has lowest value of f (say f_{min}), mark s as f_{min}-functional and change the color of s to BLACK.

While the computation of threshold value of functionality if simpler than recoverability, DFS-LABEL-F(G) still traverses the entire graph in the worst case. Thus, algorithms also run in O(($|E| + |S|)^2$)) time.

3.2.4 Complexity of Evaluating Stabilizability

Stabilizability is defined as: given function d((S_i,α_i), (S_j,α_j)) that calculates the distance (the penalty of changes) between system configurations. A DS is said s-stabilizable if for any realizable system state sequence(SST){ (S_1, α_1), (S_2, α_2), ...}:

$$d((S_{i-1}, \alpha_{i-1}), (S_i, \alpha_i)) \leq s.$$

Proposition 2
For Stabilizability, it is not always right to choose the configuration with the lowest cost for each of the systems.

Proof. This is a trivial claim as one can define a high penalty for any changes between system configurations but no penalty if the configurations remain unchanged. For any dynamic system that has to assign different configurations to achieve the lowest cost, we can not obtain best Stabilizability by choosing the configuration with the lowest cost. □

Therefore, the problem of deciding the Stabilizability of a dynamic system DS, denoted as is s-stabilizable of DS, is the following:

Instance: A dynamic system DS
Question: What is the best strategy (defined in Section 3.1.2) for DS so that we achieve the lowest threshold value s in any possible uncontrollable scenario?

Instead of proposing an algorithm for Stabilizability, we show that even with a less expressive DS, determine whether such DS is s-stabilizable or not is NP-complete. We restrict ourselves to a group of DS where every edge is labeled with same action (i.e. label function L is no longer needed) and the distance function d is constant-time computable. For simplicity, this problem is named as restricted s-stabilizable problem.

Lemma 1: Restricted s-stabilizable Problem is in NP

Proof. Given any configuration of the system(C), our verifier will calculate and check the distance of every edge (in O($|E|$) time). For a yes instance (DS, C), our certificate is any edge has a distance less than s; our verifier will accept such a configuration, otherwise our verifier will reject the configuration. □

Lemma 2: Restricted s-stabilizable Problem is in NP-Hard

Proof. We show that 3-Coloring $<_P$ Restricted s-stabilizable Problem. Given an instance of 3-Coloring $G_{color} = (V_{color}, E_{color})$, we construct a DS=$< S_0, G, d >$ for the Restricted 0-stabilizable Problem as follows.

For each vertex in $v_i \in V_{color}$, we construct a system s_i with only one variable of domain size 3: R, G, B and add s_i to S. We let $S_0 = s_0$ since we do not care about the initial system. Next, for each edge of the form $(v_i, v_j) \in E_{color}$, we add an edge (s_i, s_j) to E. Finally the distance function d is defined as:

$$d((S_i, \alpha_i), (S_j, \alpha_j)) = \begin{cases} 0 & \text{if } \alpha_i \neq \alpha_j \\ \infty & \text{otherwise} \end{cases}$$

□

We now run our solver for Restricted 0-stabilizable Problem on DS= (S_0, G, d) and return the same result it gives. (This construction is poly-size and poly-time because G has O(m) nodes and O(n) edges.)

Proposition 3
A 3-Coloring instance is satisfiable if and only if DS= (S_0, G, d) is 0-stabilizable.

It is a trivial proposition.

Proposition 4
Restricted s-stabilizable Problem is NP-Complete

Proof. We showed that Restricted s-stabilizable Problem is in NP and 3-Coloring $<_P$ Restricted s-stabilizable Problem, therefore Restricted s-stabilizable Problem is NP-Complete. □

3.3 User Friendly Software for Solving System Resilience Problems using Gecode

In the previous section, we have shown that by choosing the configuration with the lowest cost for each system in the SR-model, we can find an optimal strategy that is as good as all others strategies for most of the resilience properties. Therefore, in our prototype implementation, we assume that each node in the dynamic graph of the SR-model is already assigned with an optimum configuration. That is, each node in the dynamic graph is now a system state rather than a system.

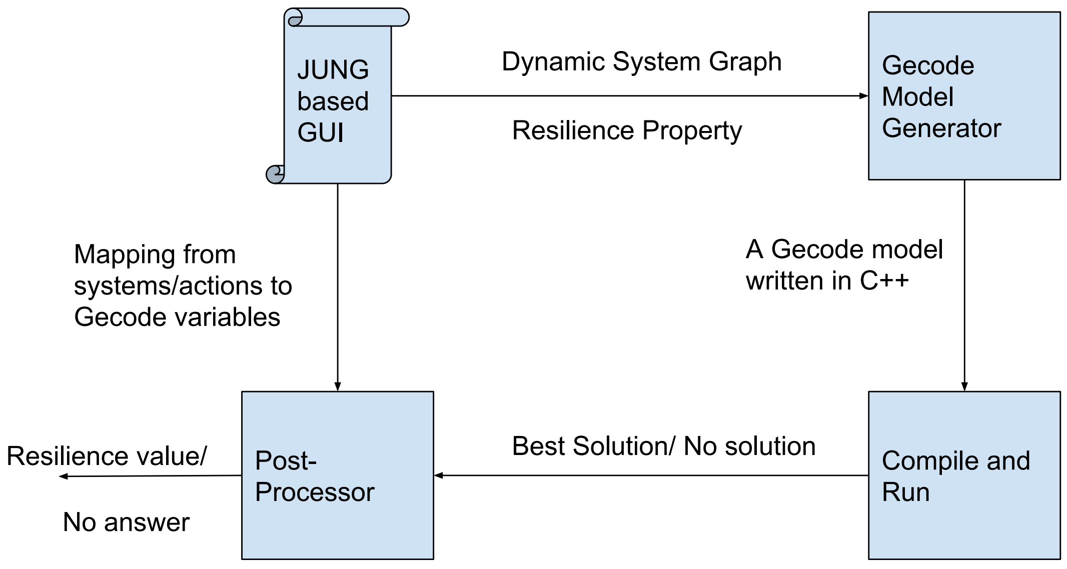


Figure 3.3: SR-solver General Solving Scheme

In this section, we present SR-solver, an integrated software for drawing SR-models and determining the resilience for SR-models. The general solving scheme for SR-solver is shown

