Reasoning About Exceptions Using Model Checking

George Fairbanks David Garlan Balaji Sarpeshkar
Reid Simmons Gil Tolle Jeannette M. Wing

August 2002
CMU-CS-02-165

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Reasoning about code that uses exceptions is difficult because exception handling mechanisms introduce many possible alternate flows of control besides the "normal" flow of control. Moreover, we often want to know something about the state of the system when an exception is raised. We describe a tool suite to help programmers reason simultaneously about exceptional flow and state. Central to our tools is an Intermediate Exception Language. We use Moped, a model checker for pushdown systems, as a backend to check IEL programs for properties expressed in Linear Temporal Logic. Our current tool suite includes a java2iel translator, which translates a subset of Java into IEL, and an iel2moped translator, which gives a pushdown systems semantics to IEL.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, in part by the National Aeronautics and Space Administration (NASA) under agreement no NCC2-1241, and in part by the National Science Foundation under Grant No. CCR-0121547. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DOD, NASA, ARO, or the U.S. Government.

Keywords: exceptions, exception handling, dynamic analysis, static analysis, model checking, linear temporal logic, Moped, SPIN, Java

1 Introduction

In safety-critical and mission-critical code, exceptions are used to signal an abnormal condition, due to a bad data value or a rare environmental event. Exception handling code executes to bring the system to a well-defined state. For example, the system may abort or continue to operate from the recovered state. Programmers use exceptions to handle boundary conditions, e.g., accessing an array out of bounds, dividing by zero, and popping from an empty stack. Designers of robust systems use exceptions to guard against all aberrant behavior they can imagine:

- If water pressure goes above a certain level, a valve must open.
- If the power fails, the backup generator must start.
- When the space vehicle faces the sun, the camera lens must be shut.

1.1 Exceptions Are Hard To Reason About

Reasoning about code that uses exceptions is inherently difficult because one is reasoning about the dynamic behavior of the system. Static analyses used to reason about dynamic behavior can be overly conservative or only approximate **need citations to back these claims**. Most do not treat exceptions comprehensively or at all.

Programming languages can help restrict the possible flows of control by placing constraints on the syntax and semantics of exception handling features. For example, it is easier to reason about the termination model found in C++ and Java than the resumption model found in Mesa and Eiffel. In the resumption model, the call stack is not "unwound"; thus, the presence of nested subroutines complicates determining the lexical context needed by the handler, and recursive resumption could lead to infinite recursion and stack overflow [BM00]. Even in the simpler termination model, variations can still complicate reasoning. For example, Java's **finally** construct and CommonLisp's similar **unwind-protect** mechanism are not found in C++. While useful, their generality allows more possible flows of control.

Programmers can exacerbate the difficulty in reasoning about exceptions. They tend to add exception handling code as an afterthought, often in response to finding errors during debugging, or worse, in response to bugs found by customers in fielded systems. These post facto additions can easily lead to convoluted code.

Programmers, however, should not be blamed for writing complex code, if it is warranted by the inherent complexity of the system or by nature of the environment in which the system is to be deployed. Indeed, they should be encouraged to use exception handling to help structure the possible return conditions of each subroutine. By imposing such structure on their code, programmers can avoid unnecessary convolution.

Ironically, programmers may shy away from an extensive use of exceptions because of the very difficulty in reasoning about the additional control paths introduced. For example, they may simply rely on a common built-in catchall *failure* exception that propagates any unhandled exception all the way to the top. It would be better to catch the exceptional condition as close as possible to the true point of failure, with the hope that the system could continue to operate.

What would help ease the difficulty of reasoning about exceptions and encourage programmers to use exceptions effectively is a an analysis tool. We have built a tool suite whose sole purpose is to help reason about exception handling. This report describes the current state of our tool suite.

1.2 Reasoning about Control Flow and State

In designing our tool suite, we imagined what kinds of questions a system designer or a programmer would want answered:

Feel free to sort these questions in some logical order and add new kinds of questions.

- Can this exception ever be thrown? Under what conditions?
  - Is the system always in a "safe" state when this exception is thrown? Here "safe" could be defined in terms of an invariant over state variables.
  - Will the system be in a "bad" state if ever this exception is thrown?
- What are the possible exceptions that can be thrown if the system is in this state (i.e., satisfies a given predicate)?
- If the system ever reaches this "bad" state, is it subsequently handled by an exception?
- If this exception is thrown, can I guarantee that I will get to a "good" system state in the next state? Eventually?
- Are all states from which an exception is potentially thrown reachable?

To answer these types of questions we need to track the program's (1) flow of control (from where is an exception raised, where is it subsequently handled, and where does control resume after handling it?) and (2) state (what are the values of the global and local variables when this exception is raised?). For example, it is not enough to know that if the "shutdown" exception is thrown, it is handled; we may also want to check that when it is thrown, all locks have been released, and that after it has been handled, all global clock variables have been reset to zero and the shared resource is free.

Whereas program analyses, such as control flow graph analysis, used by compilers are useful, they do not keep track of state. Whereas model checking facilitates reasoning about state, they are limited to finite state models and cannot handle recursive procedures. Current work on using model checkers for reasoning about software ignore exceptions, treats them naively, or limits their applicability. For example, the SLAM project [BR01] safely ignores the problem since it checks code written in C, which does not have exceptions. The Bandera project [CDHR00] ignores control flow associated with exceptions; it treats only null-pointer and type-cast exceptions in Java in a meaningful way. The ESC/Java [LNS00] project also does not explicitly track exceptional flow of control. Java PathFinder [**?**, **?**] does handle exceptions, but works only for finite state Java programs. This rich past and current work suggest some starting points of attacking our problem:

- We could start with a static analysis algorithm for generating control flow graphs with exceptions. For example, Sinha and Harrold give an algorithm for computing CFGs with exceptions for Java [SH00]. Given a CFG, we could annotate the node/edges with state information. For example, an edge, e, might be labeled with a state predicate that captures exactly when control flows along e.

- We could start with a model checker for programs that handles flow of control, but ignores or need not deal with exceptions. For example the SLAM Project's tool suite includes a model checker for boolean abstractions of C code. We could then add to such a model checker the ability to deal with exceptions. For example, we could extend SLAM to C++, thereby forcing us to model exception handling.

- We could start with standard model checkers like SMV [McM93] or SPIN [Hol97], which are used to reason about high-level designs, usually hardware, but not usually for low-level code. To handle control flow, we could a special state variable to stand for the program counter.

The problem with the first two approaches, is that we end up having to do the analysis of the original source code twice. **Need to expand on the point in the previous sentence.** The problem with the last approach is that we quickly run into the state space explosion problem when modeling the program counter explicitly.

Our approach has two novel aspects: (1) we abstract from the original source code, retaining its original flow of control, and model check our abstraction; and (2) we use a model checker that supports a pushdown automata model. These are the explicit steps in our current approach, as illustrated in Figure 1:

- We translate Java source code into an Intermediate Exception Language (IEL). The result of our translation is an abstraction of the original Java program with all control information retained. IEL, roughly speaking, is an abstract programming language.

- We translate IEL into Moped, an off-the-web model checker for pushdown systems [EHRS00].

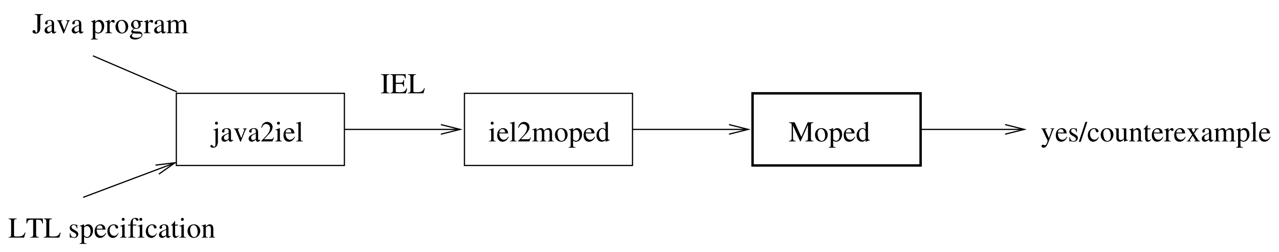

Figure 1: Exceptional Tool Suite

- We check properties of our original Java program using Linear Temporal Logic (LTL), Moped's specification language.[1]

Our use of IEL as an abstraction language has the advantage that multiple source languages can translate into IEL, hence our use of the term "intermediate." We envision for the future translators from other languages, such as C++ and CommonLisp, into IEL. Another advantage is that we can have multiple backends, i.e., different model checkers, perhaps to handle different source language semantics or to handle more complex features, such as concurrency, not easily handled by Moped.

Our use of Moped, a model checker for pushdown systems, has the advantage over other model checkers such as SPIN or SMV in that we do not have to represent the program counter explicitly, thereby avoiding a source of state space explosion. It has the advantage over the Java PathFinder model checker in that we are not restricted to finite Java programs.

We do, however, currently have other restrictions—at both the front and back ends–on what of Java we can model. Not all of a source language's features are easily represented by IEL. For example, we cannot express the creation of new objects in Java; this problem is addressed by model checkers specifically designed to analyze Java programs, e.g., the Java PathFinder [**?**] and work by Park, Stern, and Dill in model checking Java bytecode [**?**]. Not all of general exceptional flow are easily represented/checkable by Moped. For example, we cannot currently handle the full generality of Java's **finally** blocks; this problem is addressed in principle (i.e., there is no tool support) by Obdrzalek [**?**, **?**] but he models only Java control flow and not state information.

1.3 Roadmap

In the next section we introduce our Intermediate Exception Language. We give its full concrete grammar and an informal semantics. In Section 3 we describe how we translate a subset of Java into IEL. In Section 4 we describe Moped

[1]Although the figure shows the LTL specification as input to the java2iel translator, currently we feed it to Moped directly.

and our translation of IEL into Moped. Section 5 contains three examples of increasing complexity. We state ideas for future work in Section 6.

2 IEL

2.1 Design Goals

Our primary goal in designing our *Intermediate Exceptions Language* (IEL) is to have a simple language that allows us to express flow of control in the presence of exceptions. We add to a straight-line procedural language with assignments, if-then-else, loops, and procedure call the means for throwing exceptions, catching and handling them, and executing "clean up" code. We restrict IEL to have only simple types: booleans, integers, and bounded arrays. We also have assertions, as a placeholder for uninterpreted annotations or for state formulae (e.g., pre- and post-conditions, invariants).

