Mondex, an electronic purse : specification and refinement checks with the *Alloy* model-finding method

Tahina Ramananandro[*]

November 6, 2007

Abstract

This paper explains how the *Alloy* model-finding method has been used to check the specification of an electronic purse (also called smart card) system, called the *Mondex* case study, initially written in Z. After describing the payment protocol between two electronic purses, and presenting an overview of the Alloy model-finding method, this paper explains how technical issues about integers and conceptual issues about the object layout in Z have been tackled in Alloy, giving general methods that can be used in most case studies with Alloy. This work has also pointed out some significant bugs in the original Z specification such as reasoning bugs in the proofs, and proposes a way to solve them.

Keywords : *Alloy*, formal methods, model-finding, *Mondex* electronic purse, refinement, security properties, software verification, specification.

1 Introduction

1.1 The *Mondex* case study

In 1994, National Westminster Bank developed an electronic purse (or *smart card*) system, called *Mondex* [MCS, Mon]. An electronic purse is a card-sized device intended to replace "real" coins with electronic cash. In contrast to a credit or debit card, an electronic purse stores its balance in itself, thus does not necessarily require any network access to update a remote database during a transaction. So, electronic purses can be used in small stores or shops, such as bakeries, where small amounts of money are involved.

But everything regarding cash requires a critically high security level. So, in 1998, National Westminster Bank asked researchers to verify security properties about *Mondex*:

- any value must be accounted; in particular, in case of a failed transaction, lost value must be logged (it is necessary, but the converse is not true);
- no money may suddenly appear on a purse without being debited from another purse through an achieved transaction.

In fact, the loss of money is a global property that cannot be considered at the local scale of one purse.

This research led to a formal proof by hand of the *Mondex* electronic purse system[1] with the Z specification language [Spi92, WD96]. This proof has been published in 2000 by Susan Stepney, David Cooper and Jim Woodcock [SCW00]. It has critically helped the *Mondex* system be granted ITSEC security level 6 out of 6.

This proof consists in a specification relying on a refinement relation between two models:

- the abstract model, a very simple model with an atomic transaction, and each purse storing the amount of its balance and the amount it has lost;
- the concrete model, which corresponds to the actual implementation with a non-atomic transaction protocol based on message exchange through an insecure communications channel.

Several security issues are raised by the *Concrete* protocol:

- a purse can be disconnected from the system too early;
- a message can be lost by the communications channel;
- a message can be replayed several times in the communications channel, but has to be read only at most once;
- a message can be read by any purse.

A *Concrete* transaction follows a 5-step protocol:

1. The "from" purse receives a initialization message.
2. The "to" purse receives a initialization message and sends a request message.
3. The "from" purse receives the request message, decreases its balance and sends the value message.

[*]École Normale Supérieure — 45, rue d'Ulm — 75005 Paris (France) — Tahina.Ramananandro@ens.fr

[1]The whole system has been proved, except cryptographical issues

4. The “to” purse receives the value message, increases its balance and sends the acknowledgment message. It is done.

5. The “from” purse receives the acknowledgment message. It is done.

If the transaction cannot go on for some reason (for instance if one of the two purses is disconnected too early), then a mechanism of *abortion* is provided (that could occur after a timeout in the real world). Then, in abortion cases where money could be lost, aborting purses have to *log* the transaction details into a private logging archive, so that if a transaction is actually lost, then it has necessarily been logged. Later purses may also copy the contents of their private log to a global archive.

So, the system is nondeterministic, insofar as a purse can decide to abort instead of going on with the transaction. But in both cases, the specification assumes that, once purses are connected to the system, they behave correctly and follow the operation protocol. The specification also assumes that messages related to the protocol cannot be forged (they are “protected”, for instance cryptographically), they can only be replayed. However, other “foreign” messages can be forged.

The proof layout in the Z specification consists in showing that security properties hold for the *Abstract*, then refining the *Abstract* model by the *Concrete*. But, as the *Concrete* model is not constrained enough, refinement is made easier by making it two-step, through a *Between* world which has the same structure as the *Concrete* but is constrained. So:

- The *Between* is abstracted by the *Abstract* by computing the values stored by abstract purses corresponding to the *Between*; however, for each purse, those computations may involve several purses because of the logs. This proof is a backwards refinement involving a prophecy variable, *chosenLost*: among the set of transactions for which the “from” purse has already decreased its balance but the “to” purse has not increased its own one yet, no purse having aborted yet, some transactions are chosen in advance to be lost.

- The refinement of the *Between* by the *Concrete* is rather an invariant proof than a refinement proof. The proof layout is a forwards simulation.

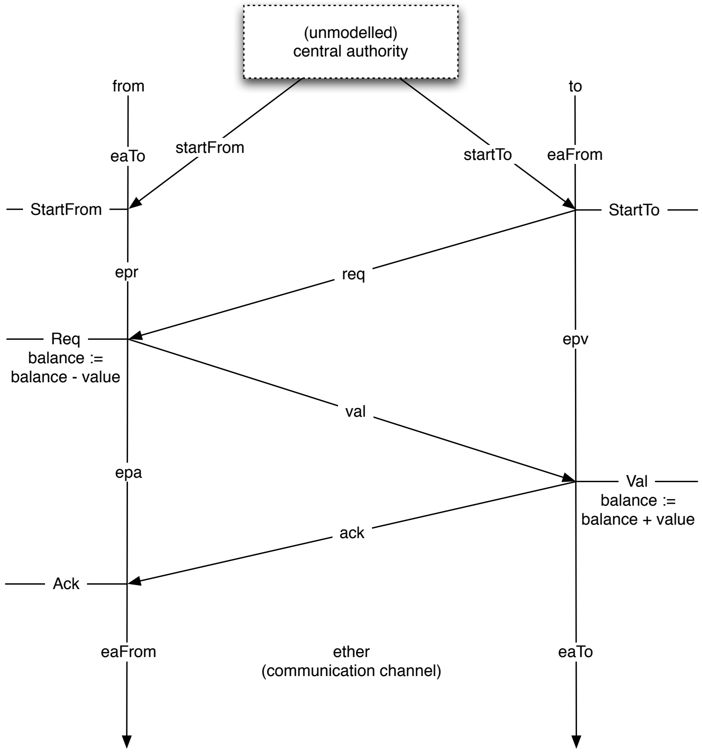

Figure 1: *Concrete 5-step protocol, with the statuses of the purses depending on the operations.*
Once purses are connected to the system, they are assumed to follow the protocol.
The central authority sending the startFrom *and* startTo *messages could correspond to pressing a button to initialize the transaction. It is not modelled: those messages spontaneously appear in the* ether.
The statuses eaTo *and* eaFrom *may be interpreted as a single “idle” status.*

1.2 The Alloy model-finding method

Alloy [Jac02, Jac06] is a modeling method that includes both a modeling language based on first-order logic and relational calculus including transitive closures, and a tool, called *Alloy Analyzer [AA]*[2] and based on model-finding through SAT-solving [Jac00], to analyze specifications in this language. The analysis consists in *checking* a theorem: the specification is translated into a SAT formula so that an instance of this formula corresponds to a *counterexample* to the theorem being checked.