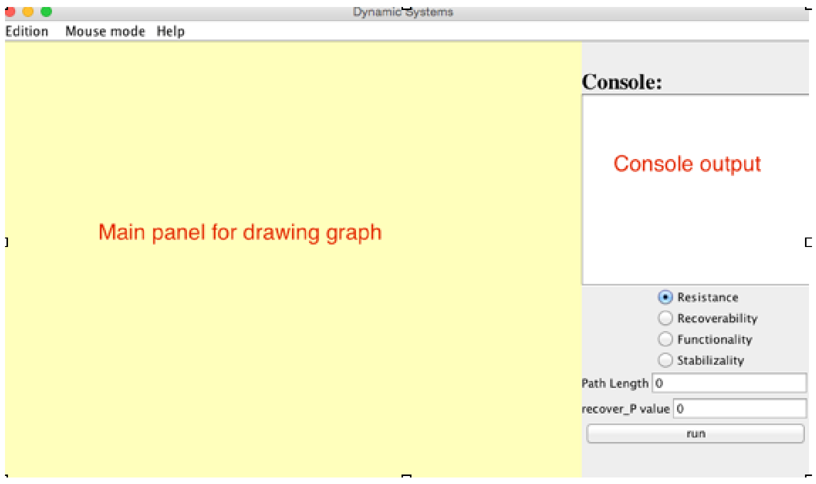


Figure 3.4: Main GUI of the SR-solver

in Figure 3.3. The GUI of SR-solver allows users to describe the dynamic graph, specify a resilience property of interest and a bond on the length of the state trajectories (SSTs). The module identified as "Gecode Model Generator" in Figure 3.3 generates the Gecode model written in C++ and then complies with the model with the Gecode library. After running the complied program, the result is sent to the "Post-Processor". If there is a solution in the result, "Post-Processor" maps the solution back to the systems and actions in the dynamic graph. SR-solver is able to compute the answers to the queries on resistance, recoverability, functionality and stabilizability. Specifically, for each query, SR-solver allows the user to add additional constraints on the realizable SSTs and to report the best strategy one can take to achieve the best resilience in the given model.

Figure 3.4 presents the main GUI window of the SR-solver, with the left half of GUI being the dynamic system graph drawing area and the right half being the control panel. The users can either load the graph from external XML files or just create the graph in the drawing area.

The software has a JUNG [33] based graphical user interface for users to draw dynamic system graphs. JUNG is a free Java software library that can be used for analysis, manipulation, and the visualization of graphs and networks. The major features of JUNG include the support of a variety of graphs, including directed and undirected graphs and hypergraphs and have the built-in mechanisms for annotating graphs as well as the metadata attached to nodes and edges in the graph.

Since the purpose of the GUI is to retrieve dynamic system graph from users in various ways, the software should realize user changes impeccably. In our implementation, we adopt a Model-View-Controller (MVC) pattern to address this issue as shown in the following:

- A **model** stores the dynamic graph G=(V, E) where V is a set of nodes (system states), E the a set of edges, together with the metadata attached to them. Nodes are extended from the basic node class in JUNG with an addition of cost metadata. Each node represents a system state in the SR-model. Edges are also extended with additional action and distance metadata.

For any nodes v_1 and $v_2 \in V$, there is an edge (v_1, v_2) with action a $\in E$ if and only if one of the consequences of applying action a on node (system state) v_1 is v_2.

- A **view** generates the graphical presentation of the graph G. JUNG provides *VisualizationViewer* for these components.

- A **Controller** manipulates the model based on user commands. Any updates to the model are immaterially updated in the view. The controller is able to achieve following operations:

  - Save the graph: the user has an option of saving the dynamic graph to a GraphML file with an extension of XML. The GraphML file keeps not only the graph, but also the position of each node/edges and the initial node.
  - Load pre-saved graph: if the user desires to open a graph saved in the external source, the user selects "Load from file" from the "Edition" menu. A dialog box appears and asks for the ".xml" file directory. The file must be a GraphML file in XML structure. When the new graph is loaded, the old one is removed if it exists.
  - Clear dynamic graph: the "Clear" button removes all nodes and edges from the drawing panel.
  - Set number of CPU cores to use: the user can set the number of CPU cores for the background solver to do parallel search in a shared memory environment. It is worth noting that the point of parallel search is not to perfectly utilize the available parallel hardware, but to make a faster search.
  - Zoom in and Zoom out: the users are able to zoom in or zoom out by using the scroll wheel.
  - Change mouse mode: there are two options of manipulating the graph. One is *Editing* mode: users can either create a new node by left-clicking on the unoccupied space or draw an edge between two nodes by "dragging" the edge out of the origin note to the destination node. Users can also update a node's or an edge's metadata and remove nodes and edges by right-clicking on the desired node or edge. The other mode is *Picking* mode: User can change the positions of a chosen node.

