

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

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

September 1, 2006

Contents

1	Introduction	6
1.1	The VSR Project	6
1.2	The <i>Mondex</i> case study	7
1.2.1	The abstract model	8
1.2.2	The concrete model	9
1.2.3	The <i>Between</i> world : operations and constraints	16
1.2.4	The refinement process	17
1.3	Overview of the Z specification language through a simplified version of <i>Mondex</i>	20
2	A simplified version of the <i>Mondex</i> specification in Alloy	26
2.1	Modeling with the Alloy specification language	26
2.1.1	Defining signatures and relations	26
2.1.2	Defining logical constraints : <i>facts</i> . The relational calculus	27
2.1.3	Signature extension and inclusion	29
2.1.4	Auxiliary predicates and functions	30
2.1.5	Assertions	32
2.1.6	Modules	34
2.2	Analyzing the specification using the <i>Alloy Analyzer</i> , a model finder	35
2.2.1	Type checking	36
2.2.2	Checking assertions. Notion of finite scope.	36
2.2.3	“Running” predicates : sanity-check simulation	38

3	Modeling the <i>MondeX</i> case study in <i>Alloy</i> : technical issues encountered	41
3.1	Finiteness	41
3.2	Integers	42
3.2.1	Sequence numbers	43
3.2.2	Amounts	44
3.3	Clear codes	48
3.4	Existential quantification and constraints	49
3.5	The identity of objects	51
4	Summary of the final model layout	55
4.1	The Common module	55
4.2	The Abstract module	55
4.3	The Concrete Purse module	57
4.4	The Concrete World module	58
4.5	The Between World module	58
4.6	The Between World operations module and the Concrete World operations module	59
4.7	The Between and Concrete initialization and finalization module	59
4.8	The Between operation consistency module	59
4.9	The Abstract/Between refinement module	59
4.10	The Between/Concrete refinement module	60
4.11	The Canonicalization module	62
5	Results	62
5.1	Bugs found in the Z specification	62
5.1.1	Abort proof schema	62
5.1.2	Authenticity	63
5.1.3	Framing schema for operations that first abort	63
5.2	Scopes and times of checks	64
5.3	Limits to the use of the <i>Alloy Analyzer</i>	66
6	Using first-order theorem provers with Alloy	69
6.1	The usual approach : Alloy atoms as FOL atoms	69
6.1.1	Principle	69
6.1.2	Simplifications	70
6.2	The “lifted” way : Alloy relations as FOL atoms	71
6.3	Results and limits	72
7	Conclusion and future work	73

References	75
List of Figures	78

Résumé

Du 6 mars au 26 août 2006, j'ai suivi un stage de recherche au MIT, précisément au CSAIL¹, au sein du *Software Design group* dirigé par Daniel Jackson. Ce groupe est axé sur la conception d'un langage de spécification, *Alloy* [Jac02, Jac06], fondé sur la logique du premier ordre et le calcul des relations avec clôture transitive, et le développement d'un logiciel, *Alloy Analyzer*[AA], pour analyser des spécifications dans ce langage en lui cherchant des modèles au moyen de la traduction de la spécification en une formule booléenne à satisfaire (technique du *model-finding* par *SAT-solving* [Jac00]) .

Le but de ce stage était de montrer les capacités du langage de spécification Alloy et du programme *Alloy Analyzer* dans le cadre de la vérification automatique de spécifications, en considérant la preuve de la spécification (initialement dans le langage de spécification Z [Spi92, WD96]) du système de porte-monnaies électroniques *Mondex* [Mon, SCW00, MCS] comme cobaye. Après avoir découvert la méthode formelle *Alloy*, j'ai modélisé le système *Mondex* en utilisant le langage de spécification *Alloy* et je l'ai vérifié pour des univers de taille réduite. J'ai alors trouvé quelques bogues et j'en ai fait état en Angleterre, lors d'un atelier le 26 mai au *Cosener's House*, dépendant du *Rutherford Appleton Laboratory* (Angleterre). À mon retour le 6 juin, j'ai amélioré cette première version pour aboutir à une spécification en *Alloy* plus complète et plus rigoureuse du système *Mondex*, avant d'essayer de généraliser les résultats obtenus à un univers de taille quelconque en utilisant des méthodes externes comme la preuve automatique de théorèmes du premier ordre.

Je souhaite remercier tous les membres du SDG pour leur aide précieuse — parfois en pleine nuit — tout au long de ce stage, et pour m'avoir initié à l'esprit du laboratoire, ce qui m'a permis de découvrir un aperçu de quelques sujets de recherche en informatique inconnus en France, en visitant les laboratoires et en assistant à des séminaires donnés par des chercheurs du monde entier.