[2]The *Alloy Analyzer* is the analysis engine for Alloy 3.0, with which the Mondex case study has been tackled. The new version of Alloy, 4.0, is based on another analysis tool, *Kodkod* [TJ07, Tor], which is a major improvement in translating specifications to SAT formulae.

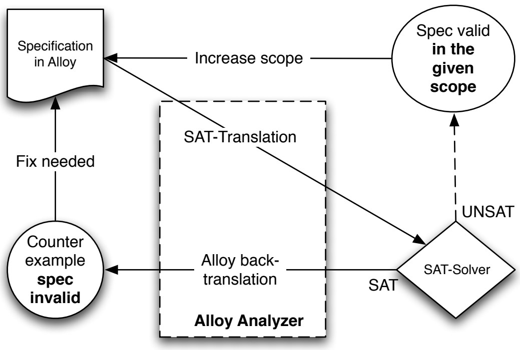

Figure 2: The *Alloy* model-finding method

A *model* of an Alloy specification is a set of *atoms*, or objects, satisfying all the *facts*, or axioms, in addition to the theorem being checked. The *scope* of the model is the cardinality of its atoms. All models considered by Alloy analyses are *finite*.

The Alloy modeling language is based on *relations*. A relation corresponds to a set of *tuples*, a tuple is an ordered combination of atoms. In Alloy, every relation has a fixed *arity*: in every relation, all the tuples have the same number of atoms.

Alloy provides the user with a relational calculus close to set theory: + (union), - (difference), **&** (intersection)... The cartesian product is denoted **->** . The most notable operator is the join operator denoted . : given two relations A of arity a and B of arity b, then $A.B$ corresponds to the following set:

$$A.B = \left\{ \left(\vec{a}, \vec{b}\right) : \exists\, x, (\vec{a}, x) \in A \wedge \left(x, \vec{b}\right) \in B \right\}$$

Special operators are also provided: ~ (for binary relations only) denotes the reciprocal relation (with the tuples turned upside down); s **<:** r (resp. r **:>** s) denotes the restriction of a relation r where the first (resp. last) components of its tuples are in the signature s.

Then, a formula consists in:

- a multiplicity formula to denote whether the relation corresponds to a non-empty set (**some**), an empty set (**no**), a singleton (**one**), a singleton or empty set (**lone**);
- an inclusion between two relations (**in**);
- a Boolean combination of formulae: {...} (and), **or**, **implies**, **not**...;
- a quantified formula: universal (**all**), existential (**some**), existential with unicity condition (**one**), unicity "if it exists" (**lone**), universal with negation (**no**). The **disj** keyword ensures the quantified variables to denote sets of tuples that are disjoint one to the other.

To construct relations of a given arity, it is necessary to declare unary relations, or *signatures*. Signatures correspond to sets of tuples. A signature is declared by **sig** *name*. A signature can also be declared as a subset of an existing signature thanks to the **in** keyword. But the user can also. Finally, the **abstract** keyword states that all the atoms of this signature belong to a signature extending it. In fact, a signature declared as **sig** *name* is implicitly considered extending the top-level abstract signature **object**.

module	::=	**module** *modname* [*signame**] *opendir** *declaration**
opendir	::=	**open** *modname* [*signame**]
declaration	::=		*sigdecl*	*preddecl*	*factdecl*	*fundecl*
sigdecl	::=		**abstract**? **sig** *signame extends*? {*args**}
			sig *signame* **in** *expr* {*args**}
extends	::=	**extends** *signame*
args	::=	*relname* : *mult*? *signame sigprodend** ,
sigprodend	::=	**->***mult*? *signame*
preddecl	::=	**pred** *predname* (*args**) *andformula*
factdecl	::=	**fact** *factname andformula*
fundecl	::=	**fun** *funname* (*args**) : *signame sigprodend** {*expr*}
expr	::=		*relname*	*funname* (*expr,**)	*expr*+*expr*
			expr-*expr*	*expr*-**>***expr*	*expr***&***expr*
			expr.*expr*	*expr*<:*expr*	*expr*:>*expr*	...
andformula	::=	{*formula**}
formula	::=		*predname*(*expr,**)	*mult expr*	*expr* **in** *expr*
			quant **disj**? *args args* andformula*	*andformula*
			andformula **implies** *andformula*
			andformula **or** *andformula*	...
mult	::=		**some**	**one**	**lone**	**no**
quant	::=		**all**	*mult*

Figure 3: The *Alloy* syntax (simplified)

The user can also declare a relation along with the signature of its first component : for instance, **sig** A $\{r$: $B\}$ declares a signature A and a binary relation r in A**->**B (that is, in $A \times B$). Multiplicity keywords as above can also occur, adding constraints on the relation.

The user can also declare predicates (**pred**) and functions (**fun**, with a "typing" indication about the return value) to factor the code of the specification.

The Alloy system is modular: a specification can be split in several module files beginning with **module** *name*. A module is then included via **open**. Modules can take signatures as parameters.

1.3 Overview of the main issues encountered while writing the Alloy specification

The use of the *Alloy* method (the *Alloy* specification language and the *Alloy Analyzer*) raised some issues at different levels, due to the logical conception of the *Alloy* specification language, or to the current implementation of the *Alloy Analyzer*. we solved them in two steps. First, we wrote a preliminary version of the *Mondex* specification in *Alloy* that used to follow the Z specification as close as possible. But this translation introduced several artifacts (such as useless signatures) that did not seem natural to the general ways of writing specifications in Alloy, so that we rewrote the specification to remove those artifacts. This "optimisation" has eventually pointed out some errors in the previous model.

On the one hand, we had to tackle rather technical issues about integers: whereas the Z specification frequently uses them, they are not well handled by the implementation of the *Alloy Analyzer*. But in fact, not all properties of integers are used, so that there are ways to represent the corresponding data more efficiently than with integers. This offers an interesting way to compute sums of sets of values without recursion on the set of values.

On the other hand, we had to tackle a more conceptual issue regarding how Z and Alloy treat the notion of the identity of objects. Whereas Z schemas define records, Alloy specifications define relations between objects that have their own identity. For instance, two abstract purses having the same *balance* and *lost* values are represented by the same record in Z, so they have to be distinguished somehow ; in Alloy, it is the converse : as different objects can have the same properties, constraints have to be added to consider them as records.