In sum, the GUI is intended to provide a pretty intuitive way to draw the dynamic system graph interactively. An example of dynamic graph in the GUI can be seen in Figure 3.5:

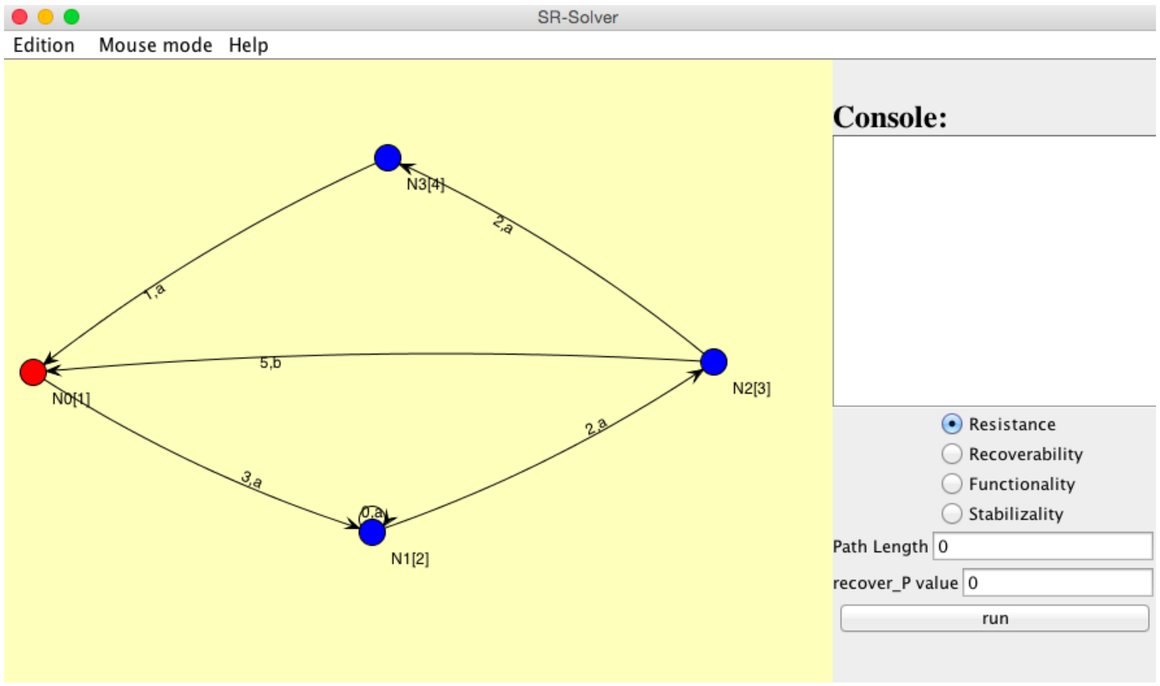


Figure 3.5: A Dynamic Graph Example

Figure 3.5 displaces a simple dynamic graph with 4 nodes (system states). The initial node is N0 with cost 1 and it's marked red. It is worth noting that while we require a predetermined configuration for each system in the dynamic graph for now, the underlining data structure and the solver supports the full definition of the SR-model. Determining the best configuration for each system can be a COP problem of its own. COP is an extension of CSP in the way that it is a CSP with a cost function that must be minimized or maximized. The COP is NP-hard, since solving it involves solving CSP which is NP-complete. Given the fact that we are potentially trying to solve an NP-complete problem, we choose an existing efficient COP solver in our implementation. Our software implements a solving engine based on Gecode [1] which provides a Constraint Optimization Problem(COP) solver with state-of-the-art performance.

There are several advantages of using Gecode as a back-end solver than other COP solvers like MiniZinc [14] or Sugar++ [31]:

- Open Source
- Rich Constraint families support: arithmetics, Boolean, ordering,...
- Portable and extendible: written in standard C++
- Efficient: won all gold medals in all categories at the MiniZinc Challenges from 2008 to 2012
- Parallel: built-in support for multithreading

The way Gecode works is by efficiently searching the finite solution space and returning all solutions found during the search by using the DFS search (or returning the best solution found by using Branch and Bound search). Just like any other CSP specification, specifying the resilience problem in Gecode consists of variables, constraints between the variables, and the cost function that maps each variable assignment with a cost. The goal is to find the variable assignment that minimizes the cost. Unlike other CSP solvers, there are other performance-related components the users can customize for the problem, such as Propagators and Branchers. A propagator's task is to infer the assignment of variables that are in conflict with the constraint. That is, a propagator may prune some assignments from the domain that are certain to violate the constraint and thus reduce the solution space. Branchers, on the other hand, are used to describe the shape of the search tree. In the current implementation of the SR-solver, we only use the predefined Propagators and Branchers, for more detailed instruction on customizing Propagators and Branchers in Gecode, they can be found in the Modeling and Programming with Gecode [2].

We are now ready to describe how to represent the resilience problem using variables, constraints and cost functions and then define branchers for solving this problem in Gecode. Recall that the resilience problem is Given a dynamic system DS= $< S_0, G, L, d >$ and a bound k on the length of realizable SSTs, we want to determine a strategy (choice of actions for each system) such that we got the smallest threshold value for the desired Resilience property.

Creating variables

Given the DS, we can easily find out the total number of distinct actions: *num_actions* and the number of nodes: *num_nodes*. We assume that each action in the DS maps to a unique integer number ranging from 0 to *num_actions*-1. Similarly, each node gets an unique integer ranging from 0 to *num_nodes*-1. Knowing this, we can create a unique mapping of actions and nodes to the natural numbers, which allow us to deal with them as integer variables.

Then we see the strategy to be determined as an array choice of *num_nodes* integer variables where each variable in the array can take values from 0 to *num_actions*-1. Another integer array of k variable we need to define is for the SSTs, which we call sst. Each variable in sst can take values from 0 to *num_nodes*-1. In Gecode, a variable declaration is of the form:

```
SR_model(void)
   : path(*this, path_length, 0, n_nodes-1),
     choice(*this,n_nodes,0,n_actions-1) {
   [constraints on variables]
   }
```

Posting constraints

Constraints are posted to sst to restrictive Gecode solver with realized SSTs. For any nodes n_1 and n_2 in the dynamic graph, if there is no edge connecting n_1 to n_2, then we post a constraint specifying that the next system state cannot be n_2. The detailed algorithms for posting sst constraints are illustrated in Algorithm 1. Similarly, for the variable array choice, we post a "choice[i] not equal to a" constraint if action a is not in the set of L(a), that is, there are no outgoing edges with action a from i.