Ideally IEL itself could serve as a source language for programmers who simply want to "sketch" out the exceptional flow of control in their designs. They would then analyze their IEL program to make sure their use of exceptions captures the intended flow of control. This analysis can be done prior to any commitment in representating data.

Initially, however, we expect that programmers will use our tool suite to analyze existing source code. As such, we also designed IEL so that multiple source languages, including C++, Java, and CommonLisp, which have exception handling features, could be translated into IEL. Thus, we want sufficient generality to express different exception handling constructs. For example, IEL's **finally** block is useful for Java and CommonLisp, but not for C++.

Since we intend IEL to be expressive enough to handle different exception handling mechanisms, we would need to provide different semantics for each model, and even perhaps for each source language from which we translate into IEL. For the most part, however, we are giving an interpretation based on a termination model of exceptions to IEL, the same as in Ada, C++, CommonLisp, ML, Modula-3, and Java. This model covers the programming languages we are currently interested in, so this interpretation does not seem overly restrictive.

Weave a couple of examples throughout next two sections.

2.2 Grammar

```
start        →  declist
declist      →  dec  declist

dec          →  exception | const | variable | procedure | assertion

exception    →  exception string [extends string]
const        →  const string num | const string true | const string false

variable     →  variable string : tydec [( num )] [:= init_exp]
tydec        →  int | bool | array of tydec [ num ]
init_exp     →  num | true | false | { init_exps }
init_exps    →  init_exp [ , init_exps]

procedure    →  procedure string ( [param_list] ) [assertions] block
param_list   →  param [ , param_list]
param        →  string : type
type         →  int | bool | array of type

assertions   →  assertion [assertions]
assertion    →  [ string ]

block        →  { [block_stmts] }
block_stmts  →  block_stmt [block_stmts]
block_stmt   →  stmt | variable
```

```
stmt          →  return | break | throw string | try_block | call | conditional |
              →  loop | block | assign | assert | check

try_block     →  try stmt [catch_list] [finally_block]
catch_list    →  catch_block [catch_list]
catch_block   →  catch string stmt
finally_block →  finally stmt

call          →  string ( [arg_list] )
arg_list      →  exp [ , arg_list]

conditional   →  if exp then stmt [else stmt]
loop          →  while exp stmt

assign        →  var := exp | var := choice
choice        →  choice [ [arg_list] ]

assert        →  assert string exp
check         →  check string exp

exp           →  num | true | false | var | !exp | op_exp | cmp_exp | ( exp )
var           →  string | var [ exp ]
op_exp        →  exp + exp | exp - exp | exp * exp | exp / exp | exp && exp | exp || exp
cmp_exp       →  exp < exp | exp > exp | exp <= exp | exp <= exp | exp = exp | exp != exp

num           →  INTEGER_LITERAL
string        →  STRING_LITERAL
```

2.3 Informal Semantics

IEL is an intermediate language, and as such, is not interpreted or executed. Therefore, there is no formal semantics for the language. However, since IEL may be used directly as a design language, it is helpful to have an informal guide to the language. This includes assumptions and restrictions not inherent in the syntax of the language. The following is a concise list of such features of IEL.

- In a variable declaration, an optional integer argument may be given after the type declaration that specifies the number of bits Moped should use for that variable.
- Multidimensional arrays are currently not supported by IEL or Moped.
- IEL assertions (as specified by the *assertions* production) are currently ignored.
- The test expression in an `if-then` or `while` statement must be a comparison expression.

- The `choice` operator represents a nondeterministic choice from some set of values. If any arguments are present, they must be integer literals.

3 Java to IEL

Weave a couple of examples throughout.

3.1 Restrictions

1. Restrictions on classes:
   - There are no calls or references to external classes (or libraries).
   - All classes are in the same package (or there are no packages defined).
   - Exception classes do not define any fields.
   - Inheritance is not supported (except for exception types).
2. Restrictions on instance fields:
   - Fields of object type must be initialized in a constructor.
   - Object fields may be initialized at most once.
3. Restrictions on methods:
   - The return type of all methods must be either `void`, `int`, or `boolean`.
   - There may only be one method named "main".
4. Other restrictions:
   - Multidimensional arrays are not supported.
   - Any arrays are initialized at most once, with a constant size.
   - A call to an object constructor may only take literal arguments.
   - All `new` expressions must be statically countable.
   - All `throw` expressions must create a new exception object.

3.2 Translation Rules

The rules to translate a Java method body will be presented as a set of inference rules that recursively produce an IEL abstract syntax tree. Also, the type signatures of the translation functions are given below. Types of structures representing Java syntactic elements are in roman, and IEL object types are given in italics. Also, where a list of elements is returned, the possible types of the elements are given in parentheses. Finally, in translation rules that check the returned type of the *exp* rule, the object checked is the *last* element of the returned List.

stmt	:	Statement $\rightarrow$ *List(Stmt, VarDec)*
bstmt	:	BlockStatement $\rightarrow$ *List(Stmt, VarDec)*
bvar	:	VariableDeclarator $\rightarrow$ String * *TyDec* * *InitExp*
bty	:	Type * *TyDec* $\rightarrow$ *TyDec*
bexp	:	Literal $\rightarrow$ *InitExp*
finit	:	ForInit $\rightarrow$ *List(AssignStmt, VarDec)*
fupd	:	ForUpdate $\rightarrow$ *List(AssignStmt)*
swstmt	:	SwitchCaseBlock $\rightarrow$ *Exp* * *BlockStmt*
swexp	:	Literal $\rightarrow$ *Exp*
cstmt	:	CatchBlock $\rightarrow$ *CatchStmt*
exc	:	AllocationExpression $\rightarrow$ String
exp	:	Expression $\rightarrow$ *List(Exp, AssignStmt, CallStmt)*

$$\frac{\forall i \quad b_i \rightarrow_{bstmt} b'_i}{\{b_1 \ldots b_n\} \rightarrow_{stmt} \text{BlockStmt}(\{b'_1, \ldots, b'_n\})}\ \textit{Block}$$

A method body in Java consists of a list of block statements, each of which can be either a standard statement or a local variable declaration.

$$\frac{s \rightarrow_{stmt} s'}{s \rightarrow_{bstmt} s'}\ \textit{BlockStmt}$$

$$\frac{\forall i \quad v_i \rightarrow_{bvar} (v'_i, \tau_i, i_i),\ (\tau, \tau_i) \rightarrow_{bty} \tau'_i}{\tau\ \{v_1, \ldots, v_n\} \rightarrow_{bstmt} \{\text{VarDec}(v'_i, \tau'_i, i_i), \ldots, \text{VarDec}(v'_n, \tau'_n, i_n)\}}\ \textit{BlockVarDec}$$

A local variable declaration in a Java block has the following syntax:
type (*name [*`[]`*][= init_exp]*)*
Since an IEL type declaration incorporates array information, we must reconcile the (optional) array information from the second chunk of the declaration (the "VariableDeclarator", in Java parlance) with the stated type.

In addition, this rule ignores all declarations of objects. In those cases where an object is declared as an instance field, it must be initialized with one of its constructors.

$$\frac{}{(\texttt{int}, \text{ArrayDec}) \rightarrow_{bty} \text{ArrayDec}(\text{IntDec})}\ \textit{BlockIntArrayDec}$$

$$\frac{}{(\texttt{bool}, \text{ArrayDec}) \rightarrow_{bty} \text{ArrayDec}(\text{BoolDec})}\ \textit{BlockBoolArrayDec}$$

$$\frac{}{(\texttt{int}, \_) \rightarrow_{bty} \text{IntDec}}\ \textit{BlockIntDec} \qquad \frac{}{(\texttt{bool}, \_) \rightarrow_{bty} \text{BoolDec}}\ \textit{BlockBoolDec}$$

The above rule resolves the type information in the declaration. For our purposes, we require that the type of the declaration be a primitive type (either `int` or `boolean`) and that array information be given after the name of the variable.

$$\frac{}{\text{v} \rightarrow_{bvar} (\text{v}, \text{null}, \text{null})}\ \textit{BlockVarName} \qquad \frac{e \rightarrow_{bexp} e'}{\text{v} = e \rightarrow_{bvar} (\text{v}, \text{null}, e')}\ \textit{BlockVarInit}$$

$$\frac{}{\text{v}[] = \texttt{new}\ \_[\text{n}] \rightarrow_{bvar} (\text{v}, \text{ArrayDec}(\_, \text{n}), \text{null})}\ \textit{BlockArrayVar}$$

$$\frac{\forall i \quad e_i \rightarrow_{bexp} e'_i}{\text{v}[] = \{e_1, \ldots, e_n\} \rightarrow_{bvar} (\text{v}, \text{ArrayDec}(\_, \text{n}), \text{ArrayInit}(\{\text{e}'_1, \ldots, \text{e}'_\text{n}\}))}\ \textit{BlockArrayInit}$$

Currently, we conservatively require that all arrays be initialized upon creation, either with a constant size or a list of literal expressions. Also, when initializing an array, we ignore the type information in the ArrayDec object for now. This information is filled in by the *bty* rule above.

$$\frac{}{\text{n} \rightarrow_{bexp} \text{IntInit}(\text{n})}\ \textit{BlockIntInit}$$

$$\frac{}{\texttt{true} \rightarrow_{bexp} \text{BoolInit}(\text{true})}\ \textit{BlockBoolInit} \qquad \frac{}{\texttt{false} \rightarrow_{bexp} \text{BoolInit}(\text{false})}\ \textit{BlockBoolInit}$$

These rules are used by the *bvar* rules above to handle variable initializers.

Next, we look at the rules to translate Java statements.

$$\frac{}{\texttt{return;} \rightarrow_{stmt} \text{ReturnStmt}}\ \textit{Return}$$

$$\frac{e \rightarrow_{exp} e' : \text{Exp}}{\texttt{return}\ e; \rightarrow_{stmt} \{\text{AssignStmt}(\textit{curMethod\_result}, e'), \text{ReturnStmt}\}}\ \textit{ReturnExp}$$

These rules handle `return` expressions. In the second case, a value is returned from the Java method. IEL does not support value-returning functions, so we have a special "return" variable for each defined procedure. The rule then sets this special variable to the value of e', which is then accessed by the caller.