After tackling those modeling issues, the obtained models have been able to find bugs in the Z specification of the Mondex electronic purse. Those bugs are related to the insufficient formalism of that Z specification.

This work has been done within an internship at MIT [Ram06]; the Alloy model files are available on the author's website [Ram].

2 Representing integers with Alloy

The implementation of integers in the *Alloy Analyzer*[3] provides not very efficient analyses. Indeed, the translation of integers and their operations into boolean formulae consumes a lot of time and space, by building the whole arithmetic circuits, and dramatically reduces the definable scope.

The idea commonly retained by *Alloy* users, and also by the researchers who develop *Alloy* themselves (within Daniel Jackson's Software Design group) is that for most models written in *Alloy*, integers may be replaced with another representation providing similar properties, and which could fit the model better. This idea holds for the *Mondex* case study, so that author-level encodings may be used, as described in this section.

2.1 Using an order rather than sequence numbers

Sequence numbers are used to distinguish different transactions led by purses. In some way, they represent a *time scale* increasing whenever a transaction begins. It is not specified how this time scale increases: only the comparison relation is used. So, we only need an order to model them.

One idea is to use the ordering module provided along with the standard distribution of the *Alloy Analyzer*: `util/ordering`.

[3] The new engine *Kodkod* for Alloy 4.0 models integers in a different way through their binary representations. Few work has been done to translate the models to this new version.

```
sig SEQNO {}
open util/ordering [SEQNO]
```

Moreover, the *Alloy Analyzer* treats this module in an optimized way, in terms of symmetry breakings when building the SAT boolean formula, rather than explicitly defining the order.

2.2 Representing amounts through coins

Even though all the first-order properties of integers are used to model amounts, they are used in a particular way. Comparisons only occur between the pre-state and the post-state of an operation: either a purse decreasing its balance, or the whole global world balance, is concerned. In particular, two balances of *different* purses (associated to different names) are never compared.

The solution proposed by members of the SDG group, namely Emina Torlak and Derek Rayside, is to use *sets of coins* to represent an amount. The amount will not be represented by the cardinality of the set, but the coins themselves, as with *real* coins in *non-electronic* purses.

2.2.1 Computing with coins

With this approach, operations are redefined as follows :

- The sum of two values is the disjoint union of the corresponding sets of coins.
- The difference of two values is the (set) difference of the corresponding sets of coins, as soon as the set being subtracted is included in the original set.
- The comparison relation is the set inclusion between sets of coins.

Indeed, when a purse decreases its *balance*, it actually *gives away part of it*. So there is how the *Abstract* world can be defined:

```
sig NAME {}
sig Coin {}
sig AbPurse {
  balance : set Coin,
  lost : set Coin
}
sig AbWorld { abAuthPurse : NAME -> AbPurse }
```

This approach allows computing the sum of sets of values through simply *gathering* them with a relational expression. Whereas the Z specification defines a sum of set of values through a recursive definition:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Totals}} \\
\hline
totalBalance, totalLost : (NAME \twoheadrightarrow AbPurse) \rightarrow \mathbb{Z} \\
\hline
totalBalance(\varnothing) = 0 \\
totalLost(\varnothing) = 0 \\
\forall f : NAME \twoheadrightarrow AbPurse;\ name : NAME;\ AbPurse \mid \\
\quad name \in \operatorname{dom} f \land \theta AbPurse = f(name) \\
\bullet \\
\quad totalBalance(f) = totalBalance(name \lhd f) + balance \\
\quad \land\ totalLost(f) = totalLost(name \lhd f) + lost \\
\hline
\end{array}
$$

in Alloy, one would simply write `S.r` where S is a set of $NAME$s and r is a relation that maps a name to some coins. For instance, if a is an *AbWorld*, one would simply write `a.abAuthPurse.balance` to compute the sum of the balances of all abstract purses.

2.2.2 Defining constraints to avoid coin sharing

However, this approach requires to define additional constraints to avoid *coin sharing*, the fact that, for instance, two amounts being added could have common coins. Indeed, such constraints ensure, for instance, the considered sums actually being *disjoint* unions of sets of coins.

First constraints are added on the *Abstract* world. They are quite simple to express:

- There is no coin common to two purses, regardless of whether it would belong to the *balance* or the *lost* store of either purse. In other words, a coin must belong to at most one purse.
- There is no coin common to the *balance* store and the *lost* store of a purse. In other words, a coin must be either not lost, or lost.

```
fact noCoinSharing {
  all w : AbWorld {
    no disj n1, n2 : NAME {
      some n1.(w.abAuthPurse).(balance + lost)
      & n2.(w.abAuthPurse).(balance + lost)
    }
    no p : AbPurse {
      p in NAME.(w.abAuthPurse)
      some p.balance & p.lost
    }
  }
}
```

These constraints only apply to abstract authentic purses, that is purses actually belonging to an abstract world (although the second could even have been defined for any abstract purse).

Then, the Concrete purses also use coins:

```
sig PayDetails {
  from, to : NAME,
  fromSeqNo, toSeqNo : SEQNO,
  value : set Coin
}
sig ConPurse {
  name : NAME,
  balance : set Coin,
  pdAuth : PayDetails,
  exLog : set PayDetails,
  nextSeqNo : SEQNO,
  status : STATUS
}
sig ConWorld {
  conAuthPurse : NAME -> lone ConPurse,
  ether : set MESSAGE,
  archive : NAME -> PayDetails
}
```

The Concrete purse is defined with the *pdAuth* information on the pending transaction involving it, the *exLog* set of logged transactions, and the *status* of the purse in the execution of the pending transaction according to 1 on page 2.

Equivalent constraints to avoid coin sharing have to be added to the *Concrete* world. In the first model, we added the following constraints :

```
fact noCoinSharingConcrete {
  all p : ConPurse {
    no p.exLog.value & p.balance -- 1
  }
  all w : ConWorld {
    no disj n1, n2 : NAME {
      some n1.(w.conAuthPurse).balance
      & n2.(w.conAuthPurse).balance -- 2
    }
    no p : ConPurse, pd : PayDetails {
      p in NAME.(w.conAuthPurse)
      pd in NAME.archive
      some p.balance & pd.value -- 3
    }
  }
}
```

1. A purse has no coin common to its balance and a transaction it has logged to its *exLog*.
2. Two distinct purses have no coin common to their balances.
3. A purse has no coin common to its balance and a transaction that has been logged in the global *archive*.