Algorithm 1: Constraints for variable array sst

```
Data:  Set of edges E,  num_nodes, bound k
Result: Set of constraints
for $i \leftarrow 0$ to $k-1$ do
    for $n_1 \leftarrow 0$ to num_nodes-1 do
        for $n_2 \leftarrow 0$ to num_nodes-1 do
            if $(n_1, n_2) \in E$ then
                Add constraint:
                If sst[i]=$n_1$ and $n_1$ chooses the action on edge $(n_1, n_2)$
                then sst[i+1] can be $n_2$;
            end
        end
    end
end
```

Algorithm 2: Constraints for variable array choice

```
Data:  Edge labeling function L,  num_actions , num_nodes
Result: Set of constraints
for $i \leftarrow 0$ to num_nodes-1 do
    for $a \leftarrow 0$ to num_actions-1 do
        if $a \notin L(i)$ then
            Add constraint: choice(i) not equal to $a$ ;
        end
    end
end
```

Cost function

Once the strategy and SSTs have been defined, the minimum threshold value for the requested property can be easily computed. In the current implementation of SR-solver, only one resilience property can be checked at a time. Therefore, the cost function is just the function for calculating such threshold value as defined in Section 3.1.1. For example, if we are interested in functionality, then the cost function is:

$$cost(SST) = \sum_{0}^{k-1} \frac{C_i(\alpha_i)}{k}$$

Branching

Gecode's predefined variable-value branching is used. We have two branchers - for choice variables and for the sst:

```
        branch(*this,choice,INT_VAR_NONE(), INT_VALUES_MIN());
        branch(*this, sst, INT_VAR_NONE(), INT_VALUES_MAX());
```

The branching starts by selecting the first unsigned variable in the choice array: choice[0] with the minimum value from the domain of choice. If there is no conflict with the constraints for choice, then the search continues. The branching again selects the first unassigned variable with a minimum value from the domain. Branching commands are executed in order of creation, that is to say, the sst brancher will not start assigning values to sst until the choice brancher have assigned every variable of the choice array with a value. This order is especially important because branching a realizable SST requires a complete strategy.

Searching for solutions

Gecode's parallel branch and bound engine is generic with respect to any type of Gecode model and it is written in C++. The search engine uses a work-stealing architecture for the parallel search. Initially, all the work of exploring the search tree is allocated to the main CPU, making the CPU busy. All other CPUs are initially idle and try to steal the work of exploring parts of the search tree from the main CPU. The work-stealing architecture is indeterministic, as the work that is stolen depends on machine load and other factors. The degree of parallelism is specified using the following command:

```
  Gecode::Search::Options search_op;
  search_op.threads = n_threads;
```

n_threads is a number entered by the user via GUI's "Set number of CPU cores to use" menu. This number sets the number of CPU cores to use during the branch and bound search. By default, $n\_threads = 1$. Suppose x is the number of CPU cores in the machine. If *n_threads* is greater than x, then only the actual number of CPU cores are utilized by the search engine.

We use the default parallel branch and bound engine with the cost function defined in the previous section to find the worst case of each strategy. Then, we select the optimal strategy with the lowest cost from all strategies return by the branch and bound engine.

An Example

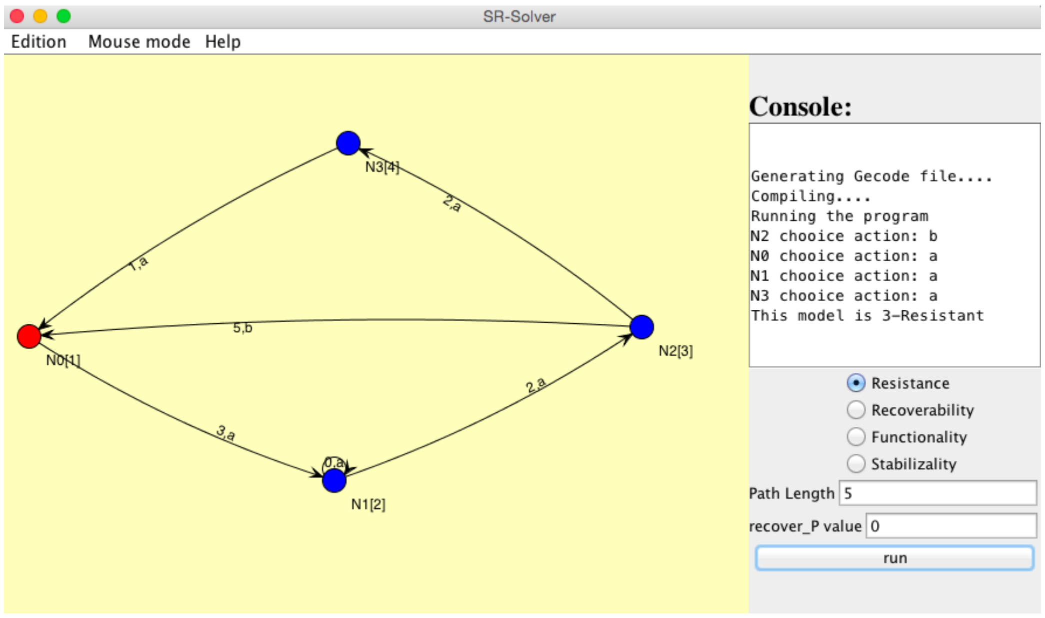
We demonstrate the SR-solver by using the dynamic system presented in Figure 3.5. From the figure, it is obvious that a strategy is just the choice of action for system state N2 since it is the only system in this dynamic system graph with more than one actions. Suppose the length of sst is fixed to 5, by running the SR-solver, we get:

- 3-Resistance by choosing action b at N2.
  If we choose action a at N2, then the worst case scenario sst is $N0 \sim N1 \sim N1 \sim N2 \sim N3$. This sst is 4-Resistance.