¹Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge (Massachusetts) 02139, United States of America

Abstract

I have been attending a research internship since March 6th, until August 26th, 2006, at MIT, precisely CSAIL¹, within the Software Design group led by Daniel Jackson. This research group aims at developing a specification language, called *Alloy* [Jac02, Jac06] and based on first-order logic and relational calculus including transitive closures, and a tool, called *Alloy Analyzer* [AA] and based on model-finding through SAT-solving [Jac00], to analyze specifications in this language.

The aim of this internship was to show the capabilities of the Alloy specification language and the Alloy Analyzer in automated specification checking, by tackling the proof of the specification (initially in the Z specification language [Spi92, WD96]) of the *Mondex* electronic purse system [Mon, SCW00, MCS] as a case study. After discovering the Alloy formal method, I modeled the *Mondex* system using the Alloy specification language and checked it for small scopes with the Alloy Analyzer, finding some bugs I showed in England at a workshop on May 26th at Cosener's House, a dependency of *Rutherford Appleton Laboratory* (England). When I came back on June 6th, I improved that initial version to produce a more complete, rigorous Alloy specification of the *Mondex* system, before trying to generalize the results to any scope through external methods such as automated first-order theorem proving.

I would like to acknowledge all the SDG members for their useful help — sometimes overnight — throughout this internship and for the spirit of the laboratory which allowed me to discover an overview of some Computer Science research topics which are not tackled in France through visiting the laboratories and attending talks led by researchers from all over the world.

1 Introduction

1.1 The VSR Project

The UK Computing Research Committee has launched several projects called *Grand Challenges in Computing* [Com]. Such projects meet several strong criteria like historical interest, internationality, and general purpose, not to mention their actual, scientifically challenging level. Conferences are organized every two years to discuss the global progress of the projects and the opportunities to add new projects as *Grand Challenges*.

The sixth *Grand Challenge* (GC6), called *Dependable software evolution* [GC6], led by Jim Woodcock², with the cooperation of Tony Hoare among others, addresses automated systems, thus involving works on software robustness and verification.

One of the current projects tackled within GC6 is the *Verified Software Repository* [JOW06, VSR], which is a network (also called VSR-net) gathering different workgroups with their own verification methods. The VSR-net aims at finding fully automatic formal methods for software verification. To achieve this, different formal methods are to tackle the same case study to point out their features and their limits, and eventually to improve them. The main case study chosen so far is the *Mondex* electronic purse (also called *smart card*) system [Mon, MCS], a specification which has been proven by hand in Z [Spi92, WD96, SCW00].

The VSR-net has led several meetings, the last two of them at Rutherford Appleton Laboratory (England) on January 13-14th and on May 26-27th, 2006. The next VSR-net meeting will occur at the University of York, England, on October 5-6th, 2006. The following candidates have been taking part to the project so far :

- Z, the original language for the *Mondex* hand proof. Automatization attempt with the help of the *Z/Eves* theorem prover [ZE, FW06]
- The B method [Abr96a] : Event-B [Abr96b, BY06]
- KIV [KIV, SGHR06]
- OCL [WK99, Gog06]
- *Perfect Developer* [CC04, PD, Cro06]
- Raise [GHH⁺95, GH06]
- VDM [Jon06, Jon90]
- Alloy [Jac02, Jac06], a specification language, with the help of the *Alloy Analyzer* model-finder [AA, Jac00]

There are for sure many other methods for software verifying. Among them, proof assistants like *Coq* [Coq] are very well known. But, despite their extended logical skills (higher order logic, calculus of constructions), they are not automated enough to efficiently tackle part to the VSR-net project.

²University of York, United Kingdom

1.2 The *Mondex* case study

In 1994, National Westminster Bank developed an electronic purse (or *smart card*) system, called *Mondex* [Mon]³. An electronic purse is a card-sized device intended to replace “real” coins with electronic cash. Contrary 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. *Mondex* has spread over Great Britain and other places around the world such as Hong Kong or New York. Other systems have been developed since, such as *Moneo* [Mno] in France.

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
- no money may suddenly appear on a purse without being debited from another purse through an achieved transaction

This research, funded by DataCard (once National Westminster Development Team, then Platform 7), led to the publication, in 2000, of a hand proof of the *Mondex* electronic purse system with the Z notation [SCW00], by Susan Stepney, David Cooper and Jim Woodcock. Thanks to this proof, the *Mondex* system has been 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 simplified 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 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

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

³Since then, this system has been sold to *MasterCard International*.

lost, then it has necessary 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
- the refinement of the *Between* by the *Concrete* is rather an invariant proof than a refinement proof.