Another point to note is that the *exp* rule, which will be specified later, does not necessarily return an object of type Exp. The Exp class represents an IEL expression. However, since method calls and assignments are expressions in Java but statements in IEL, the *exp* rule must return a generic Absyn object. Therefore, an assignment expression, for instance, will return a Stmt object when translated, rather than an Exp. Here, we apply the constraint that the translated expression must be an IEL Exp object.

$$\frac{}{\texttt{break;} \rightarrow_{stmt} \text{BreakStmt}}\ \textit{Break}$$

The `break` statement is included in IEL only for completeness. Currently, the IEL to Moped translator does not support `break` statements, but they may be still be used in `switch` blocks to end cases. Also, `continue` statements are currently not handled.

$$\frac{}{;\ \rightarrow_{stmt} \text{BlockStmt(null)}}\ \textit{Empty}$$

This rule handles the trivial statement. As such, it is not intended to handle semicolons at the end of statements, but rather, semicolons used as standalone statements.

$$\frac{e \rightarrow_{exp} e' : \text{Exp} \quad s \rightarrow_{stmt} s'}{\texttt{if}\ (e)\ s \rightarrow_{stmt} \text{IfStmt}(e', s', \text{null})}\ \textit{IfThen}$$

$$\frac{e \rightarrow_{exp} e' : \text{Exp} \quad s_1 \rightarrow_{stmt} s_1' \quad s_2 \rightarrow_{stmt} s_2'}{\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \rightarrow_{stmt} \text{IfStmt}(e', s_1', s_2')}\ \textit{IfThenElse}$$

$$\frac{e \rightarrow_{exp} e' : \text{Exp} \quad s \rightarrow_{stmt} s'}{\texttt{while}\ (e)\ s \rightarrow_{stmt} \text{WhileStmt}(e', s')}\ \textit{While}$$

$$\frac{e \rightarrow_{exp} e' : \text{Exp} \quad s \rightarrow_{stmt} s'}{\texttt{do}\ s\ \texttt{while}\ (e); \rightarrow_{stmt} \{s', \text{WhileStmt}(e', s')\}}\ \textit{DoWhile}$$

$$\frac{i \rightarrow_{finit} i' \quad e \rightarrow_{exp} e' : \text{Exp} \quad u \rightarrow_{fupd} u' \quad s \rightarrow_{stmt} s'}{\texttt{for}(i;\ e;\ u)\ s \rightarrow_{stmt} \{i', \text{WhileStmt}(e', \text{BlockStmt}(\{s', u'\}))\}}\ \textit{For}$$

In a Java `for` loop, the init, test, and update expressions are all optional. For our purposes, we require that they all be present.

$$\frac{\forall i\ e_i \rightarrow_{sexp} e_i'}{\{e_1, \ldots, e_n\} \rightarrow_{finit} \{e_1', \ldots, e_n'\}}\ \textit{ForInitExp}$$

$$\frac{\tau\ \{v_1, \ldots, v_n\} \rightarrow_{bstmt} \{v_1', \ldots, v_n'\}}{\tau\ \{v_1, \ldots, v_n\} \rightarrow_{finit} \{v_1', \ldots, v_n'\}}\ \textit{ForInitDecs}$$

A valid init expression here can be either an assignment to an existing variable (as specified in the *sexp* rule below), or a local variable declaration. In the case of a local declaration, we require that the variable be initialized to a constant value. In addition, a comma-separated list of expressions may be substituted for a single expression.

$$\frac{e \rightarrow_{exp} e' : \text{AssignStmt}}{e \rightarrow_{sexp} e'}\ \textit{StatementExp}$$

$$\frac{\forall i\ e_i \rightarrow_{sexp} e'_i}{\{e_1, \ldots, e_n\} \rightarrow_{fupd} \{e'_1, \ldots, e'_n\}}\ \mathit{ForUpdate}$$

A valid update expression may only be a list of assignment statements to existing local variables.

$$\frac{e \rightarrow_{exp} e' : \mathrm{Exp} \quad \forall i\ g_i \rightarrow_{swstmt} (n_i, \{b'_{i1} \ldots b'_{im}\})}{\texttt{switch}\ (e)\ \{g_1 \ldots g_n\} \rightarrow_{stmt} \mathrm{IfStmt}(e' = n_1, \{b'_{11} \ldots b'_{1m}\}, \ldots, \mathrm{IfStmt}(e' = n_n, \{b'_{n1} \ldots b'_{nm}\}, \mathrm{null}))}\ \mathit{SwitchNoL}$$

$$\frac{e \rightarrow_{exp} e' : \mathrm{Exp} \quad \forall i < n\ g_i \rightarrow_{swstmt} (n_i, \{b'_{i1} \ldots b'_{im}\}) \quad g_n \rightarrow_{swstmt} (\mathrm{null}, \{b'_{n1} \ldots b'_{nm}\})}{\texttt{switch}\ (e)\ \{g_1 \ldots g_n\} \rightarrow_{stmt} \mathrm{IfStmt}(e' = n_1, \{b'_{11} \ldots b'_{1m}\}, \ldots, \mathrm{IfStmt}(e' = n_{n-1}, \{b'_{(n-1)1} \ldots b'_{(n-1)m}\}, \{b'_{n1} \ldots b}$$

We translate a `switch` statement into a set of nested if-then-else clauses. If there is no `default` clause present, then the final IEL else clause is left as null.

$$\frac{e \rightarrow_{swexp} e' \quad \forall i\ b_i \rightarrow_{bstmt} b'_i}{\texttt{case}\ e :\ b_1 \ldots\ b_n \rightarrow_{swstmt} (e', \{b'_1, \ldots, b'_n\})}\ \mathit{SwitchCase}$$

$$\frac{\forall i\ b_i \rightarrow_{bstmt} b'_i}{\texttt{default} : b_1 \ldots b_n \rightarrow_{swstmt} (\mathrm{null}, \{b'_1, \ldots, b'_n\})}\ \mathit{SwitchDefaultCase}$$

Each switch case may have a list of block statements.

$$\frac{}{\mathrm{n} \rightarrow_{swexp} \mathrm{IntExp(n)}}\ \mathit{SwitchIntExp}$$

$$\frac{}{\texttt{true} \rightarrow_{swexp} \mathrm{BoolExp(true)}}\ \mathit{SwitchBoolExp} \qquad \frac{}{\texttt{false} \rightarrow_{swexp} \mathrm{BoolExp(false)}}\ \mathit{SwitchBoolExp}$$

A general `case` must specify as the test value an integer or boolean literal, and not a general expression. This rule exists to enforce this constraint.

$$\frac{b_1 \rightarrow_{stmt} b'_1 \quad \forall i\ c_i \rightarrow_{cstmt} c'_i \quad b_2 \rightarrow_{stmt} b'_2}{\texttt{try}\ b_1\{c_1 \ldots c_n\}\ \texttt{finally}\ b_2 \rightarrow_{stmt} \mathrm{TryStmt}(b'_1, \{c'_1, \ldots, c'_n\}, b'_2)}\ \mathit{TryBlock}$$

A Java `try` block can have the following forms: `try-catch`, `try-finally`, and `try-catch-finally`. Although the above rule seems to indicate otherwise, IEL supports all of the above forms. The statements b_1 and b_2 above are required to be Java blocks.

$$\frac{b \rightarrow_{stmt} b'}{\texttt{catch}(\tau\ \_)\ b \rightarrow_{cstmt} \mathrm{CatchStmt}(\tau, b')}\ \mathit{Catch}$$

As with `try` and `finally` clauses above, a `catch` statement must consist of a Java block. The exception type τ must be a valid (defined) exception type.

$$\frac{e \rightarrow_{exc} e'}{\texttt{throw}\ e \rightarrow_{stmt} \text{ThrowStmt}(e')}\ \textit{Throw}$$

$$\frac{}{\texttt{new}\ e(\_) \rightarrow_{exc} e}\ \textit{ExceptionName}$$

Currently, we do not support dynamic creation of exception objects. So, a `throw` expression must explicitly create an exception object immediately before throwing it.

Now, we look at the rules for translating expressions.

$$\frac{}{\text{n} \rightarrow_{exp} \text{IntExp(n)}}\ \textit{IntConst}$$

$$\frac{}{\texttt{true} \rightarrow_{exp} \text{BoolExp(true)}}\ \textit{BoolConst} \qquad \frac{}{\texttt{false} \rightarrow_{exp} \text{BoolExp(false)}}\ \textit{BoolConst}$$

$$\frac{}{\text{v} \rightarrow_{exp} \text{SimpleVar(v)}}\ \textit{SimpleVar} \qquad \frac{e \rightarrow_{exp} e' : \text{Exp}}{\text{v}[e] \rightarrow_{exp} \text{ArrayVar}(\text{v}, e')}\ \textit{ArrayVar}$$

These rules handle variable access expressions. Of particular note is the return type of these rules. In the IEL abstract syntax, both the SimpleVar and ArrayVar classes are subclasses of the VarExp class. So, under these rules, any variable access expression, when translated, will return an object of type VarExp, which can be either type of expression.

$$\frac{e \rightarrow_{exp} e' : \text{VarExp}}{{+}{+}e \rightarrow_{exp} \text{AssignStmt}(e', \text{OpExp}(e', \text{IntExp}(1), +))}\ \textit{PreIncrement}$$

$$\frac{e \rightarrow_{exp} e' : \text{VarExp}}{{-}{-}e \rightarrow_{exp} \text{AssignStmt}(e', \text{OpExp}(e', \text{IntExp}(1), -))}\ \textit{PreDecrement}$$

The increment and decrement expressions in Java provide an example of a situation in which the *exp* translation rule produces an IEL object that is not of type Exp. These two, in particular, produce an AssignStmt object that increments or decrements the specified variable.