- Not $< 2, Q > -$Recoverable.
  This is due to the fact that N1 can stuck in a self-loop for any arbitrary number of time steps. In the worst case sst is $N0 \sim N1 \sim N1 \sim N1 \sim N2$. Since the cost of last system state is 3, this dynamic system is not $< 2, Q > -$Recoverability whatever the choice of action at N2.

- 2-Functional by choosing action b at N2. Similar to the resistant case, if we choose action a, the worst case sst is $N0 \sim N1 \sim N1 \sim N2 \sim N3$. This sst is $\dfrac{1+2+2+3+4}{5} = 2.4$-Functional

- 3-Stabilizable by choosing action a at N2.
  Note that for the Stabilizability, unlike other resilience properties, the optimum strategy is to choose action a at N2. If we choose action b, then it is 5-Stabilizable.



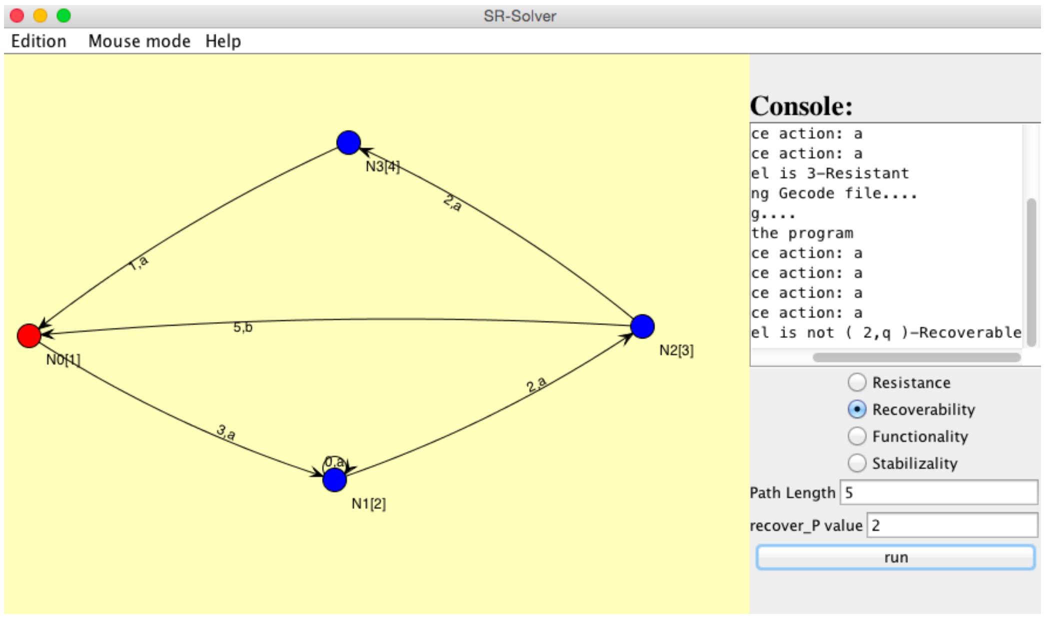
```
Generating Gecode file....
Compiling....
Running the program
N2 chooice action: b
N0 chooice action: a
N1 chooice action: a
N3 chooice action: a
This model is 3-Resistant
```

[✓ Resistance] Recoverability Functionality Stabilizality

Path Length 5

recover_P value 0

(a)



```
ce action: a
ce action: a
el is 3-Resistant
ng Gecode file....
g....
the program
ce action: a
ce action: a
ce action: a
ce action: a
el is not ( 2,q )-Recoverable
```

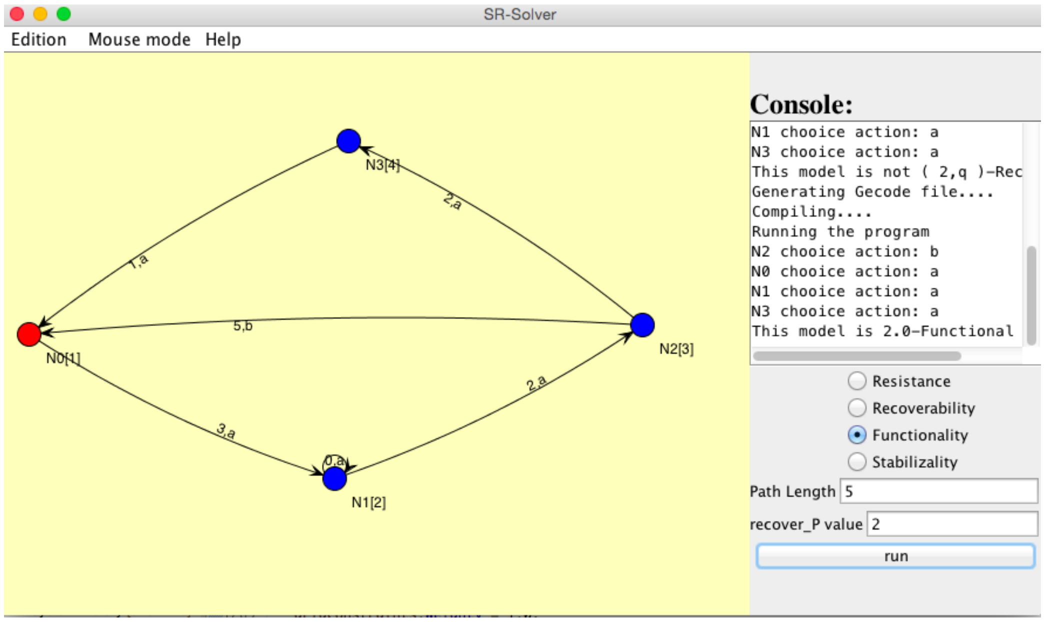
( ) Resistance
(•) Recoverability
( ) Functionality
( ) Stabilizality