1.2.1 The abstract model

The abstract model consists in a world of abstract *authentic* purses. Authenticity is an intrinsic property of the state : it is defined by the choice of the subset of authentic purses.

An abstract purse has two state components storing values :

- *balance* stores the amount of money available to the purse for foregoing transactions
- *lost* stores the amount of money lost through failed transactions initiated by the purse

Three operations between abstract states are possible :

- *AbIgnore* does nothing
- *AbTransferOkay* is a successful transfer between two purses : the “from” purse decreases its *balance* at the same time as the “to” increases its own one
- *AbTransferLost* is a lost transfer between two purses : the “from” purse decreases its *balance* and increases its *lost* store at the same time. This operation is provided to abstract lost transactions in the *Concrete* world, as no abortion is provided in the *Abstract*.

Those operations are *atomic* : they modify the stores of both involved purses at the same time.

The specification considers the *AbTransfer* operation as the main transfer operation, that is any *AbTransferOkay* and any *AbTransferLost* operation, to make the nondeterminism of the protocol explicit.

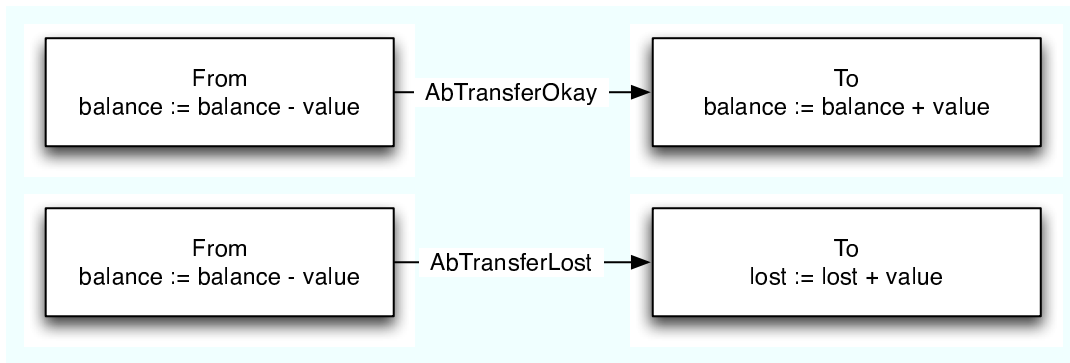


Figure 1: *Abstract (atomic) transactions : successful, lost.*

1.2.2 The concrete model

The concrete model consists not only in a world of concrete authentic purses, but also a communication channel, called *ether*, and an *archive* which is a global logging system for failed transactions.

1.2.2.1 The *ether* communication channel The *ether* communication channel is simply a set of messages. A purse sends a message by simply putting it into the *ether*. It receives a message simply by reading it from the *ether*. But the *ether* is insecure :

- a received message can stay in the *ether*, so it can be replayed. Thus, operations have to ensure that a purse reads a message at most once.
- a message can be lost, disappearing from the *ether* before being read
- a message can be read by any purse. Thus, the specification has to ensure that operations are not triggered when a purse reads a message relevant to other purses.
- a message irrelevant to the protocol can be forged. However, the specification assumes that such a message does not lead to undesired behaviours of the purses. Moreover, messages relevant to the protocol are assumed to be “protected” and unforgeable (for instance, they may be encrypted, but those issues are not tackled by the specification).

1.2.2.2 The concrete transaction protocol Contrary to the abstract model, a concrete transaction is not atomic at all : it must follow 5 steps. The names of the operations refer to the names of the messages received by the processing purses.

1. *StartFrom* : the “from” purse receives a *startFrom* message giving the identity of the “to” purse and the amount being transferred.
2. *StartTo* : conversely, the “to” purse receives a *startTo* message giving the identity of the “from” purse and the amount being transferred. It sends a *req* (for “money **r**equest”) message to the “from” purse.
3. *Req* : the “from” purse receives the *req* message. It decreases its balance and sends the *val* (for “**v**alue”) message to the “to” purse.

4. *Val* : the “to” purse receives the *val* message. It increases its balance and sends the *ack* (for “**acknowledgment**”) message to the “from” purse. The “to” purse is done.
5. *Ack* : the “from” purse receives the *ack* message. The “from” purse is done.

Neither the *startFrom* nor the *startTo* message are sent by the purses themselves, but by a global authority which is not modelled here. In real world, this could correspond to pressing a button to initialize the transaction. Thus, in the model, those messages spontaneously appear in the *ether*.

Once the purses are connected to the system, they are assumed to behave properly and follow the protocol.