But although constraint 2 makes sense, constraints 1 and 3 are too strong. Indeed, as regards constraint 3:

- Assume the “to” purse has received the money and sends the acknowledgment message. If the “from” purse aborts before receiving it, logging the transaction into its *exLog,* then this constraint prevents the “from” purse from copying the details relevant to this transaction to the global *archive.* Indeed, the “to” *balance* contains the coins corresponding to the *value* of the transaction.
- Assume the “to” purse has just sent the request message but aborts, then logging the transaction into its *exLog.* If the “from” purse aborts before receiving this message, then it will have kept the coins of the transaction value in its *balance.* Thus, the “to” purse will not be able to copy the details relevant to this transaction to the global *archive.*

```
fun allLogs (c : ConWorld) : ConPurse -> PayDetails
  { c.archive + (c.conAuthPurse <: exLog.c) }
fun authenticFrom (c : ConWorld) : set PayDetails
  { from.(c.conAuthPurse) }
fun authenticTo (c : ConWorld) : set PayDetails
  { to.(c.conAuthPurse) }
fun fromLogged (c : ConWorld) : set PayDetails
  { authenticFrom (c) & ConPurse.(allLogs (c) & ~from) }
fun toLogged (c : ConWorld) : set PayDetails
  { authenticTo (c) & ConPurse.(allLogs (c) & ~to) }
fun toInEpv (c : ConWorld) : set PayDetails
  { authenticTo (c) & to.status.c.epv
    & (iden & to.(pdAuth.c)).PayDetails }
fun fromInEpr (c : ConWorld) : set PayDetails
  { authenticFrom (c) & from.status.c.epr
    & (iden & from.(pdAuth.c)).PayDetails }
fun fromInEpa (c : ConWorld) : set PayDetails
  { authenticFrom (c) & from.status.c.epa
    & (iden & from.(pdAuth.c)).PayDetails }

fun definitelyLost (c : ConWorld) : set PayDetails
  { toLogged (c) & (fromLogged (c) + fromInEpa (c)) }

fun maybeLost (c : ConWorld) : set PayDetails
  { (fromInEpa (c) + fromLogged (c)) & toInEpv (c) }
```

Figure 4: Alloy definitions of the *definitelyLost* and *maybeLost* sets of transactions

In both cases, the corresponding transaction is "locked" in the *exLog*, which consequently cannot clear it through a *ClearExceptionLog* operation.

Roughly speaking, the point is to find constraints which could be equivalent to the abstract constraint preventing a coin to be "lost and not lost" at the same time. The solution may be found by referring to the *Abstract/Between* refinement relation. It relies on the definition of two functions :

- *definitelyLost* corresponds to the set of details referring to transactions definitely lost. Those transactions are either logged by the two purses, or logged by the "to" purse while the "from", having sent the money, is still expecting an acknowledgment.

- *maybeLost* corresponds to critically ambiguous transactions. The "to" purse expects the value. The "from" purse has already sent it, and either expects the acknowledgment, or has logged the transaction before the "to" received the value.

In both cases, we know that the value has been debited from the "from" balance but not yet credited to the "to" balance. Then, it is sound to replace constraint 3 above with the following one, stating that no coin in the value of a transaction in *definitelyLost* or *maybeLost* may be in a purse *balance* at the same time :

```
all w : ConWorld {
  no p : ConPurse {
    p in NAME.(w.conAuthPurse)
    some p.balance & (definitelyLost (w) + maybeLost (w)) -- new 3
  }
}
```

This constraint prevents a coin from being in a *balance* and a *lost* store at the same time, even if the purses are distinct.

As regards constraint 1, it is too strong if the following situation arises : the "to" purse logs the transaction just after sending the request, but the "from" aborts before receiving it (thus it does not log). Then, no money has been sent yet, but the transaction has been logged by the "to" purse. In that case, the "to" purse cannot receive the corresponding coins in a further transaction attempt involving them, because they are already in the logged transaction, even though they are still in the "from" balance.

All those situations have been found by counterexamples while trying to rewrite the Alloy specification. Indeed, those constraints were first defined within the *Concrete* world, so that the *Between/Concrete* refinements did not hold. Actually, some constraints in the *Between* were not necessarily kept through operations, so they were too strong. Thus, they caused some legal operations not to arise. The new constraints defined here have been moved to the *Between* world. This has allowed them to be checked as invariants through the *Between/Concrete* refinement.

2.2.3 Redefining *chosenLost* set : coins as a tracking system

Using coins has another interesting effect : they allow to better *track* the amounts through operations.

The *Abstract/Between* refinement relies on a prophecy variable, *chosenLost*, gathering the ambiguous (pending) transactions that are chosen *in advance* to be lost. This prophecy variable causes the protocol to be nondeterministic.

But this set, used to compute the *lost* values of the abstract purses, is uniquely known for a given *Between* world, as soon as the corresponding *Abstract* world is known. Indeed, thanks to the constraint preventing a coin from belonging to the values of two distinct transactions considered ambiguous, it is possible to determine to which transaction a coin corresponds. It is easily possible to show that the *definitelyLost* and *maybeLost* sets of transactions are disjoint (see definition above in Section ...). Indeed, for the former, the "to" purse has to have logged the transaction, whereas for the latter, the "to" purse has to be still waiting for the value being credited, which means that the transaction is still pending. It is also obvious that coins being accounted in the *Abstract* model correspond to either a concrete *balance*, or a *definitelyLost* or *maybeLost* transaction amount, the latter case including the case of a transaction chosen lost. So there are four solutions:

- the coin is in a concrete *balance*: then, it will be accounted into the abstract *balance* of the corresponding purse;

- the coin is in a *definitelyLost* transaction: then, it will be accounted into the *lost* of the "from" purse of this transaction;

- the coin is in a transaction considered *maybeLost*, but not chosen lost: then, it will be accounted into the *balance* of the "from" purse of this transaction;

- the coin is in a transaction considered *maybeLost* and chosen lost: then, it will be accounted into the *lost* of the "from" purse of this transaction.

Then, it is possible to "revolve" this table to define the *chosenLost* set. Just take the transactions of *maybeLost*, the coins of which are in an abstract *lost* :

```
fun getChosenLost (a : AbWorld, b : BetweenWorld) : PayDetails {
  NAME.(a.abAuthPurse).lost.(~value :> maybeLost (b))
}
```

3 Records in Z, objects in Alloy

3.1 The identity of objects

A major difference between Z and Alloy is how they represent objects. On the one hand, a Z schema defines records, so that two records having the same values denote the same object. On the other hand, Alloy specifications define relations between atomic objects, each of which has its own identity regardless of how it is related to others.