Path Length 5

recover_P value 2

run

(b)



```
N1 chooice action: a
N3 chooice action: a
This model is not ( 2,q )-Rec
Generating Gecode file....
Compiling....
Running the program
N2 chooice action: b
N0 chooice action: a
N1 chooice action: a
N3 chooice action: a
This model is 2.0-Functional
```

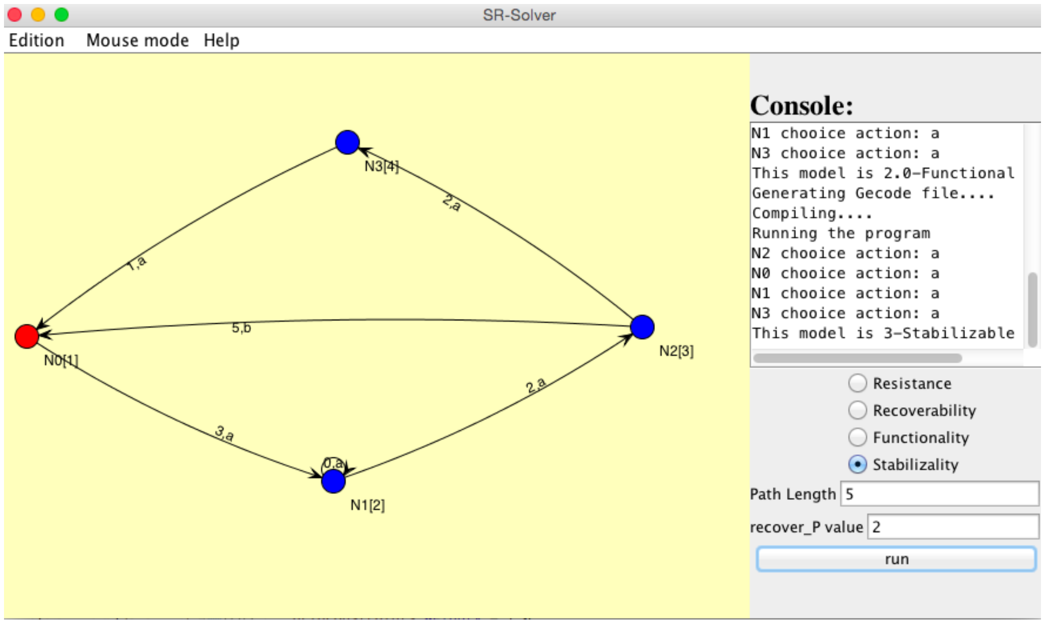
( ) Resistance
( ) Recoverability
(•) Functionality
( ) Stabilizality

Path Length 5

recover_P value 2

run

(c)



```
N1 chooice action: a
N3 chooice action: a
This model is 2.0-Functional
Generating Gecode file....
Compiling....
Running the program
N2 chooice action: a
N0 chooice action: a
N1 chooice action: a
N3 chooice action: a
This model is 3-Stabilizable
```

[✓ Stabilizality]

Path Length 5

recover_P value 2

(d)

Figure 3.6: The dynamic system presented in Figure 3.5 is: (a) 3-Resistance. (b) Not $< 2, Q > -$Recoverable. (c) 2-Functional. (d) 3-Stabilizable.

3.4 Related work

The work described in this chapter is, to our knowledge, the first complexity analysis and implementation for resilience problem based on SR-model. In this section, we review work related to the problem of system resilience.

Discrete event dynamic systems (DEDS) [26] are similar to the SR-model we considered in this chapter. A DEDS consists of discrete state spaces and event-driven state transitions where the transitions occur at a discrete instant of time. The difference between discrete event systems and SR-models lies in how dynamics are presented in the system. A DEDS, unlike the SR-model, does not allow an explicit occurrence of controlled decisions and exogenous events at the same time. We believe that our model is more realistic in the context where a system is subject to changes under the combination of controlled decisions and exogenous events at certain time step.

As to properties of dynamic systems regarding resilience, the exact set of resilience properties is largely debatable. In [4], the authors introduced a weaker notion of stabilizability: *k-maintainability*. A DEDS is said k-maintainable if certain "good" states can be reached in k steps and if there is no interference from the environment during those steps. While k-maintainability is close related to stabilizability, it is not directly relevant in the SR-model as we always deal with the exogenous event in any circumstances. In [32], the authors focus on other aspects of system resilience, namely robustness and resiliency. Robustness is defined as the capacity of self-protecting from unexpected events while resiliency is the ability of self-healing in case of system failures. They propose a Petri net based framework where both robustness and resiliency can be formally defined and measured.

3.5 Conclusions

This chapter provides the study, complexity analysis and implementation of a user-friendly software for the resilience problem. We have shown that the complexity of determining the resilience of a given SR-model can be as hard as NP-complete. Then, we provide a novel approach that used a constraint optimization solver to solve the resilience problem. We can confirm that Gecode is, indeed, a viable approach to real-life system resilience analysis. Finally, a fully graphical user interface for modeling SR-models was created, thus making the technology for solving system resilience problems accessible to general users.