1.2.2.3 Properties of the concrete purse

- A concrete purse has only one state component storing value, *balance*. Indeed, contrary to the abstract model, where lost value is logged regardless of the transaction involving it, lost transactions are logged individually in the concrete.
- But the purse does not directly log lost transactions into the public *archive* : first it has to log them into a private log called *exLog* (for “**exception log**”); moreover, it does not need to transfer it to the global *archive*. The *exLog* store may log several aborted transactions.
- A purse can only take part in at most one transaction at a given time : it stores it into its *pdAuth* (for “**payment details authenticated**”) property.
- To prevent purses from being misled by the messages they read, a system of sequence numbers, *SEQNO*, helps uniquely identify the transaction. Each purse has its own sequence number, *nextSeqNo*, increasing at each transaction, so that a transaction is uniquely defined by the identity of the “from” and “to” purse and their initial sequence numbers. This information is also provided in the *startFrom* and *startTo* messages.
- A purse also stores its *status*, among one of the following values :
 - the “from” purse, at the beginning of the *StartFrom* operation, is in *eaTo* (for “**expecting any to**”).
 - the “to” purse, at the beginning of the *StartTo* operation, is in *eaFrom* (for “**expecting any from**”).
 - the “from” purse, at the end of the *StartFrom* operation, is in *epr* (“**expecting request**”).
 - the “to” purse, at the end of the *StartTo* operation, is in *epv* (“**expecting value**”).
 - the “from” purse, at the end of the *Req* operation, is in *epa* (“**expecting acknowledgment**”).

The Z specification introduces an asymmetry in the protocol, through the two statuses *eaTo* and *eaFrom*. But it enforces purses to necessarily alternate roles between different transactions. Moreover, many peripheral operations turn purses into *eaFrom* rather than *eaTo* as an idle status. Thus, many people analysing the specification decided to merge those two statuses. So did we.

Finally, the possible statuses are, in function of the role of the purse in the transaction :

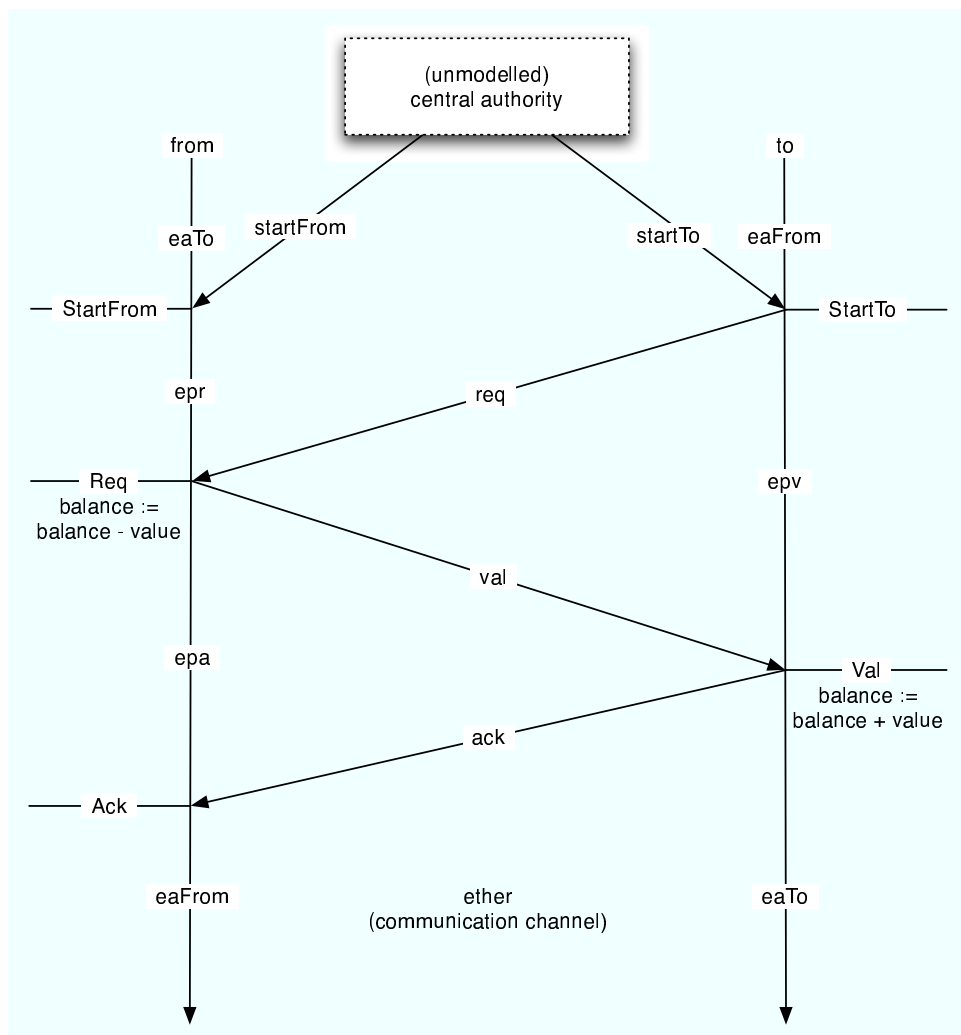


Figure 2: Concrete 5-step protocol, with the statuses of the purses depending on the operations.