Notably omitted here are rules to translate postfix increment and decrement expressions, for instance, `x++`. While these appear to be identical to the prefix versions, they are not permitted because the Java semantics handle them differently. Specifically, the value of a postfix expression is the value of the variable *before* it is modified. To avoid unnecessarily complicating the translation and the IEL semantics, we have decided to avoid these types of expressions.

$$\frac{e \rightarrow_{exp} e' : \text{Exp}}{!e \rightarrow_{exp} \text{OpExp}(\text{null}, e', !)}\ Not$$

$$\frac{e \rightarrow_{exp} e' : \text{Exp}}{-e \rightarrow_{exp} \text{OpExp}(\text{IntExp}(0), e', -)}\ UnaryMinus$$

$$\frac{e_1 \rightarrow_{exp} e_1' : \text{Exp} \quad e_2 \rightarrow_{exp} e_2' : \text{Exp}}{e_1\ op\ e_2 \rightarrow_{exp} \text{OpExp}(e_1', e_2', op)}\ OpExp$$

$$\frac{e_1 \rightarrow_{exp} e_1' : \text{VarExp} \quad e_2 \rightarrow_{exp} e_2' : \text{Exp}}{e_1 = e_2 \rightarrow_{exp} \text{AssignStmt}(e_1', e_2')}\ AssignExp$$

$$\frac{e_1 \rightarrow_{exp} e_1' : \text{VarExp} \quad e_2 \rightarrow_{exp} e_2' : \text{CallStmt}}{e_1 = e_2 \rightarrow_{exp} \{e_2', \text{AssignStmt}(e_1', \text{SimpleVar}(calledMethod\_result))\}}\ CallAssign$$

Since procedure calls in IEL do not return values, we have to deal with the special case where the result of a Java method call is assigned to a variable. In this case, we declare a special IEL variable for each procedure that should return a value. This variable is assigned the appropriate return value prior to exiting the procedure in question. Then, the local variable can be assigned the value of this special "return" variable.

$$\frac{e_1 \rightarrow_{exp} e_1' : \text{VarExp} \quad e_2 \rightarrow_{exp} e_2' : \text{Exp}}{e_1\ op = e_2 \rightarrow_{exp} \text{AssignStmt}(e_1', \text{OpExp}(e_1', e_2', op))}\ AssignOp$$

This rule handles arithmetic assignment expressions in Java, e.g. `x += 5`.

3.3 Translating a Java Program

The above rules will translate a Java method, but they will not suffice to translate an entire program consisting of multiple classes. The following algorithm will show how to take a whole Java program and produce an IEL abstract syntax tree.

Given:	J: a set of ASTs of the Java classes
Require:	classes represented by C can be compiled by `javac`
Output:	E: a set of IEL exception declarations
	C: a set of IEL constant declarations
	V: a set of IEL variable declarations
	P: a set of IEL procedure declarations
Declare:	P_c: a mapping from Java constructor names and types to IEL procedure names
	P_t: a set of procedure templates
	V_t: a set of variable templates

```
for all $c \in J$ do
  if $c$ extends some class $c'$ then
    add $(c, c')$ to $E$ $\{c$ must be an exception$\}$
  else
    for all declarations $d \in c$ do
      if $d$ is a constructor then
        determine $t$, a list of the input types to $d$
        create $d'$, a fresh constructor for class $c$
        add new mapping $(d, t) \rightarrow d'$ to $P_c$
      else if $d$ is a method then
        translate the formal parameters into $p$
        get $m$, the name of the method
        extract $b$, the body of the method add new procedure template
        $(c, m, p, b)$ to $P_t$
      else $\{d$ is a field declaration$\}$
        get $f$, the name of the field
        determine $t$, the IEL type of the field
        determine $i$, an optional initializer for the field
        add new variable decl template $(c, f, t, i)$ to $V_t$
      end if
    end for
  end if
end for
for all $m \in P_t$ do
  translate and store the body of $m$
end for
for all $v \in V_t$ do
  if $v$ is a constant then
    add a new $v'$ to $C$
  else if $v$ is static then
    add a new $v'$ to $V$
  else
    add a new $v'$ to $V$ for each instantiation of $v$'s class
  end if
end for
for all $m \in P_t$ do
  if $m$ is static then
    add a new $m'$ to $P$
  else
    add a new $m'$ to $P$ for each instantiation of $m$'s class
  end if
end for
```

4 IEL to Moped

4.1 Model Checking Pushdown Systems

Moped [Sch] is a tool for model checking for pushdown systems. What follows is an synopsis of a formal description of the theory underlying Moped, taken from [EKS00].

A pushdown system has a finite set of control locations, a finite stack alphabet, a finite set of transition rules, an initial control location, and a bottom stack symbol. Each transition rule takes a configuration of the system to another, where a configuration is a pair of a control location and a stack.

Moped's specification language is Linear Temporal Logic. The validity of an LTL formula is defined for a run of a state transition system, where a run is an infinite sequence of states.

Moped can handle these variants of the model checking problem, for a given property ϕ, written in LTL:

- Is ϕ valid for the initial configuration?
- Compute the set of all configurations, reachable or not, that violate ϕ.
- Compute the set of all reachable configurations that violate ϕ.

The stack component of pushdown systems allow us to model the dynamic call chain of a program, i.e., through a stack of activation records. No special state variable is needed to model the program counter or to track a procedure's return location; where to return from a procedure call is implicit in the stack discipline. Pushdown systems also can handle recursion. For these reasons we chose Moped as our backend model checker. Pushdown systems cannot model concurrent programs; thus we are restricted to date to model checking only sequential programs.

4.2 Restrictions due to Moped

- Moped cannot support an efficient heap, due to the state-space explosion of a single large array. We do not translate programs that allocate dynamic data, and do not include support in IEL for these programs.
- Moped does not support pointer-style indirect reference. This can be simulated by indexes into arrays, but again we start to deal with large arrays and the concomitant state-space explosion.

4.3 Restrictions on Finally Blocks

Correctly maintaining the Java semantics for finally blocks has proven to be one of the more difficult parts of this translation.

Code within finally blocks must execute in a context that does not contain a propagating exception. If a try block exists within a finally block, the test for thrown exceptions after the call returns should not see any exceptions propagating when the finally block started. If that thrown exception is not caught within the finally block, it should supersede any previously propagating exceptions and cause the finally block to end immediately. If the exception is caught within the finally block, and the finally block ends normally, then the original exception must continue to propagate.

We considered several approaches to this problem. First, we tried omitting the exception propagation check after a procedure call if the call was inside a finally block. We hoped to not let exceptions propagate out of finally blocks as a first approximation. This provided an incorrect semantics in the case where an exception is thrown within that procedure call. Without the check, the try block will complete normally and the exception will then propagate from the check after the next procedure call outside of the try block.

Next, we tried including a flag in the Moped program that was set upon entry into a finally block. This flag stopped the check after function calls when it was set, producing a similar effect to before, but not limited purely to finally blocks. We set the flag upon entry to a finally block, and cleared it upon exit. We also cleared the flag when an exception was thrown. This approach proved incorrect as well. If a finally block executes because of an exception, and another exception is both thrown and caught within the finally block, then the original exception is cancelled out and ceases to propagate.

To correctly translate finally blocks, we need to save the contents of all exception variables when a finally block is entered, clear the variables, and replace the contents of the variables when the finally block exits normally. This could be accomplished for finally blocks of arbitrary nesting depth by executing a finally block as a subroutine and saving the exceptions on the stack. But, Moped's local variables cannot be shared between subroutines, leaving the code within the finally block unable to access the variables in the enclosing procedure. We could push the currently-propagating exception information onto a separate data stack when entering a finally block, and pop it when leaving normally. We decided that allocating space for a stack with $numberOfExistingExceptions$ bits times $depthOfDeepestFinallyBlock$ might overwhelm Moped, and the additional stack-manipulation code might slow down the execution.

We choose to deal with this problem by including support for one level of finally blocks, and not supporting nested finally blocks. For each exception, we allocate both a variable and a save variable. When a finally block is entered, we copy the exceptions to the save variables, and clear them. When the finally block exits, we restore the contents of the exception variables. If an exception is propagating when the finally block is entered, and no new exception is thrown from within the finally block, then the original exception will propagate after being saved and restored. If a new exception is thrown and caught within the finally block, then the original exception will not be touched and will continue after the finally block ends. If a new exception is thrown and not caught within the finally block, control will leave the finally block without restoring the saved exceptions. The new exception will propagate. We believe this to be a reasonable approximation, and it is extendable to support greater finite nesting depths if we later find real-world programs that require more than one level of finally blocks.

4.4 IEL to Moped Translation Rules

The translation from IEL to Moped is presented below as a set of inference rules specifying a recursive translation. These rules are presented in the following format:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{IEL Exp}(\text{subExp}_1, \ldots, \text{subExp}_n), \text{parameter}_1, \ldots, \text{parameter}_m} \\
\hline
\quad & \text{subExp}_1 \mapsto \mathit{subExpTranslated}_1 \\
& \ldots \\
& \text{subExp}_n \mapsto \mathit{subExpTranslated}_n \\
\cline{2-2}
& \texttt{moped translation} \\
& \mathit{subExpTranslated}_k \\
& \texttt{moped translation} \\
\cline{2-2}
& \text{nextstate} = \text{(explained below)}
\end{array}$$

The first section of the rule indicates the specific type of node in the IEL abstract syntax tree being translated by the rule. The elements in the following parentheses are the subtrees below the node. The parameters following the rule can be thought of as arguments to the translation function, and are used to hold state throught the recursive translation.

The next section of the rule describes the recursive translations that have to occur before this translation can occur. These are described as mappings from the elements of the subtree to translated versions of those elements, to be used in the third section of this rule. This translation uses several environments to manage variable declarations and procedure declarations. Actions on these environments, like add and lookup, may also be described in this line of the rule. Finally, the lines in this rule should be read in order, as the result of one recursive action may be used as input to a subsequent recursive action within the same rule.

The third and fourth parts of the rule describe the output of the translation. The third part contains Moped code in typewriter face, the contents of the parameters in roman face, and the output of recursive translations in italic face. This code is the primary "return value" of each rule.