We believe that we are taking important initial steps in addressing the core aspects of system resilience, namely the underlying complexity of evaluating resilience properties and the preliminary implementation of the SR-solver. As a future direction, we plan to extend the capabilities of SR-solver to capture the full definition of dynamic systems. Another direction is to add support for user preferences over resilience properties. As can be seen in the previous section, the optimal strategies for each of the resilience properties are not necessarily the same. Adding preference will address this problem and thus further increase the applicability of the SR-solver.

Bibliography

[1] Generic constraint development environment. `http://www.gecode.org/`. Accessed: 2015-10-30.

[2] Modeling and programming with gecode. `www.gecode.org/doc-latest/MPG.pdf`. Accessed: 2015-10-30.

[3] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.

[4] Chitta Baral, Thomas Eiter, Marcus Bjäreland, and Mutsumi Nakamura. Maintenance goals of agents in a dynamic environment: Formulation and policy construction. *Artif. Intell.*, 172(12-13):1429–1469, August 2008.

[5] Roman Barták. Constraint programming: In pursuit of the holy grail. In *In Proceedings of the Week of Doctoral Students (WDS99 -invited lecture*, pages 555–564, 1999.

[6] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder (extended abstract). Technical report, IN TAP 2009: SHORT PAPERS, ETH, 2009.

[7] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. A theoretical framework for the declarative debugging of datalog programs. In *Semantics in Data and Knowledge Bases, Third International Workshop, SDKB 2008, Nantes, France, March 29, 2008, Revised Selected Papers*, pages 143–159. 2008.

[8] John L. Casti. *X-events : the collapse of everything.* William Morrow, New York, 2013.

[9] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190 – 210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 04).

[10] Rina Dechter. *Constraint processing.* Morgan Kaufmann Publishers, San Francisco, 2003.

[11] Marc Denecker, Maurice Bruynooghe, and Victor Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, 2:2001, 2001.

[12] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 130–145. Springer Berlin Heidelberg, 2008.

[13] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: A tutorial. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, pages 1–8, Berlin, Heidelberg, 2012. Springer-Verlag.

[14] NICTA Optimisation Research Group. Minizinc, a modelling language and tool chain for constraint optimisation problems. `www.minizinc.org`. Accessed: 2015-10-30.

[15] Peter Hartmann, Monika Maidl, David von Oheimb, and Richard Robinson. A case study in decentralized, dynamic, policy-based, authorization and trust management–automated software distribution for airplanes. In *Security and Trust Management*, pages 68–83. Springer, 2011.

[16] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[17] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 239–249, New York, NY, USA, 1996. ACM.

[18] Peter Ladkin. Formal methods in modern critical-software development. `http://www.abnormaldistribution.org/2009/06/22/formal-methods-in-modern-critical-software-development/`. Accessed: 2015-10-30.

[19] Monika Maidl, David von Oheimb, Peter Hartmann, and Richard Robinson. Formal security analysis of electronic software distribution systems. In Michael Harrison and Mark-Alexander Sujan, editors, *Proc. of the 27th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, volume 5219 of *LNCS*, pages 415–428. Springer, 2008. `http://ddvo.net/papers/SAFECOMP08.html`.

[20] Sarah Mallet and Mireille Ducassé. Generating deductive database explanations, 1999.

[21] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.

[22] Hiroshi Maruyama. Towards systems resilience. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–4, June 2013.

[23] Ann S. Masten. Resilience in developing systems: Progress and promise as the fourth wave rises. *Development and Psychopathology*, 19:921–930, 6 2007.

[24] Artur Mikitiuk, Eric Moseley, and Miroslaw Truszczynski. Towards debugging of answer-set programs in the language pspb. In *Proceedings of the 2007 International Conference on Artificial Intelligence, ICAI 2007, Volume II, June 25-28, 2007, Las Vegas, Nevada, USA*, pages 635–640, 2007.

[25] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *In AAAI*, pages 430–435. AAAI Press/MIT Press, 2005.

[26] Peter J. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, January 1987.

[27] Peter J. Ramadge and W. Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989.

[28] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[29] Oxana Sachenkova. Using CSP solvers for Partial Configuration in Automotive Configuration Problems. Master's thesis, Chalmers University of Technology, GOTHENBURG, SWEDEN, 2011.

[30] Nicolas Schwind, Tenda Okimoto, Katsumi Inoue, Hei Chan, Tony Ribeiro, Kazuhiro Minami, and Hiroshi Maruyama. Systems resilience: A challenge problem for dynamic constraint-based agent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 785–788, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[31] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. Sugar++: A sat-based max-csp/cop solver. In *Proceedings of the 3rd International CSP Solver Competition*, pages 77–82, 2009.

[32] Madjid Tavana, Timothy E. Busch, and Eleanor L. Davis. chapter Modeling Operational Robustness and Resiliency with High-Level Petri Nets, pages 170–191. IGI Global, Hershey, PA, USA, 2013.

[33] The JUNG Framework Development Team. Jung, java universal network graph framework. `http://jung.sourceforge.net/index.html/`. Accessed: 2015-10-30.

[34] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.

[35] Christial Wieland. Two explanation facilities for the deductive database management system DeDEx, 1990.

[36] Johan Wittocx and Marc Denecker. The idp system: a model expansion system for an extension of classical logic. In *LaSh*, pages 153–165, 2008.