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.

- Either purse may be in *eaFrom* or *eaTo* status, when it is idle.
- The “from” purse may be successively in *epr* before sending money, then in *epa* after sending money.
- The “to” purse may be only in *epv* during a transaction.

1.2.2.4 The *Abort* operation : when a transaction may be lost Sometimes, the transaction may lead to a stalemate, for instance when the communications channel loses a message, or when a purse is disconnected from the system too early. In such cases, the purse has to abort to avoid being locked in this transaction. (In real life, this abortion may occur after a timeout, but this issue is not modelled.) Thanks to abortion, the purse is then available to other purses for other transactions. Indeed, a purse cannot be involved in two transactions at the same time.

But as the ether is lossy, a purse cannot know the exact progress of the transaction. In particular, it does not know the status of the counterparty (the other purse involved in the transaction). That is why the *Abort* operation is local to a purse. When a purse aborts, the counterparty might not have aborted yet ; moreover, it is impossible for a purse to know whether the counterparty has aborted the transaction. It is, however, impossible to “abort the *Abort*” operation. Thus, a purse having aborted may no longer receive messages relevant to that transaction.

When a transaction is aborted, money may be lost, if the transaction is aborted after the “from” purse sent the money but before the “to” purse receives it. That is why, in case of abortion, a purse has to *preventively* log the transaction if there is a risk of losing money.

Suppose the “to” purse wants to abort the transaction in progress. It is, then, in *epv* status : expecting the “from” purse to send the money (*val* message). But, there is no way to know whether the money has actually been sent, before receiving the *val* message. If the “to” purse aborts the transaction before receiving the message, then it will never receive the message. Though, the “from” purse might have sent the money. That is why the “to” purse in *epv* has to log the transaction in its *exLog* when it aborts.

Now suppose the “from” purse wants to abort the transaction in progress. It is, then, either in *epr* or in *epa* status, that is expecting the “to” purse to send respectively either the money request or the acknowledgement.

If the “from” purse is in *epr*, then it has not received yet the request, so it has not sent any money yet. In this case, no money may be lost, so there is no need for the purse to log the transaction.

If the “from” purse is in *epa*, then it has already sent the money (as it expects the “to” purse to confirm having received it). If it aborts the transaction, it will never receive the acknowledgement, so it will never know whether the money sent has actually arrived. That is why the “from” purse in *epa* has to log the transaction in its *exLog* when it aborts.

Even though the *Abort* operation is local to a purse, it may be possible to globally (that is, for an external observer who would take all the purses into account — but not the ether, which is lossy) determine whether money is lost. But there are still cases where it is too early to know anything. More precisely :

- The money is definitely lost :
 - if *both* purses abort and log the transaction

- or if the “to” purse logs the transaction while the “from” is in *epa* (having already sent the money)
- The money may be lost, that is the situation is critically ambiguous and cannot be recovered to the previous state, when the “to” purse has neither received the money nor logged the transaction yet, but the “from” has just sent the money and :
 - either is expecting the acknowledgment
 - or has logged the transaction
- On the contrary, no money is lost yet before the “from” purse has sent the money, even though the “to” purse could have logged the transaction (in the latter case, the transaction is ambiguous, but not critically ambiguous, as the previous state can be recovered if the “from” purse aborts before sending the money).
- The money is definitely received when the “to” purse receives the *val* message and sends the *ack*, even though the “from” purse could have aborted and logged the transaction before receiving the *ack* message

1.2.2.5 The global *archive* and the archiving protocol When a purse aborts, the failed transaction, in a relevant case, is logged into an internal *exLog* local to the purse. But the purse may clear its *exLog*, provided it has copied all its information to the global *archive* to prevent any loss of logged data.

A purse may always copy some information from its *exLog* to the global *archive*. This operation, called *Archive*, is totally non-deterministic, as the amount of copied information is unknown a priori (this step could have to repeat several times until all the contents of the *exLog* is copied) ; moreover, this operation may happen at any time, even during a transaction, which does not have to be aborted, since nothing is modified locally to the purse.