The fourth part of the rule contains the "nextstate" variable when translating IEL statements, and nothing otherwise. States in Moped code must be uniquely named. We accomplish this by identifying each state with the name of the procedure containing the IEL statement from which the Moped state transition was derived. We then append a unique number to the state name. This number increases as the translation proceeds through the sequential moped states. "nextstate" is defined to be the next available number for subsequent statements. This value is also "returned" from each statement rule.

When a global variable declaration, exception declaration, or procedure declaration is being translated, the purpose of the rule is to add the output to an environment for later use in the "Start" rule described below. Nextstate is not needed, and therefore is not displayed.

Start(decList)

Exceptions= $\emptyset$, Globals= $\emptyset$, Procedures= $\emptyset$
decList $\mapsto_{dec}$ Exceptions, Globals, Procedures
Globals $\cup$(`Exception`, `(1)`, `0`)
Globals $\cup$(`Exception_save`, `(1)`, `0`)
Globals $\mapsto_{varList}$ *globalDecs*
Procedures$_1$ $\mapsto_{stateList}$ *stateList*$_1$
Procedures$_1$.Locals $\mapsto_{varList}$ *localDecs*$_1$
...
Procedures$_k$ $\mapsto_{stateList}$ *stateList*$_k$
Procedures$_k$.Locals $\mapsto_{varList}$ *localDecs*$_k$
Procedures $\mapsto_{procList}$ *procedureDecs*
Globals $\mapsto_{varAssignList}$ *globalInitAssign*
Globals $\mapsto_{frame}$ *frameGlobals*

```
global int globalDecs;
local (stateList_1) int localDecs_1;
...
local (stateList_k) int localDecs_k;
(q <init1>)
procedureDecs
q <init1> --> q <main1> "initialize globals" (globalInitAssign)
q <normalend> --> q <normalend> "normal termination" (frameGlobals)
q <exnend> --> q <exnend> "exceptional termination" (frameGlobals)
```

This rule is the beginning of the translation. It calls all other rules recursively. An IEL program is expressed as a list of declarations. Each declaration is translated individually. Three environments, named Exceptions, Globals, and Procedures are filled with the output of these translations. The format of the entries in these environments will be described with the rules for variable, exception, and procedure declarations below. Once these environments have been filled, the contents are translated into the Moped code above. The details of these translations will be explained below.

The overall structure of a Moped program begins with a list of the global variable declarations. The addition of the `Exception` variable to the global environment establishes the root of the exception subtyping hierarchy as a real variable. `Exception_save` will be explained in the section on finally blocks.

Next, we associate the list of states corresponding to each procedure with a list of the local variables declared in that procedure. The following line defines the initial configuration of the pushdown system. We set this to be the state called "init1". Next, we append the code of each procedure. Our translation creates the state "init1" to be the initial state. This state initializes the global variables to the correct values, and then transitions to the state "main1". This is the first state of the required main procedure. Finally, we create two states named "normalend" and "exnend". These states are designed to be included in LTL formulas, and they represent a normal ending of the program and an exceptional ending respectively. These termination states only self-loop, as this is the accepted way of terminating execution in Moped.

The *stateList* translation produces a comma-separated list the names of each Moped state associated with a procedure. The *varList* translation produces a comma-separated list of Moped variable declarations for each variable in the environment. The form of a Moped variable declaration is described below the **VarDec** rule. The *procList* translation concatenates the Moped code associated with each procedure, separated by blank lines.

The *varAssignList* translation and the *frame* translation will be explained below the **VarDec** rule, as an explanation of the variable environments is necessary first.

4.4.1 *dec* Translation Rules

$$\frac{\textbf{VarDec}(\text{name, ty, init})}{\dfrac{\begin{array}{l}\text{ty} \mapsto_{ty} tyDec \\ \text{init} \mapsto_{exp} initExp\end{array}}{\text{Globals} \cup (\text{name}, tyDec, initExp)}}$$

Global variable declarations are only entered into the Globals environment. Each entry in the environment contains the name of the variable, a string representing the type as produced by the *ty* translation, and may contain a string representing an initializer expression. The *exp* and *ty* translations are detailed at the end of this section.

In Moped, changes to the state of a variable use the syntax $var\texttt{' = }exp$, indicating that as the state transition is taken, the value of *var* changes to the result of evaluating the expression *exp*. The *varAssignList* translation seen in the **Start** rule is defined over variable environments like these. It produces a list of these assignment expressions, separated by the `&` sign. The translation *varAssign* is used later, in the rule for `AssignStmt`, and produces a Moped expression with a single assignment.

The *frame* translation mentioned in the **Start** rule is also defined over these variable environments. The output of the translation is a Moped expression with the form $var_{1_{name}}\texttt{' = }var_{1_{name}}\ \texttt{\& ...\&}\ var_{k_{name}}\texttt{' = }var_{k_{name}}$. When included in a state transition, this expression ensures that each variable in the environment stays the same as the transition is taken. We call this the *frame axiom*. Without this axiom, the transition system would not accurately reflect the original IEL program, as all variables would be free to change on their own at any time.

We will use *frameGlobals* as a shorthand notation without including the translation "Globals $\mapsto_{frame}$ *frameGlobals*" in the recursive section.

As a side note, the frame axiom as applied to an array variable has a somewhat different form in Moped. Moped expressions can use the for-all operator `A` in this manner: `(A _i (0,` n`-1)` arr`'[_i] =` arr`[_i])`. This expression maintains the values of each element in the array. When element k of the array must change while the rest stay the same, we use the following expression: `(`arr`'[`k`] =` exp`) & (A _i (0,`n`-1) _i =` k `|` arr`'[_i] =` arr`[_i])`.

ExceptionDec(name, parent)

$$\frac{}{\begin{array}{l}\text{Globals} \cup (\text{name}, \texttt{(1)}, \texttt{0}) \\ \text{Globals} \cup (\text{name\_}\texttt{save}, \texttt{(1)}, \texttt{0}) \\ \text{Exceptions} \cup (\text{name}, \text{parent})\end{array}}$$

Exceptions ∪ (name, parent)

Exception declarations are entered into the Exceptions environment, which maintains the subtyping hierarchy of the exceptions for use in rules to be described later. Each exception declaration is also entered into the Globals environment, as a non-array variable with one bit allocated for it, represented by the string `(1)`, and an initalizer string of `0`. We translate exceptions into boolean variables that start false, and are set to true as described below in the rule for **Throw**.

ProcedureDec(pname, paramNameList, body)

$$\begin{array}{l}\text{pname.Locals} = \emptyset \\ \text{paramNameList} \mapsto_{stringList} paramList \\ \text{pname.Locals} \vdash (\text{body}, \text{pname}, \text{nextstate} = 1, \text{exjump} = 0) \mapsto_{stmt} (bodyCode, N) \\ <\ code\ below\ > \mapsto code \\ \text{Procedures} \cup (\text{pname}, code, paramList, \text{pname.Locals})\end{array}$$

```
q <pname0> --> q <> "exceptional return" (frameGlobals)
bodyCode
q <pnameN> --> q <> "normal return" (frameGlobals)
```

Procedure declarations are entered into the Procedures environment. Each entry in the Procedures environment contains the name of the procedure, the Moped code corresponding to the procedure, a list of the names of the formal parameters, and a variable environment named Locals. This environment is identical in structure to the Globals environment described above.

We define the zeroth states to be the "exceptional return" state, and the last state to be the procedure is the "normal return" state. The moped code `q <>` pops the pushdown stack, and transitions to the popped state.

The *stringList* translation just produces a list of strings representing the names of the formal parameters. This list will be stored for later use in building the assignment expression.

ProcedureDec(`main`, _, body)

pname.Locals= $\emptyset$
pname.Locals $\vdash$ (body, `main`, nextstate = 1, exjump = 0) $\mapsto_{stmt}$ $(bodyCode, N)$
$<$ *code below* $>$ $\mapsto$ *code*
Procedures $\cup$(`main`, *code*, *paramList*, pname.Locals)

`q <main0> --> q <exnend> "exceptional main return" (`*frameGlobals*`)`
bodyCode
`q <main`N`> --> q <normalend> "normal main return" (`*frameGlobals*`)`

All IEL programs must contain a `main` function, following C conventions, and this main function is translated slightly differently. The zeroeth and last states transition to the `normalend` and `exnend` states described in the **Start** rule instead of popping the call stack.

4.4.2 *stmt* Translation Rules

The *stmt* translation takes four arguments. The first argument is a part of the IEL abstract syntax tree representing a statement. The second argument is the name of the procedure containing the statement. The third argument is the next available state number for use in numbering the Moped states corresponding to the statement. This argument is set to 1 at the beginning of each new procedure. The fourth argument is the number of a state in the current procedure to jump to when an exception is thrown. This argument is set to 0 at the beginning of each new procedure. This indicates that in the event of an exception being thrown, the system will transition to the state *pname*0, which will return from the current procedure.

The *stmt* translation returns two elements. The most important returned element is a string containing the Moped code corresponding to the statement. These code fragments are concatenated together by the **BlockStmt** rule below. The other returned element is the number immediately after the last state number used in that rule.

BlockStmt(**Stmt**(stmt), stmtList), pname, ns, exj

(stmt, pname, ns, exj) $\mapsto_{stmt}$ $(stmtCode, N)$
(stmtList, pname, N, exj) $\mapsto_{stmtList}$ $(stmtListCode, N')$

---

stmtCode
stmtListCode

---

nextstate = N'

BlockStmt(**VarDec**(name, ty, init), stmtList), pname, ns, exj

ty $\mapsto_{ty}$ *tyDec*
init $\mapsto_{exp}$ *initExp*
(name, *tyDec*, *initExp*) $\mapsto_{varInitAssign}$ *varInitAssign*
(stmtList, pname, ns+1, exj) $\mapsto_{stmtList}$ $(stmtListCode, N')$
pname.Locals $\cup(name, tyDec, initExp)$
Globals $\cup$ pname.Locals $\cap$ $\neg$name $\mapsto_{frame}$ *initFrame*

`q <pname(ns)> --> q <pname(ns+1)> "init assign" (`*varInitAssign* `&`
    initFrame`)`
stmtListCode

nextstate = N'

IEL has block structure. Elements allowed within a block statement are other statements and variable declarations. These statements are translated in order, and the "nextstate" counter is increased as described in each specific *stmt* rule.