Appendix A

LogicQL Grammar

```
% LogicQL language grammar
% Heng Liu, May 2015


% Warning1: Using TXL default [id] definition, the complete LogicQL Identifier
    definition is a little bit different.
% Warning2: Using old LogicQL floating number definition(version 3.1 and
    earlier).
% Warning3: Currently no support for two LogicQL data-types : date and time
% Warning4: Currently no support for LogicQL aggregates
keys
 true false
end keys

compounds
   != <= >= -> <- << >> =>
end compounds

comments
  //
  /* */
end comments

define identifier
    [id]
  |[id] : [id]
end define


define LogicQL_program
    [repeat clause]
end define

define clause
      [constraint]
  | [logicQLrule]
  | [comment]
  | [fact]
end define
```

```
define fact
      [formula][end_formula]
end define

define logicQLrule
      [formula] <- [formula] [end_formula]
end define

define constraint
       [formula] -> [formula] [end_formula]
   |   -> [formula] [end_formula]
   |   '! [formula] [end_formula]
end define

define formula
       [atom]
   | [expr] [repeat comparator_expr+]
   | [formula] ', [formula]
   | [formula] '; [formula]
   | '! [formula]
   | ( [formula] )
end define

define end_formula
      .
   |  [space].
end define

define atom
     [opt deltaop] [identifier] [opt size] [opt stage] ( [arglist] )
end define

define deltaop
       +
   | -
   | ^
   | *
end define

define size
     '[ [integernumber] ']
end define

define stage
       @prev
   | @previous
   | @init
   | @initial
   | @final
end define

define arglist
       [list expr]
end define

define comparator_expr
     = [expr]
   | != [expr]
```

```
  | < [expr]
  | > [expr]
  | >= [expr]
  | <= [expr]
end define

define more_expr
      , [expr]
end define

define expr
       [identifier]
  | [literal]
  | [expr] [arithop] [expr]
  | [identifier] '[ [arglist] ']
  | ( [expr] )
end define

define literal
       [stringlit]
  | [boolean]
  | [number]
end define

define arithop
       +
  | -
  | *
  | /
end define

define boolean
       true false
end define
```

code/logicQL.Grm

Appendix B

IDP Grammar

```
% IDP language grammar
% Heng Liu, May 2015

% Warning1: Only contains fraction of IDP language that are used in the
  translation
% Warning2: Using TXL default [id] definition.
% Warning3: Currently no support for Namespaces and include statements
% Warning4: Currently no support for IDP aggregates

keys
 include using vocabulary type isa contains true false
  nat char  abs
end keys

compounds
   ~= <= >= <=>  -> <- ?< ?=< ?= ?> ..
end compounds

comments
  //
  /* */
end comments



define IDP_program
    [NL] [IDP_vocabulary_part] [NL]
    [NL] [IDP_theory_part] [NL]
    [NL] [IDP_structure_part] [NL]
end define

% *************vocabulary_part start*************
define IDP_vocabulary_part
      vocabulary [id] '{ [NL] [repeat IDP_symbol_declaration] '}
end define


define IDP_symbol_declaration
      [IDP_type_declaration] [NL]
  |[IDP_predicate_declaration] [NL]
  |[IDP_function_declaration] [NL]
end define

define IDP_type_declaration
     type [id] [opt IDP_subtype_declaration] [opt IDP_supertype_declaration]
end define

define IDP_subtype_declaration
     isa [list id]
end define

define IDP_supertype_declaration
     contains [list id]
end define

define IDP_predicate_declaration
   [id] '( [list id]  ')
end define

define IDP_function_declaration
   [id] '( [list id] ') : [id]
end define
% *************vocabulary_part end*************


% *************theory_part start*************

define IDP_theory_part
   theory [id] : [id] '{ [NL] [repeat IDP_theory_content] [NL] '}
end define

define IDP_theory_content
   [IDP_sentence] '. [NL]
  |[IDP_inductive_definitions] [NL]
end define

define IDP_sentence
   'true
  |'false
  | [id] '( [list IDP_term] ')
  | [IDP_term] '= [IDP_term]
  | '~ [IDP_sentence]
  | [IDP_sentence] & [IDP_sentence]
  | [IDP_sentence] | [IDP_sentence]
  | [IDP_sentence] => [IDP_sentence]
  | [IDP_sentence] <= [IDP_sentence]
  | [IDP_sentence] <=> [IDP_sentence]
  | ! [repeat IDP_id_with_type+] ': [IDP_sentence]
  | ? [repeat IDP_id_with_type+] ': [IDP_sentence]
end define

define IDP_id_with_type
   [id] '[ [id] ']
  | [id]
end define
% no function here. Treat all functions as predicates
define IDP_term
   [id]
```

```
     |[IDP_constant]
end define

define IDP_constant
    [stringlit]
  |[number]
end define
define IDP_inductive_definitions
    '{ [NL] [repeat IDP_inductive_rules] [NL] '}
end define

define IDP_inductive_rules
    ! [repeat IDP_id_with_type+] ': [IDP_sentence] <- [IDP_sentence] '. [NL]
    | [IDP_sentence] <- [IDP_sentence] '. [NL]
end define

% *************theory_part end*************
% *************structure_part start*************
define IDP_structure_part
    structure [id] : [id] '{ [repeat IDP_structure_containt] '}
end define

define IDP_structure_containt
     [IDP_type_enumeration]
   | [IDP_predicate_enumeration]
   | [IDP_function_enumeration]
end define

define IDP_type_enumeration
    [id] = '{ [IDP_term] [repeat IDP_term_semicolon_list] '}
  | [id] = '{ [IDP_term] .. [IDP_term] '}
end define

define IDP_term_semicolon_list
    '; [IDP_term]
end define

define IDP_predicate_enumeration
    [id] = '{ [opt '(] [list IDP_term] [opt ')] [repeat
     IDP_predicate_term_semicolon_list] '}
end define

define IDP_predicate_term_semicolon_list
    '; [opt '(] [list IDP_term] [opt ')]
end define

define IDP_function_enumeration
     [id] = '{ [list IDP_term] -> [IDP_term] [repeat
     IDP_function_semicolon_list] '}
end define

define IDP_function_semicolon_list
    ';  [list IDP_term] -> [IDP_term]
end define

% *************structure_part end*************
```

code/IDP.Grm