Regardless of this copying process, a three-step process may happen for a purse to clear its *exLog*. These steps are constrained so that no transaction logged in the *exLog* may be lost. That is, when an *exLog* is about to be cleared, its contents has to be already present in the global *archive*.

1. A purse receives a *readExceptionLog* message (generated by an unmodelled “global authority”, so spontaneously appearing in the *ether*) : it is required to send an *exceptionLogResult* message carrying one piece of information related to a logged transaction from its *exLog* store. This operation, called *ReadExceptionLog*, models the fact that the “global authority” does not read the *exLog* of a purse as a whole, but only one logged transaction after the other. To perform this operation, the purse has to first abort, so as to eventually send the information about the current transaction if it is logged.
2. The central authority sends a *exceptionLogClear* message carrying a clear code along with the name of a purse which could clear its *exLog*. The clear code is unique to the set of transactions in the *exLog*, through a globally defined and fixed injection (in practice, it could work with something like a hash function), and that corresponding set of transactions is constrained to be included in the global *archive*.

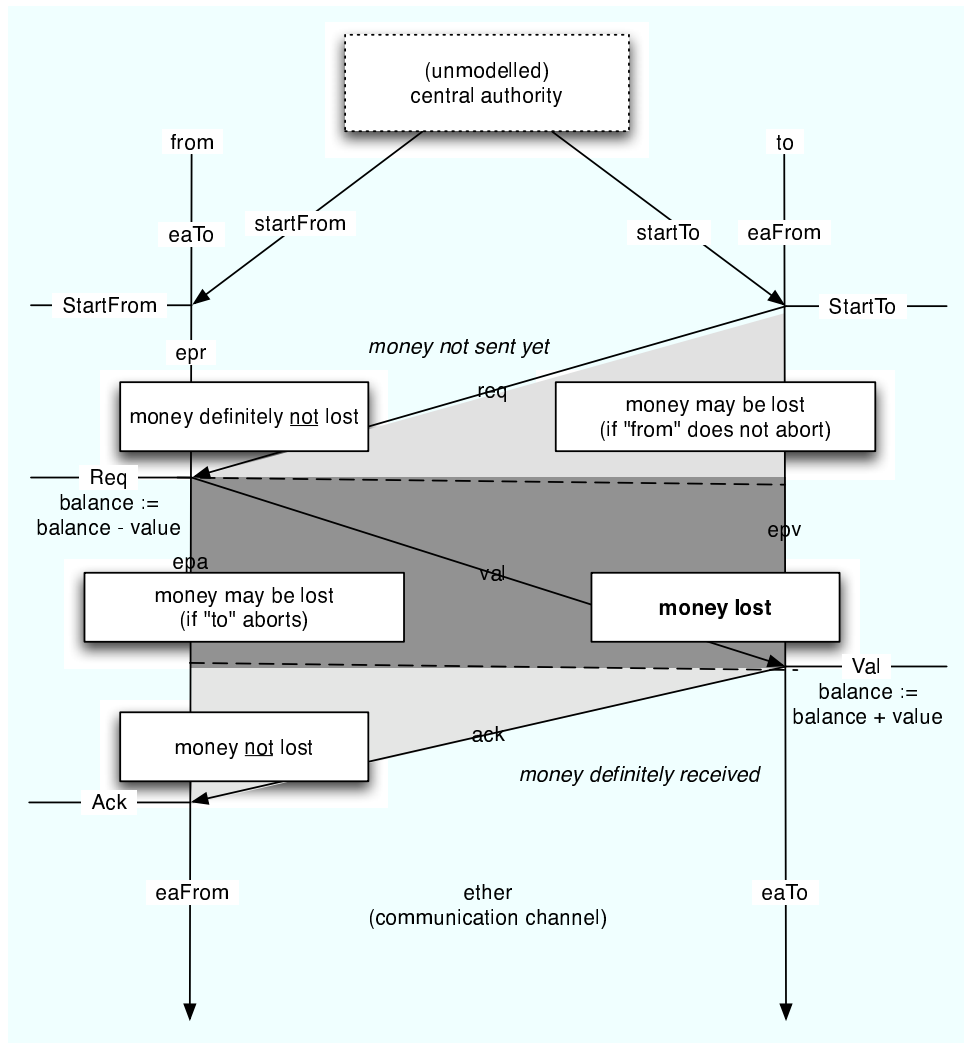


Figure 3: Abort operation : cases when money is lost or not.

This diagram shows what happens when a purse first aborts.

The zone between the “req” sending and the “ack” receival, represent the cases when an aborting purse has to log the transaction into its exLog.

The most critical zone, between the “req” receival and the “val” receival, represent the cases when the money has been sent but not received yet : if the “to” purse aborts in this zone, then money is definitely lost.