Variable declarations are stored in the local variable environment for the enclosing procedure, and are translated into a transition that initializes the variable to a given expression. This time, the frame expression includes both global and local variables, excluding the variable being initialized. We will use *frameAllExcept*(name) without reference as a shorthand for the "Globals $\cup$ pname.Locals $\cap$ $\neg$name $\mapsto_{frame}$ *initFrame*" translation.

ReturnStmt(), pname, ns, exj

`q <pname(ns)> --> q <> "explicit return" (`*frameGlobals*`)`

nextstate = ns+1

AssignStmt(varRef, value), pname, ns, exj

value $\mapsto_{exp}$ *valueExp*
(varRef, *valueExp*) $\mapsto_{varAssign}$ *varAssign*

`q <pname(ns)> --> q <pname(ns+1)> "assign" (`*varAssign* `&`
    frameAllExcept(varRef)`)`

nextstate = ns+1

This rule uses assign statements as described in the **VarDec** rule, to set the value of a single variable.

IfStmt(check, thenClause), pname, ns, exj

check $\mapsto_{exp}$ *checkExp*
(thenClause, pname, ns+1, exj) $\mapsto_{stmt}$ $(thenCode, N)$
Globals $\cup$ pname.Locals $\mapsto_{frame}$ *frameAll*

`q <pname(ns)> --> q <pname(ns+1)> "if true" (`*checkExp* `&` *frameAll*`)`
`q <pname(ns)> --> q <pname(`N`)> "if false" (!(`*checkExp*`) &` *frameAll*`)`
thenCode

nextstate = N

In translating if-then statements, we use multiple transitions out of a starting state. An expression and a negated expression distinguish the two branches.

We will use $frameAll$ without reference as a shorthand notation for the frame axiom applied to both global and local variables.

IfStmt(check, thenClause, elseClause), pname, ns, exj

---

check $\mapsto_{exp}$ $checkExp$
(thenClause, pname, ns+1, exj) $\mapsto_{stmt}$ $(thenCode, N)$
(elseClause, pname, N+1, exj) $\mapsto_{stmt}$ $(elseCode, N')$

`q <pname(ns)> --> q <pname(ns+1)> "if true" (`$checkExp$` & `$frameAll$`)`
`q <pname(ns)> --> q <pname(`$N+1$`)> "if false" (!(`$checkExp$`) & `$frameAll$`)`
$thenCode$
`q <`$pname(N)$`> --> q <`$pname(N')$`> "jump past else" (`$frameAll$`)`
$elseCode$

---

nextstate = N'

WhileStmt(check, body), pname, ns, exj

---

check $\mapsto_{exp}$ $checkExp$
(body, pname, ns+1, exj) $\mapsto_{stmt}$ $(bodyCode, N)$

`q <pname(ns)> --> q <pname(ns+1)> "while" (`$checkExp$` & `$frameAll$`)`
`q <pname(ns)> --> q <pname(`N`+1)> "not while" (!(`$checkExp$`) & `$frameAll$`)`
$bodyCode$
`q <pname(N)> --> q <pname(ns)> "loop while" (`$frameAll$`)`

---

nextstate = $N+1$

The only loops included in IEL are while loops. These are translated into a bidirectional test, and a looping state transition at the end of the while body.

CallStmt(procName, argExpList), pname, ns, exjump

Procedures$_{\text{procName}}$ $\mapsto_{procParamList}$ $paramNames$
argExpList $\mapsto_{expList}$ $argExps$
$(paramNames, argExps)$ $\mapsto_{varAssignList}$ $paramAssignCode$
pname.Locals $\mapsto_{frameCall}$ $frameLocalsForCall$

`q <pname(ns)> --> q <procName1 pname(ns+1)> "call procName"`
`    (`$paramAssignCode$` & `$frameGlobals$` & `$frameLocalsForCall$`)`
`q <pname(ns+1)> --> q <pname(ns+2)> "normal ret"`
`    (!(Exception = 1) & `$frameAll$`)`
`q <pname(ns+1)> --> q <pname(exjump)> "exceptional ret"`
`    (Exception = 1 & `$frameAll$`)`

---

nextstate = ns+2

Procedure calls make use of Moped's pushdown stack. The syntax above indicates a transition to $procName1$, while pushing the state "pname(ns+1)" onto the stack, for later execution.

We implement exception propagation by following each call with a test to see if an exception was thrown within that procedure. If so, we transition to the state currently indicated as the exception handler. If not, we move on.

The *procParamList* translation extracts the names of the formal parameters of the procedure. The *expList* translation translates each expression into a Moped string, and returns a list. The *varAssignList* produces a string containing a series of `&`-separated assignment expressions as described earlier.

The *frameCall* translation acts similarly to the standard *frame* translation. Because we are pushing the next state onto the stack, we must indicate that the variables will not change when that state is popped by a later return statement. Moped indicates an assignment to variables in a pushed state with two primes after the variable name. *frameCall* builds the standard frame expression with two primes instead of one, as in `var'' = var`.

ThrowStmt(exnName), pname, ns, exj

$\text{Exceptions}_{\text{exnName}} \mapsto_{exnAncestors} exnAncestorSet$

```
q <pname(ns)> --> q <pname(exj)> "throwing exnName"
    (exnAncestorSet_1' = 1 & ... & exnAncestorSet_k' = 1
    frameAllExcept(exnAncestorSet))
```

nextstate = ns+1

A throw statement makes use of the exception ancestry hierarchy, to set the thrown exception and all ancestors to one. This enables us to test the root `Exception` variable to find out if any exception has been thrown, and allows a catch block to catch all subclassed exceptions.

TryStmt(body, catchList), pname, ns, exj

(catchList, pname, ns+2, exj, resumejump = ns+1) $\mapsto_{catch}$ $(catchCode, N)$
(body, pname, N, exjump = ns+2) $\mapsto_{stmt}$ $(bodyCode, N')$

```
q <pname(ns)> --> q <pname(N)> "jump past catch" (frameAll)
q <pname(ns+1)> --> q <pname(N')> "jump after try block" (frameAll)
catchCode
bodyCode
```

nextstate = N'

Statements within a try block are translated with a new exception handling state, namely the first associated catch block. The catch block is translated with another parameter, a state for resuming after handling an exception.

4.4.3 *cstmt* and *catch* Translation Rules

CatchStmt(name, body), pname, ns, exj, resj

(body, pname, ns+2, exj) $\mapsto_{stmt}$ $(bodyCode, N)$
Exceptions $\mapsto_{exnAll}$ $allExnSet$

`q` <pname(ns)> `--> q` <pname(ns+1)> `"caught exn" (`*name* `= 1 &` *frameAll*`)`
`q` <pname(ns)> `--> q` <pname(N+1)> `"didn't catch exn"`
  `(!(`*name* `= 1) &` *frameAll*`)`
`q` <pname(ns+1)> `--> q` <pname(ns+2)> `"clear exns"`
  `(`$allExnSet_1$`' = 0 & ...&` $allExnSet_k$`' = 0 &` $frameAllExcept(allExnSet)$`)`
$bodyCode$
`q` <pname(N)> `--> q` <pname(resj)> `"resume normally" (`*frameAll*`)`

---

nextstate $= N + 1$

The named exception is tested, and if the exception is not set, then control transfers to the next catch block. If this is the last catch block, control will transfer to a state that transfers to the current exception handler, propagating it upwards. If the exception is set, then all exceptions are cleared and the catch body executes, transitioning to the resume state upon completion.

CatchList(catchBlock, catchList), pname, ns, exj, resj

(catchBlock, pname, ns, exj, resj) $\mapsto_{cstmt}$ $(catchCode, N)$
(catchList, pname, N, exj, resj) $\mapsto_{catch}$ $(catchListCode, N')$

$catchCode$
$catchListCode$

---

nextstate $= N'$

CatchList(catchBlock), pname, ns, exj, resj

(catchBlock, pname, ns, exj, resj) $\mapsto_{cstmt}$ $(catchCode, N)$

$catchCode$
`q` <pname(N)> `--> q` <pname(exj)> `"unhandled exn" (`*frameAll*`)`

---

nextstate $= N + 1$

The last catch block is followed by a statement that propagates the exception upward, indicating an unhandled exception.

4.4.4 Finally Block *stmt* Translation Rules

TryStmt(body, finally), pname, ns, exj

(finally, pname, ns+3, exj) $\mapsto_{stmt}$ $(finallyCode, N)$
(body, pname, $N+2$, exjump = ns+1) $\mapsto_{stmt}$ $(bodyCode, N')$
Exceptions $\mapsto_{exnAll}$ $allExnSet$
Exceptions $\mapsto_{exnSaveAll}$ $allExnSaveSet$
$(allExnSaveSet, allExnSet) \mapsto_{varAssignList} exnSave$
$(allExnSet, 0) \mapsto_{varAssignList} exnClear$
$(allExnSet, allExnSaveSet) \mapsto_{varAssignList} exnRestore$

```
q <pname(ns)> --> q <pname(N+2)> "jump to try body" (frameAll)
q <pname(ns+1)> --> q <pname(ns+2)> "save exns"
    (exnSave & frameAllExcept(allExnSaveSet))
q <pname(ns+2)> --> q <pname(ns+3)> "clear exns"
    (exnClear & frameAllExcept(allExnSet))
finallyCode
q <pname(N)> --> q <pname(N+1)> "restore exns"
    (exnRestore & frameAllExcept(allExnSet))
q <pname(N+1)> --> q <pname(N'+1)> "no exn before finally"
    (!(Exception = 1) & frameAll)
q <pname(N+1)> --> q <pname(exj)> "exn before finally"
    (Exception = 1 & frameAll)
bodyCode
q <pname(N')> --> q <pname(ns+1)> "run finally" (frameAll)
```