3.1.1 Simulating objects with Z records and names

In Z, an abstract purse is only a *record* with two fields, *balance* and *lost*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AbPurse}} \\
\hline
\textit{balance}, \textit{lost} : \mathbb{Z} \\
\hline
\textit{balance} \geq 0 \\
\textit{lost} \geq 0 \\
\hline
\end{array}$$

So, when two abstract purses have the same *balance* values and the same *lost* values, then it is impossible to distinguish them.

Now consider the definition of the *AbWorld* abstract world. It is a set of purses. To distinguish between two purses having the same values, the Z specification introduces *names*. This method is commonly retained in object-oriented Z specifications [Hal90].

$[\textit{NAME}]$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AbWorld}} \\
\hline
\textit{abAuthPurse} : \textit{NAME} \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; \textit{AbPurse} \\
\hline
\end{array}$$

In Alloy, the notion of property only corresponds to the way atomic objects are related to each other. That is why names are not necessary: keeping them would introduce an artifact in the Alloy specification. So the purses can simply be defined through the following signature:

```
sig AbPurse {
  balance: set Coin,
  lost: set Coin
}
```

So, an Abstract world is simply a set of purses, a subset of the `AbPurse` signature.

3.1.2 Simulating records with Alloy objects : canonicalization

Conversely, in Alloy, there is no notion of records and fields, as two distinct objects may be related to the same values by the relations.

However, it is possible to simulate Z's behaviour by introducing a notion of records in Alloy. One solution could be to *canonicalize* signatures: that is, to introduce canonicalization constraints which enforce two abstract worlds having the same properties to be equal :

The main purpose of this constraint would be to reduce the search space by eliminating redundant cases when analyzing the specification. However, such a canonicalization constraint may be also necessary for the *Abstract* purses, as the refinement relation could — and does, without this constraint — give different purses having the same *balance* and *lost* fields.

The Z specification also defines "true" records, for instance *TransferDetails* and *PayDetails* which represent respectively abstract and concrete transaction details.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{TransferDetails}} \\
\hline
\textit{from}, \textit{to} : \textit{NAME} \\
\textit{value} : \mathbb{Z} \\
\hline
\textit{value} \geq 0 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PayDetails}} \\
\hline
\textit{TransferDetails} \\
\textit{fromSeqNo}, \textit{toSeqNo} : \mathbb{Z} \\
\hline
\textit{fromSeqNo} \geq 0 \\
\textit{toSeqNo} \geq 0 \\
\hline
\end{array}$$

So, such data types have to be represented in Alloy as records, i.e. with the canonicalization constraint.

```
sig TransferDetails {
  from, to : Purse,
  value : set Coin
}
sig PayDetails extends TransferDetails {
  fromSeqNo, toSeqNo : SEQNO
}
fact payDetailsCanon {
  no disj p, p' : PayDetails {
    p'.from = p.from
    p'.to = p.to
    p'.fromSeqNo = p.fromSeqNo
    p'.toSeqNo = p.toSeqNo
  }
}
```

3.2 Consequence : existential quantification and constraints

The simulation proofs require to show that for any *Between* operation and *Abstract* post-state, there *exists* an *Abstract* pre-state such that the *Abstract* operation holds (as required by the backwards refinement), and similarly for the *Between/Concrete* refinement proof (but forwards).

It is important to understand the notion of existence in the right way. Indeed, in the Z notation, an existential theorem corresponds to the fact that a record with the right field values may be *constructed*. That is, if the theorem is stated in an existential way, the proof will give the witness.

But in *Alloy*, existence is the *actual* existence of the corresponding atomic objects in the model. That explains the following behaviour in the *Abstract/Between* refinement. Let us try to show the following predicate for the *Between Abort* operation, using the method of "encapsulating" the *chosenLost* set into a specific signature as a field of this signature:

```
sig ChosenLost {pd : set PayDetails}
assert ReqEx {
  all b, b' : BetweenWorld, a' : AbWorld, cl' : ChosenLost  {
    {
      Rab (a', b', cl'.pd)
      Req (b, b')
    } implies some a : AbWorld, cl : ChosenLost {
      Rab (a, b, cl.pd)
      AbIgnore (a, a')
    }
  }
}
```

Then, a counterexample would come: the model with only one *ChosenLost* object, preventing some cases where the *ChosenLost* must change from the post-state to the pre-state.

This is also the reason why a sanity-check property has to be verified through simulating a predicate rather than trying to check an existential assertion. Indeed, if we naively tried to show that there exists a *BetweenWorld*, to show that the constraints are not too strong and allow an object to exist:

```
assert BetweenEx {
   some BetweenWorld
}
```

then, the immediate counterexample comes: the empty model, with no atoms at all!

A naive idea would be to constrain the Alloy model to match the Z notion of existence, that is to constrain any constructible object to exist. But that idea is very naive, as an immediate problem arises with the *Alloy Analyzer*: the scope dramatically grows.

That is why the only solution is to construct the witness in the theorem itself, and to *assume* that an object exists once we have *enough properties* to define it. For instance, an *Abstract* world is completely determined if we know its *abAuthPurse*, that is the set of all its authentic purses and their properties. Thus, we can consider that the *Rab* abstraction relation, which computes the values of *balance* and *lost* fields of the authentic purses of an *Abstract* world abstracting the given *Between* world and the *ChosenLost* variable, constructs an object which has the structure of an *Abstract* world.

But assuming the existence of a constrained object does not make sense: thus it is necessary to not define constraints as such, but define them as predicates which will be used as implication hypotheses in assertions. For instance, instead of defining and using the *Abstract* and *Between* worlds as follows:

```
sig BetweenWorld extends ConWorld {}
fact BetweenConstraints {...}

assert RabIgnore {
  all b, b': BetweenWorld, a' : AbWorld,
  cl' : set PayDetails {
    {
      Rab (a', b', cl')
      Ignore (b, b')
    } implies some a : AbWorld {
      Rab (a, b, cl')
      AbIgnore (a, a')
    }
  }
}
```

it is a better idea to define constraints as predicates rather than facts:

```
sig AbWorld {abAuthPurse : NAME -> AbPurse}
pred Abstract (a : AbWorld) {
  a.abAuthPurse : NAME -> lone AbPurse
  ... -- and abstract coin sharing constraints
}
pred Between (b : ConWorld) {...}
```

This also allows to check constraints (including coin sharing constraints defined in the previous section) as invariants.

Then, the abstraction relation could be also defined "structurally", with no references to the "constraints":

```
pred Rab (a : AbWorld, b : BetweenWorld, cl : set PayDetails) {
  a.abAuthPurse.AbPurse = b.conAuthPurse.ConPurse -- 1
  all n : NAME {
    n in b.conAuthPurse.ConPurse implies {
      one n.(a.abAuthPurse) -- 2
      n.(a.abAuthPurse).balance = ...
      n.(a.abAuthPurse).lost = ...
    }
  }
}
```

1. The authentic names are the same for the abstract as for the between world.
2. for any authentic name, there is exactly one corresponding abstract purse.