The dashed lines denote events important for a purse but happening at the other, so that it is impossible to know that it has happened until the subsequent message is received.

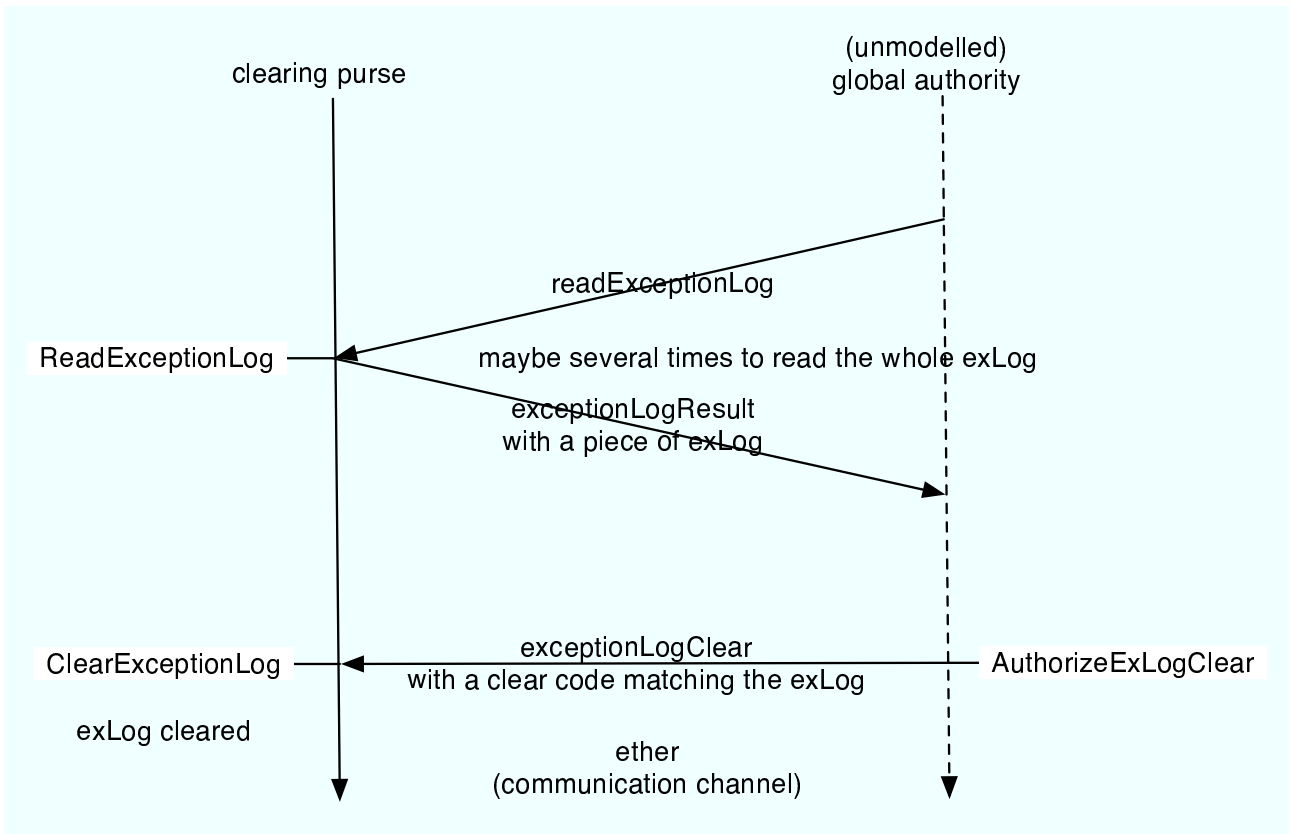


Figure 4: *Clearing process*

As the global authority is not modelled, the protocol only relies on the constraints enforced by the operations for a purse to receive a `exceptionLogClear` message and treat it.

3. The purse receives a `exceptionLogClear` message carrying its name along with a clear code. For the purse to abort, the clear code must correspond to the whole contents of its `exLog`. Moreover, the purse has to first abort, which could allow the current transaction to be logged into the `exLog` before it becomes cleared. If those constraints are met, then the purse may clear its `exLog`.

1.2.2.6 Summary of *Concrete* operations

There are four categories of *Concrete* operations.

- The operations related to the transaction protocol
 1. *StartFrom*
 2. *StartTo*
 3. *Req*
 4. *Val*
 5. *Ack*
- The *Abort* operation

- The operations related to the archive protocol
 1. Archive
 2. ReadExceptionLog
 3. AuthorizeExLogClear
 4. ClearExceptionLog
- In addition to those protocols, two technical operations are also defined in the *Concrete* model :
 - *Ignore* does nothing
 - *Increase* increases the sequence number of a purse without doing anything else. This does not affect the current transaction, which keeps the original sequence numbers of the two purses, numbers that should be less than the current ones.