nextstate = $N' + 1$

TryStmt(body, catchList, finally), pname, ns, exj

(finally, pname, ns+3, exj) $\mapsto_{stmt}$ $(finallyCode, N)$
(catchList, pname, N+2, ns+1, resumejump = ns+1) $\mapsto_{catch}$ $(catchCode, N')$
(body, pname, N', exjump = N+2) $\mapsto_{stmt}$ $(bodyCode, N'')$
Exceptions $\mapsto_{exnAll}$ $allExnSet$
Exceptions $\mapsto_{exnSaveAll}$ $allExnSaveSet$
$(allExnSaveSet, allExnSet)$ $\mapsto_{varAssignList}$ $exnSave$
$(allExnSet, 0)$ $\mapsto_{varAssignList}$ $exnClear$
$(allExnSet, allExnSaveSet)$ $\mapsto_{varAssignList}$ $exnRestore$

```
q <pname(ns)> --> q <pname(N')> "jump to try body" (frameAll)
q <pname(ns+1)> --> q <pname(ns+2)> "save exns"
   (exnSave & frameAllExcept(allExnSaveSet))
q <pname(ns+2)> --> q <pname(ns+3)> "clear exns"
   (exnClear & frameAllExcept(allExnSet))
finallyCode
q <pname(N)> --> q <pname(N+1)> "restore exns"
   (exnRestore & frameAllExcept(allExnSet))
q <pname(N+1)> --> q <pname(N"+1)> "no exn before finally"
   (!(Exception = 1) & frameAll)
q <pname(N+1)> --> q <pname(exj)> "exn before finally"
   (Exception = 1 & frameAll)
catchCode
bodyCode
q <pname(N")> --> q <pname(ns+1)> "run finally" (frameAll)
```

nextstate = $N'' + 1$

This rule applies the exception-saving technique explained earlier, supporting a single level of finally blocks only. Before the finally block executes, all of the exception variables are assigned to the exception-save variables, and the exception variables are cleared. If the finally block exits normally, all of the exception-save variables are assigned back to the exception variables, and a test is made that will propagate an exception if one exists. Note that the exceptional jump variable for the finally block is the previously existing value of “exjump”. If an exception is thrown within the finally block, then control will immediately translate to the next outer handler without restoring the saved exceptions.

4.4.5 Assertion *stmt* Translation Rules

AssertStmt(name, exp), pname, ns, exj

exp $\mapsto_{exp}$ $assertExp$

```
q <pname(ns)> --> q <name> (assertExp & frameAll)
q <pname(ns)> --> q <name_fail> (!(assertExp) & frameAll)
q <name> --> q <pname(ns+1)> (frameAll)
q <name_fail> --> q <name_fail> (frameAll)
```

nextstate = ns+1

CheckStmt(name, exp), pname, ns, exj

exp $\mapsto_{exp}$ *assertExp*

```
q <pname(ns)> --> q <name> (assertExp & frameAll)
q <pname(ns)> --> q <name_fail> (!(assertExp) & frameAll)
q <name> --> q <pname(ns+1)> (frameAll)
q <name_fail> --> q <pname(ns+1)> (frameAll)
```

nextstate = ns+1

We translate the assert and check statements into tests that pass through a state called "name", named by the user, if the expression is true. If the expression is false, they pass through a state called "name"`_fail`. In the assert statement, the failure state loops infinitely, halting the execution of the program. In the check statement, the failure state is only passed through. These states are designed to be included as propositions in LTL formulas. The examples section demonstrates how they may be used by the programmer.

4.4.6 *exp* and *ty* Translation Rules

These rules produce less complex output, and so are presented in standard inference rule form. Actual moped code is in typewriter face, and variables are in italic face.

$$\frac{}{IntExp(i) \mapsto_{exp} i}$$

$$\frac{}{BoolExp(true) \mapsto_{exp} \texttt{1}}$$

$$\frac{}{BoolExp(false) \mapsto_{exp} \texttt{0}}$$

$$\frac{var \mapsto_{var} var'}{VarExp(var) \mapsto_{exp} var'}$$

$$\frac{exp_1 \mapsto_{exp} exp_1' \quad op \mapsto_{op} op' \quad exp_2 \mapsto_{exp} exp_2'}{OpExp(exp_1, op, exp_2) \mapsto_{exp} exp_1' \; op' \; exp_2'}$$

This table defines the operator symbol translation.

op translation

IEL Op	Moped Op
+	+
-	-
*	*
/	/
&&	&
||	|
<	<
>	>
<=	<=
>=	>=
=	=
!=	!=
!	!

$$\frac{}{SimpleVar(name) \mapsto_{var} name}$$

$$\frac{var \mapsto_{var} var' \quad index \mapsto_{exp} index'}{ArrayVar(var, index) \mapsto_{var} var'\texttt{[}index'\texttt{]}}$$

$$\frac{}{IntDec(bits) \mapsto_{ty} \texttt{(}bits\texttt{)}}$$

$$\frac{}{BoolDec() \mapsto_{ty} \texttt{(1)}}$$

$$\frac{ty \mapsto_{ty} ty'}{ArrayDec(size, ty) \mapsto_{ty} \texttt{[}size\texttt{]}ty'}$$

5 Examples

Our first example, a *Single Resource* that can be locked or unlocked, illustrates the basic constructs of IEL and how even simple exceptional flow of control can be subtle to reason about. Our second example, the *N Queens* Problem, is a standard search example illustrating an interesting use of exceptions, in particular for effecting both shallow and deep backtracking. We use the third example, *Vending Machine*, taken from Sinha and Harrold's paper, to show how our Exceptional Tool Suite works from end to end. Our last example, *Finally Blocks*, is a simple demonstration of our finally block handling.

5.1 Single Resource

This first example is a simple locked resource example. In the program, the resource is locked, and then an exception is thrown by a subroutine. This exception is caught, and the resource is unlocked. The program then loops. We have commented out the call to `unlock` in the exception handler, thus causing the program to call `lock` twice in a row.

```
var locked: int

exception an_exception

procedure main() {
  locked := 0
  while true=true {
    try {
      lock()
      randomException()
      unlock()
    }
    catch an_exception {
      /* If you uncomment out the following, the program is OK */
      /* unlock() */
    }
  }
}

procedure randomException() {
  throw an_exception
}

procedure lock() {
  if locked = 1 then
    error()
  if locked = 0 then
    locked := 1
}

procedure unlock() {
  if locked = 0 then
    error()
  if locked = 1 then
    locked := 0
}

procedure error() {
  __assert error true=true
}
```

We have inserted a check at the beginning of `lock` and `unlock` to ensure that the lock is in the proper state before changing it. We name the error condition in order to reference it in an LTL formula by using the IEL `__assert` statement with a name of `error` and an expression that always evaluates to true.

We translate the IEL program into input for our model checker, and check the validity of the following LTL formula:

```
!<>error
```

The model checker returns false, indicating that an error exists in the program. We can determine where the error lies from examining the state trace outputted by the model checker. After uncommenting the noted line above, the model checker returns true.

5.2 N Queens

Next, we turn our attention to the N-queens problem. The program below solves N-queens using exceptions to implement a backtracking search. We can use our tool to determine whether N-queens is solvable for a given value of N, by inputting the following formula into our model checker:

```
<>normalend
```

If the checker returns true, then eventually the program will end normally, and the problem is solvable for the specific N. If not, then the program must end exceptionally.

```
var qj: array of int[4]
var conflict: bool

exception Conflict

procedure conflict(i: int, j: int, n: int) {
  var qi:int := 0
  conflict := false

  while (qi < i) {
    if (i = qi || j = qj[qi] || (i+j) = (qi+qj[qi]) ||
        (i-j) = (qi-qj[qi])) then {
      conflict := true
      return
    }
  qi := qi + 1
  }
}
```

```
procedure addqueen(i1: int, j1: int, n1: int) {
  while (j1 <= n1) {
    try {
      conflict(i1, j1, n1)
      if (conflict = true) then throw Conflict
      else {
        qj[i1] := j1
        if (i1 != n1) then addqueen(i1+1, 0, n1)
        return
      }
    }
    catch Conflict {
      if (j1 = n1) then throw Conflict
    }
    j1 := j1 + 1
  }
}

procedure main() {
  var n_queens: int := 4
  addqueen(0, 0, n_queens-1)
}
```

Our model checker returns true for N=1 and N greater than or equal to 4, as is expected. We have tested values of N from one to twelve.

5.3 Vending Machine

This example is a simplified control program for a vending machine, written in Java. We use this example to demonstrate how our toolset handles try-catch constructs and data. The full source code is presented in Appendix A, with auxilliary classes omitted for brevity.