Then, the assertion could be stated as follows:

```
assert RabIgnore {
  all b, b': BetweenWorld, a, a' : AbWorld,
  cl' : set PayDetails {
    {
      Abstract (a')
      Rab (a', b', cl')
      Ignore (b, b')
      Rab (a, b, cl')
    } implies {
      Abstract (a) -- 1
      AbIgnore (a, a')
    }
  }
}
```

It is worth noting that multiplicity constraints also have to be defined as additional constraints.

Then, the following lemma would avoid conclusion 1 to be checked each time:

```
assert RabEx {
  all b : ConWorld, a : AbWorld, cl : set PayDetails {
    {
      Rab (a, b, cl)
    } implies {
      Abstract (a)
    }
  }
}
```

That is, the abstraction relation (provided the *chosenLost* set of transactions consists in only critically ambiguous transactions that may be lost, a constraint that has to be defined in the abstraction relation) always defines an *Abstract* world starting from a *Between*. Or, in other words, any object that would have the same structure of an *Abstract* world but would abstract a given *Between* world through the abstraction relation, automatically verifies the constraints of an *Abstract* world, thus is itself a "true" abstract world.

4 Results

4.1 Bugs found in the Z specification

The use of the *Alloy Analyzer* gave some counterexamples not related to the way of modeling the *Mondex* specification in *Alloy*. Indeed, some of those counterexamples correspond to real *bugs* in the original Z specification. Those bugs were discovered very early, in analysing the initial specification. However, the optimized specification gave no further bugs.

The *Alloy Analyzer* found two bugs related to reasoning errors in the specification. This points out the fact that the proofs led in the Z specification [SCW00] are not formal enough, as they rely on informal comments that can be only implicitly checked by automated formal methods. Those informal comments can induce a wrong reasoning schema, leading to a proof that is formally valid but useless as it is not the proof of a given theorem: when a theorem is split into lemmas, the link between the theorem and the lemmas is sometimes shown only through informal comments, not through a formal proof. The first bug, about the *Abort* proof schema of the *Abstract/Between* refinement, illustrates the effect of an incorrect informal splitting of a theorem into cases, leading to an incorrect proof of the theorem, whereas the second bug, about the framing schema, points out how a lemma is incorrectly used in the proof of a theorem even though the proof of the lemma itself is correct.

The *Alloy Analyzer* also found a bug in the specification itself: missing constraints about authenticity. This notion is important insofar as it prevents money from magically appearing or disappearing during an operation, either because a purse appears or disappears, or because a purse is making a transaction with a non-authentic purse.

4.1.1 Abort proof schema

Mostly, the *Alloy* method allows to directly check the specification without going through intermediate lemmas. But some theorems consumed too much time of analysis, or even did not terminate if directly checked at once. So, for such theorems, we had to go into the proof details.

Consider the *Abort* operation on a *Between* world. This operation is triggered by a purse when it decides to get rid of a transaction it is involved in, for instance after a timeout when the other purse has been disconnected too early. In the *Abstract/Between* refinement, this operation refnes *AbIgnore*, the *Abstract* no-op. Indeed, the actual transfer only happens once the "to" purse is credited via the *Val* operation. Thus, any other operation is considered abstractly to be no-op.

For this *Abort/AbIgnore* refinement, a check on a scope of 8 did not terminate after 2 days of computation. So, it was necessary to tackle a lemma.

The *Abort* operation can be split into three cases:

1. when the transaction has gone so far that aborting it leads to definitely losing the money;
2. when the transaction has not gone far enough to decide;
3. when there was no transaction to abort (the purse was idle).

Case 3 is easy to separate. Just discriminate on the status of the purse, when the aborting purse has no pending transaction, hence nothing to abort.

To distinguish between cases 1 and 2, the Z proof claims that it is enough to discriminate on whether the transaction in progress is in *maybeLost*, that is critically ambiguous, arguing that in this case, the "to" purse is necessarily aborting.

Actually, this is false, as the *Alloy Analyzer* generates a counterexample where the transaction in progress is in *maybeLost* but the "from" purse is aborting, not the "to". It is worth noting that a transaction becomes lost only when the "to" purse has logged the transaction. For instance, the "from" purse may abort after having sent the money whereas the "to" purse has still neither received the value nor aborted.

The right condition that makes the proof work — and thus, the theorem hold, as expected — is that the aborting purse is the "to" purse waiting to be credited. This is actually one of the two cases when the transaction is in progress. The other case is when the aborting purse is the "from" purse expecting the acknowledgment. The latter case never causes money to be lost.

The false claim has been present only in the informal text of the proof : it has not been formalized why splitting the proof of *Abort* through that condition worked. That is why this bug has not been found by other methods as of May 2006.

4.1.2 Framing schema for operations that first abort

To make the proof easier, and to avoid showing several times that *Abort* refines *AbIgnore*, it is wise to show that operations that first abort (that is, operations initializing a transaction or a log clear) may be decomposed into elementary operations, the first being *Abort*.

The problem is that if such decomposition theorems are tackled with the *Alloy Analyzer*, they generate counterexamples! So there is necessarily a bug in the Z specification.

Actually, whereas some elementary operations output specific messages, *Abort* outputs a generic message called $\perp$. The Z proof argues that operations first aborting are defined through a *framing schema* Φ, that is through a definition of the form :

$\exists \Delta \mathit{ConPurse} \bullet \Phi \wedge (\mathit{Abort};\ \mathit{ElementaryOp})$

where ; is the *composition* operation.

Then, the Z proof argues that this can be decomposed into two parts :

$(\exists \Delta \mathit{ConPurse} \bullet \Phi \wedge \mathit{Abort})\,;$

$(\exists \Delta \mathit{ConPurse} \bullet \Phi \wedge \mathit{ElementaryOp})$

using a lemma assuming that Φ is of the following form :

ConWorld

$\mathit{conAuthPurse} : \mathit{NAME} \nrightarrow \mathit{ConPurse}$

Φ

$\Delta \mathit{ConWorld}$
$\Delta \mathit{ConPurse}$
$n? : \mathit{NAME}$

$n? \in \mathrm{dom}\ \mathit{conAuthPurse}$
$\mathit{conAuthPurse}\ n? = \theta \mathit{ConPurse}$
$\mathit{conAuthPurse}' = \mathit{conAuthPurse} \oplus \{n? \mapsto \theta \mathit{ConPurse}'\}$

But, even though the lemma itself might be true, actually the *process* is wrong because Φ is actually *not* of the specified form! Actually, the lemma neglects the non-functional fields of *ConWorld*, among which is the *ether*. This means that messages are not handled by this schema. This explains the obtained counterexamples, for which *Abort* and the elementary operation output different messages, so that it is impossible to compose them.