The first step in our toolset translates this object-oriented Java program to the simplified and procedural IEL. The IEL source is presented below.

```
/* Vending machine IEL code */
exception IllegalAmountException
exception IllegalCoinException
exception ZeroValueException
exception SelectionException
exception IllegalSelectionException extends SelectionException
exception SelectionNotAvailableException extends SelectionException

const VendingMachine_Dispenser_MIN_SELECTION 1
const VendingMachine_Dispenser_MAX_SELECTION 6
```

```
/* Various constants, definitions not given */
const VendingMachine_MAX_ATTEMPTS 5
const VendingMachine_INSERT 0
const VendingMachine_VEND 1
const VendingMachine_RETURN 2
const VendingMachine_action 0
const VendingMachine_coin 5
const VendingMachine_selection 3

/* There is an instance variable defined here for each instance of a
   class that is instantiated. Since this program only creates one
   VendingMachine object, there is one set of instance variables. */
var VendingMachine_totValue_1:int
var VendingMachine_currValue_1:int
var VendingMachine_currAttempts_1:int

var VendingMachine_valueOf_1_result:int

var VendingMachine_Dispenser_available_1_result:bool
var VendingMachine_Dispenser_value_1_result:int

/* constructor */
procedure VendingMachine_VendingMachine_1() {
  VendingMachine_totValue_1 := 0
  VendingMachine_currValue_1 := 0
  VendingMachine_currAttempts_1 := 0
}

procedure VendingMachine_insert_1(coin:int) {
  var value:int
  VendingMachine_valueOf_1(coin)
  value := VendingMachine_valueOf_1_result
  if (value = 0) then
    throw IllegalCoinException
  VendingMachine_currValue_1 := VendingMachine_currValue_1 + value
I:
}

procedure VendingMachine_valueOf_1(coin:int) {
  if( coin = 5 || coin = 10 || coin = 25 ) then {
    VendingMachine_valueOf_1_result := coin
    return
  }
  VendingMachine_valueOf_1_result := 0
}
```

```
procedure VendingMachine_returnCoins_1() {
  if (VendingMachine_currValue_1 = 0) then
    throw ZeroValueException
  VendingMachine_currValue_1 := 0
  VendingMachine_currAttempts_1 := 0
R:
}

procedure VendingMachine_vend_1(selection:int) {
  if (VendingMachine_currValue_1 = 0) then
    throw ZeroValueException
  try {
    VendingMachine_Dispenser_dispense_1(VendingMachine_currValue_1,selection)
    var bal:int
    VendingMachine_Dispenser_value_1(selection)
    bal := VendingMachine_Dispenser_value_1_result
    VendingMachine_totValue_1 :=
VendingMachine_totValue_1 + VendingMachine_currValue_1 - bal
    VendingMachine_currValue_1 := bal
    VendingMachine_returnCoins_1()
  }
  catch SelectionException {
AC:    VendingMachine_currAttempts_1 := VendingMachine_currAttempts_1 + 1
    if (VendingMachine_currAttempts_1 < VendingMachine_MAX_ATTEMPTS) then
      {}
    else {
      VendingMachine_currAttempts_1 := 0
      throw SelectionException
    }
  }
  catch ZeroValueException {
  }
}

procedure VendingMachine_Dispenser_dispense_1(currVal:int, sel:int) {
  if (sel < VendingMachine_Dispenser_MIN_SELECTION ||
sel > VendingMachine_Dispenser_MAX_SELECTION) then {
    throw IllegalSelectionException
  }
  else {
    VendingMachine_Dispenser_available_1(sel)
    if( VendingMachine_Dispenser_available_1_result = false) then {
      throw SelectionNotAvailableException
    }
    else {
      var val:int
      VendingMachine_Dispenser_value_1(sel)
      val := VendingMachine_Dispenser_value_1_result
      if (currVal < val) then
        throw IllegalAmountException
    }
  }
D:
}

procedure Dispenser_available_1(sel:int) {
  if( sel = 2 || sel = 4 ) then {
    Dispenser_available_1_result := false
    return
  }
  Dispenser_available_1_result := true
}

procedure Dispenser_value_1(sel:int) {
  Dispenser_value_1_result := 50
}

procedure main() {
  VendingMachine_VendingMachine_1()
  while(true=true) {
    try {
      try {
        if VendingMachine_action = VendingMachine_INSERT then {
          VendingMachine_insert_1(VendingMachine_coin)
        }
        else if VendingMachine_action = VendingMachine_VEND then
          VendingMachine_vend_1(VendingMachine_selection)
        else if VendingMachine_action = VendingMachine_RETURN then
          VendingMachine_returnCoins_1()
      }
      catch SelectionException {
VendingMachine_returnCoins_1()
      }
      catch IllegalCoinException {
MRC: VendingMachine_returnCoins_1()
      }
      catch IllegalAmountException {
      }
    }
    catch ZeroValueException {}
  }
```

```
}
```

We wish to verify the following property:

```
[](insert -> <>(returnCoins || dispense))
```

Once coins are inserted, we would like to be sure that the coins will be returned, or a product will be dispensed. We would like to ensure that the vending machine has no modes in which it will simply keep the user's money.

We begin by annotating the IEL source lines `I`, `R`, and `D` with assertions to mark these specific points in the program. This was also seen in the lock example above. Possible avenues for future improvement include allowing the developer to make these annotations in the Java source, instead of at the IEL level. This may be done through a class with methods that have no actual Java meaning, but are treated specially by the translator.

We then make use of the IEL `choice` statement, to introduce nondeterminism in order to model the user actions.

The main function becomes:

```
procedure main() {
  VendingMachine_VendingMachine_1()
  while(true=true) {
  try {
    try {
        VendingMachine_action := choice[VendingMachine_INSERT,
                                        VendingMachine_VEND,
                                        VendingMachine_RETURN]

        if VendingMachine_action = VendingMachine_INSERT then {
  VendingMachine_coin := choice[5,10,25]
          VendingMachine_insert_1(VendingMachine_coin)
        }
        else if VendingMachine_action = VendingMachine_VEND then {
          VendingMachine_selection := choice[1,2,3,4,5,6,7]
          VendingMachine_vend_1(VendingMachine_selection)
}
        else if VendingMachine_action = VendingMachine_RETURN then {
          VendingMachine_returnCoins_1()
}
 ...
}
```

For this program and the above formula, our model checker returns false. Examining the trace, we see that the IllegalAmountException is continually thrown up to the main level, after the action choice statement selects VEND. We infer that if the amount of money currently in the vending machine does not match the price of any item, then the user can continually press vend and never recieve either product or coins. A somewhat trivial example, yes, but it is a violation of the above LTL formula.

We change the program to increase the "attempts" counter when an IllegalAmountException is thrown, similar to that done in line `AC`. We also change the program to return the coins if that exception propagates up to the level of the main procedure, similar to line `MRC`. The user can no longer select VEND an unbounded number of times. This revised program now satisfies the formula.

5.4 Finally Blocks

Our last example is a simple program that uses a finally block.

```
exception exnA
exception exnB

procedure throwA()
{
  throw exnA
}

procedure throwB()
{
  throw exnB
}

procedure main()
{
  try {
    throwA()
  } catch exnA {
    throwB()
    throwA()
  } finally {
    try {
      throwA()
    } catch exnB {
      /* do nothing */
    }
  }
}
```

This program demonstrates exceptional entry into finally blocks, when exception B is thrown from the first catch block. This program also demonstrates an how exception thrown and not caught in the finally block can supersede an exception thrown prior to entering the finally block.

We use this LTL formula: `<>normalend`. As we expect, the model check returns false. This program can only end exceptionally. Examining the backtrace, we see that the program ends with exception A. We see exception B in the save variables described earlier.

```
  [ exceptional termination ]
q (Exception=1 & Exception_save=1 & exnA=1 & exnA_save=0 & exnB=0
    & exnB_save=1)
 <exnend>
```

Our model checker behaves correctly on this example.

6 Future Work

- Concurrent threads of control.
  - Use assume/guarantee approach but need to develop proof rules to handle exceptions.
  - Develop model checker for more powerful automaton. Restrict class of programs and/or properties to check, to make this practical.

7 Appendix A: Java Source for Vending Machine Example

```
public class VendingMachine {
    private int totValue;
    private int currValue;
    private int currAttempts;
    private Dispenser d;

    final int MAX_ATTEMPTS = 5;
    static final int INSERT = 0;
    static final int VEND = 1;
    static final int RETURN = 2;
    static final int action = 0;
    static final int coin = 27;
    static final int selection = 3;

    public VendingMachine() {
        totValue = 0;
        currValue = 0;
        currAttempts = 0;
        d = new Dispenser();
    }
```

```
public void insert( int coin ) throws IllegalCoinException {
    int value = valueOf( coin );
    if( value == 0 )
        throw new IllegalCoinException();
    currValue += value;
}

public int valueOf( int coin ) {
    if( coin == 5 || coin == 10 || coin == 25 )
        return coin;
    return 0;
}

public void returnCoins() throws ZeroValueException {
    if( currValue == 0 )
        throw new ZeroValueException();
    currValue = 0;
    currAttempts = 0;
}

public void vend( int selection ) throws Exception {
    if( currValue == 0 )
        throw new ZeroValueException();
    try {
        d.dispense( currValue, selection );
        int bal = d.value( selection );
        totValue += currValue - bal;
        currValue = bal;
        returnCoins();
    }
    catch( SelectionException s ) {
        currAttempts++;
        if( currAttempts < MAX_ATTEMPTS ) {
            // original example prints message
        } else {
            currAttempts = 0;
            throw new SelectionException();
        }
    }
    catch( ZeroValueException z ) {
    }
}

public static void main( String argv[] ) throws Exception {
    VendingMachine vm = new VendingMachine();
    while( true ) {
            try {
                try {
                    switch( action ) {
                    case INSERT: vm.insert( coin ); break;
                    case VEND: vm.vend( selection ); break;
                    case RETURN: vm.returnCoins(); break;
                    }
                }
                catch( SelectionException s ) {
                    vm.returnCoins();
                }
                catch( IllegalCoinException i ) {
                    vm.returnCoins();
                }
                catch( IllegalAmountException i ) {
                    // original example prints message
                }
            }
            catch( ZeroValueException z ) {
            }
        }
    }
}

public class Dispenser {
    final int MIN_SELECTION = 1;
    final int MAX_SELECTION = 6;

    public void dispense( int currVal, int sel ) throws Exception {
        if( sel < MIN_SELECTION || sel > MAX_SELECTION )
            throw new IllegalSelectionException();
        else
            if( !available( sel ) )
                throw new SelectionNotAvailableException();
            else {
                int val = value( sel );
                if( currVal < val )
                    throw new IllegalAmountException();
            }
    }

    public boolean available( int sel ) {
        if( sel == 2 || sel == 4 )
            return false;
        return true;
    }
    public int value( int sel ) {
        return 50;
    }
}
```

References

[BM00] Peter A. Buhr and W.Y. Russell Mok. Advanced exception handling mechnanisms. *IEEE Transactions in Software Engineering*, 26(9):820–836, September 2000.

[BR01] Thomas Ball and Sriam K. Rajamani. The slam toolkit. In *Proceedings of CAV'01*, pages 260–264, Paris, France, July 2001. Springer-Verlag. LNCS 2102.

[CDHR00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings of the SPIN Software Model Checking Workshop*. Springer-Verlag, August 2000. LNCS.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV'00*. Springer-Verlag, July 2000. LNCS 1855.

[EKS00] Javier Esparza, Antonin Kucera, and Stefan Schwoon. Model-checking ltl with regular valuations for pushdown systems. In *Proceedings of TACS'01*. Springer-Verlag, October 2000. LNCS 2215.

[Hol97] Gerard Holzmann. The model checker spin. *IEEE Trans. on Soft. Eng.*, 23(5):1–17, May 1997.

[LNS00] K. Rustan M. Leno, Greg Nelson, and James B. Saxe. Esc/java user's manual. Technical Report Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[McM93] Ken McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[Sch] Stefan Schwoon. Moped web site. `http://wwwbrauer.informatik.tu-muenchen.de/~schwoon/moped/`.

[SH00] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions in Software Engineering*, 26(9):849–871, September 2000.