One solution is to constrain the generic message $\perp$ to be necessarily in the *Between ether*. In that case, the composition does work, as the *Abort* operation does not add any new message to the *ether*. But the lemma would still have to be adapted, for instance by handling some non-functional fields (such as *ether*) and by showing a modified form of this lemma where the first operation does not modify the non-functional fields but the second may do so.

4.1.3 Authenticity

The original Z specification requires that for any "from" purse expecting a request, its *pdAuth*, that is the current transaction details held by the purse, must be *authentic* : its *from* field must match the "from" purse.

But, even though a general constraint requires the purse to match either the *from* or the *to* field, there is no more precise constraint for the "to" purse expecting the value, or even the "from" purse expecting the acknowledgment.

Due to this lack, trying to check the *Abort/AbIgnore* refinement yields a counterexample. Actually, while trying to check this refinement with the method described above, two counterexamples are (successively) generated in addition to the one related to the *Abort* refinement itself:

- the one if the purse holds a *pdAuth* indicating that it is actually the *from* purse, but expecting to be credited (a state in which only a "to" purse can be);
- the other if the purse holds a *pdAuth* indicating that it is actually the *to* purse, but expecting an acknowledgment (a state in which only a "from" purse can be).

This lack of authenticity creates an inconsistency in the actual role played by the purse in the transaction: their status does not match the indication in their transaction information.

Adding the corresponding contraints in the *Concrete*, or even in the *Between* world, solves this problem and suppresses these counterexamples.

This bug has also been found by other methods like Z/Eves [FW06] or KIV [SGHR06].

4.2 Scopes and times of checks

The choice of the scope for a theorem is a very tough issue. Indeed, the user has to find a balance between the time they want to spend checking an assertion, and the confidence level they require for it.

At least, for each signature, the scope should be as large as the number of quantifications over objects of this signature. Indeed, if the scope is not large enough, then hypotheses may not be able to hold, and the theorem would be trivially true within this scope.

It is often admitted that a scope of 8 is reasonable for most models.

Actually, as regards the *Mondex* case study:

- Given an operation, it is sound to bound the number of abstract or concrete worlds to the number of times they are quantified over in the formula. Indeed, outside the considered operation, states are independent on each other.
- But this reasoning does not apply to purses: whereas it is sound to require at least 2 purses (the "from" and the "to"), they *do* depend on other purses because of their local *exLog*. In particular, even the computation of the corresponding abstract *balance* and *lost* does depend on several purses. Moreover, it is also interesting to consider some non-authentic purses.

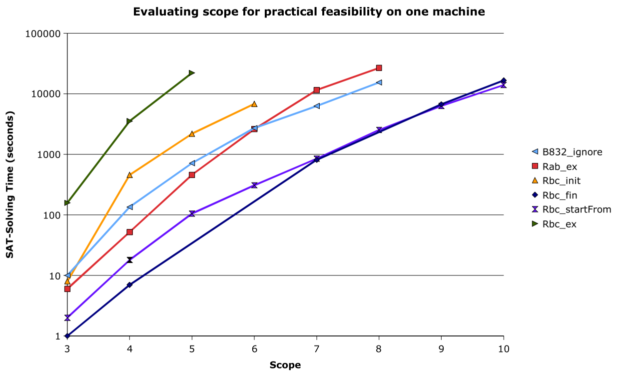

Figure 5: *Time exponentially increases with the scope This graph was obtained with the first model.*

- No bound on transactions or messages may be found either, for a similar reason.

The problem is that the time of checking exponentially increases with the scope.

Besides scope problems, intensive SAT-solving raises technical issues:

- machines have to be powerful enough to be able to tackle the problem, so the times of checks also depend on the speed of the processor and the amount of memory;
- but even on a given machine, the same problem being tackled by different SAT-solvers may take different times, or even crash.

Roughly speaking, SAT-solvings have been tackling from a few seconds to several hours, up to one day, except for the *Abort/Between* refinement which has been stopped after two days of unsuccessful computation.

Whereas the scopes have been successively checked for the first model, the final model has been directly checked for a scope of 10 (modulo restrictions for worlds), except for the *Abstract/Between* refinement and the *Between* model consistency where the scope has been limited to 8, as for the first model. It is worth noting that in that case, the times are sensitively longer for the final model than for the first model. On the one hand, this is due to the constraints, which were too strong in the first model, and have been weakened in the final model. On the other hand, it might be also due to the way the *Alloy Analyzer* constructs the search space. Indeed, in the final model, there are almost no facts: all the "constraints" are defined by predicates then used as hypotheses in implication formulae in assertions. Thus, the *Alloy Analyzer* might have to consider *every possible combination* of the atoms to define relations.

4.3 Limits to the use of the *Alloy Analyzer*

Because *Alloy* is based on first-order, even despite transitive closures, some properties such as the finiteness of the set of purses in an Abstract world, have to be dropped. But as regards the *Mondex* case study, finiteness properties may be shown *indirectly* by showing, for instance, that during an operation, such sets are obtained by union or symmetric difference from pre-state sets which are assumed to be finite. This is true following the definition of the operations.

But what is more annoying is the *finite scope*. Indeed, the checks led with the *Alloy Analyzer* only show that the theorems hold for a certain number of atoms. More generally, as discussed in [FPB+05], in no way can the *Alloy Analyzer* be used to give a rigorous proof of the checked theorems.

A first attempt could be to try to increase scopes by improving ambient conditions (machines, etc.), or even by using the newer version of the *Alloy Analyzer* based on *Kodkod* currently being developed by SDG. But those methods are still bounded, and do not generalize.

We could also try to show a *small model theorem*, a meta-theorem which could in some way "compute a minimal scope", or *threshold*, for signatures. For instance Lee Momtahan's idea [Mom04] would be to show that, starting from a scope, it is possible to compute a threshold for one signature, for which any greater scope than this threshold would be automatically true, other signatures keeping the same scope. But this approach is still not powerful enough because:

- the extended signature may not be quantified over (except skolemizable quantifications);
- only one signature scope may be extended at the same time.

So, it could be wise to get rid of the scope issue and to choose a more direct approach of really *proving* assertions. Then this will require the use of *external* tools, that is other than the *Alloy Analyzer*. It would be also an interesting way to show that the *Alloy* specification language can be tacked with different methods, not only model-finding.

Prioni [AKMR03] translates an *Alloy* specification into the input language of the *Athena* [Ath] proof assistant, which is based on a logic with powerful relational calculus. But the problem is that *Athena*, as a proof assistant, is not automated enough.

It makes sense to consider that the more expressive the logic, the less automated the tool. Then comes up an apparently interesting solution: automated *first-order* theorem provers.

Indeed, if finiteness properties are dropped, then it is interesting to point out the fact that the *Mondex* case study can be entirely written as a first-order theory, and even without transitive closures. Actually, any higher-order quantification can be turned into first-order. For instance, to clear a set of transaction details from the logs, the Z specification computes a code, called *clear code*, to represent the set being cleared. So, it is possible to quantify over this clear code instead of the whole set being cleared. Moreover, as operations are considered individually, transitive closures are not useful : there are no theorems to be shown about sequences of operations.

5 Conclusion and related work

The *Alloy* formal method, based on first-order relational logic with transitive closures, allowed to specify the *Mondex* case study almost entirely, that is just dropping the properties about finiteness, even though those properties may be shown indirectly. Then, without those properties, this work shows that the *Mondex* case study can be rewritten as a first-order theory, even without transitive closures.

Despite some implementation issues that should be improved in its successor version currently under development by the SDG group, the use of the *Alloy Analyzer* allows to rapidly and efficiently develop a specification ; thanks to *model-finding*, sanity checks are made in a straightforward way. The *Alloy Analyzer* also allowed us to find *bugs* in the original Z specification. Those bugs may be relevant to the specification itself as much as to the proof, or even to informal comments guiding the proof. Those bugs have also been found by other methods such as Z/Eves or KIV, so the *Alloy Analyzer* can fairly compete in finding bugs in specifications.

However, beyond finding those bugs, the *Alloy Analyzer* itself does not provide any *proof* of the theorems, as discussed in [FPB+05], only a confidence level depending on the size of the search space. But although this is often considered to be enough in industrial software verification, it would not fit to try to prove security-sensitive specifications such as the *Mondex* case study.

So it is necessary to extend the results obtained with the *Alloy Analyzer*. Lee Momtahan's work upon a small model theorem [Mom04] could be a first step towards generalizing results given by the model-finder. But its too strong constraints over the specification, requiring signatures to not be quantified at all, do not fit the *Mondex* case study. So, other formal methods have to complete the use of the *Alloy Analyzer*. *Prioni* [AKMR03] intends to use *Alloy* specifications with the *Athena* proof assistant, which is not fully automatic. But trying to handle *Alloy* models in first-order logic could be also interesting. We have done some first attempts [Ram06], but only *Abstract* security properties have been shown so far. In fact, to be able to practically use theorem provers, it would be necessary to improve the conception of automated theorem provers, which is the concern raised by competitions such as TPTP [TPT]. For more general cases than *Mondex* which might use transitive closures, Tal Lev Ami's work [LAIR+05] could represent an interesting first-order logic complement to *Alloy*, as it moreover tries to handle transitive closures.

It could be also interesting to develop syntactic analysis of Alloy specifications, or even to automatize relational calculus and reasoning directly at the formula level, which would make the constraint of finite scope irrelevant, as discussed in [FPB+05].

6 Acknowledgments

I would like to acknowledge Daniel Jackson and all the SDG members for their useful help — sometimes overnight — throughout, and even after, this internship, and for letting me discover the spirit of the laboratory, which introduced me to some research topics not tackled in France, through visiting the laboratories and attending talks led by researchers from all over the world.

I would also like to acknowledge all the VSR-net members for their attendance on the meetings, in particular Jim Woodcock and Juan Bicarregui for their hospitality and their particularly enjoyable organization of the VSR-net events in England.

I would also like to acknowledge Patrick Cousot, the students' advisor within the Computer Science Department of the École Normale Supérieure, for connecting Daniel Jackson and me, and more globally for his reliable pieces of advice about research topics.

References

[AA] The Alloy Analyzer. `http://alloy.mit.edu`.

[AKMR03] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating Model-Checking and Theorem Proving for Relational Reasoning. In *7th International Seminar on Relational Methods in Computer Science (RelMiCS)*, 2003.

[Ath] The Athena interactive theorem proving system. `http://www.cag.csail.mit.edu/~kostas/dpls/athena`.

[FPB+05] Marcelo F. Frias, Carlos G. López Pombo, Gabriel A. Baum, Nazareno M. Aguirre, and Thomas S. E. Maibaum. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Methodol.*, 14(4):478–526, 2005.

[FW06] Leo Freitas and Jim Woodcock. Mondex in Z/Eves. Slides for the 3rd VSR-net workshop, 5 2006.

[Hal90] Anthony Hall. Using Z as a Specification Calculus for Object-oriented Systems. In *VDM90 : VDM and Z Formal Methods in Software Development, Lecture Notes in Computer Science*, number 428, pages 290–318, 1990.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *Proceedings of ACM SIGSOFT Conferences on Foundations of Software Engineering*, 11 2000.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis.* The MIT Press, 2006.

[LAIR+05] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Shmuel Sagiv, S Srivastava, and Greta Yorsh. Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In *Proceedings of 20th International Conference on Automated Deduction*, pages 99–115, 2005.

[MCS] The Mondex Case Study. `http://qpq.csl.sri.com/vsr/private/repository/MondexCaseStudy`.

[Mom04] Lee Momtahan. Towards a Small Model Theorem for Data Independent Systems. *Electronic Notes in Theoretical Computer Science*, 128(6), 3 2004.

[Mon] The Mondex electronic purse system. `http://www.mondex.com`.

[Ram] Tahina Ramananandro. The Mondex Case Study with Alloy. `http://www.eleves.ens.fr/~ramanana/work/mondex`.

[Ram06] Tahina Ramananandro. Mondex, an electronic purse : specification and refinement checks with the Alloy model-finding method. Internship report, MIT and École normale supérieure, 2006.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. *An electronic purse: Specification, Refinement and Proof.* Technical Monograph PRG–126. Oxford University Computing Laboratory, Programming Research Group, 2000.

[SGHR06] Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, and Wolfgang Reif. The mondex challenge: Machine-checked proofs for an electronic purse. Technical report, Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg, 2 2006.

[Spi92] J. Michael Spivey. *The Z notation: a Reference Manual.* Prentice Hall, 2nd edition edition, 1992.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[Tor] Emina Torlak. Kodkod, model finder for first order relational logic. `http://web.mit.edu/emina/www/kodkod.html`.

[TPT] Thousands of Problems for Theorem Provers. `http://www.cs.miami.edu/~tptp`.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.