1.2.3 The *Between* world : operations and constraints

The *Between* world, required for the refinement process, has the same structure as the *Concrete* one, but, contrary to the latter it is constrained. It is meant to complete the refinement between the *Concrete* and the *Abstract* models. Indeed, as the *Concrete* is unconstrained, a direct *Abstract/Concrete* refinement would have had too few hypotheses. Then, *Between* constraints add useful hypotheses to make such a refinement possible. Thus, the *Abstract* is refined by the *Between* instead of the *Concrete*, the latter refining the *Between* as an invariant proof rather than a refinement proof.

The main difference between the *Concrete* and the *Between* is the *ether* : the *Between* *ether* is *not* lossy. That is, the operations in both *Between* and *Concrete* are defined following the aforementioned requirements, but whereas the *Concrete* operations may forget sending messages to the *ether*, the *Between* operations are built with a full *ether* : whenever a purse sends a message, it necessarily appears in the *ether* at the end of a *Between* operation.

The *Between* constraints are mainly relevant to the *ether*.

- No future messages may appear in the *ether*. That is, no *req* or *val* or *ack* messages may appear in the *ether* before the corresponding purses actually send them through the operations. This constraint is expressed through verifying sequence numbers.
- In the same way, no future transactions may have been already logged by the purses.
- Messages may appear in the *ether* only according to the statuses of the purses. For instance, if the “to” purse is still in *epv*, the *ack* message corresponding to this transaction may not have already appeared in the *ether*.
- The clear codes carried by the *exceptionLogClear* messages of the *ether* refer to transactions actually logged in the global *archive*
- The set of all the transaction details logged by the relevant “to” purses (that is, either in their *exLog* or in the global *archive*) has to be finite. This constraint is needed for the *Abstract/Between* refinement, see below.

Other constraints are proper to the *Between*, especially because its *ether* is reliable :

- Whenever a transaction is logged, it must correspond to an existing *req* message in the *ether*.
- A *req* message in the *ether* only refers to an authentic “to” purse. This constraint along with the previous one avoids spurious non-authentic messages or logs.

1.2.4 The refinement process

For easier proofs, a two-step refinement process is used. Indeed, it is easier to tackle a refinement between the *Abstract* and the *Between* constrained version of the *Concrete* model first, than to try to directly refine *Abstract* by the unconstrained *Concrete*.

The general method is to define an *abstraction relation*, or *refinement relation* (depending on the way : “the abstract model abstracts the concrete model”, but “the concrete model refines the abstract model”). Indeed, defining an *abstraction function* is not always possible. But instead, a function depending on other variables in addition to the concrete state can be defined in such cases like the *Mondex* case study.

To put it in a nutshell, a *Between* model is created to both refine the *Abstract* model and abstracts the *Concrete* model.

In this document, we present the proof layout adopted by the Z specification, which we also decided to follow for our *Mondex* specification in *Alloy*.

1.2.4.1 Abstract/Between The proof schema is based on a *backwards* simulation : given a *Between* operation and the corresponding *Abstract* post-state, find an *Abstract* pre-state such that the corresponding *Abstract* operation holds.

In this refinement, there is no abstraction function, but an abstraction *relation*. Indeed, let’s suppose we would define such a function. Given a *Between* world, this function would have to compute, for each authentic *Between* purse, the *balance* and *lost* stores of its corresponding *Abstract* purse (it is obvious that the set of *Between* purses should be bijected with the corresponding set of *Abstract* purses). But an unambiguous computation is impossible in general, for instance when purses abort and log too early (see above).

But if one allows to *choose* which critically ambiguous states (money sent but not received yet) are considered lost, then there is a unique possible computation. For any purse :

- the abstract *balance* equals the concrete *balance* plus the sum of the amounts of the critically ambiguous transactions which are not chosen to be lost.
- the abstract *lost* equals the sum of the amounts of the transactions definitely lost (when it is possible to globally determine such a state after one of the two purses has aborted, see above) and the critically ambiguous transactions which are chosen to be lost.

The amount of an unambiguous transaction is either accounted in the abstract *lost* of the “to” purse or in its abstract *balance* depending on whether the transaction is lost or not.

As the set of transactions chosen lost is used in a backwards simulation, and does not affect the *Between* states themselves (it is only relevant to the abstraction relation), it is a *prophecy variable* : one has to “guess” its pre-value from its “prophecized” post-value and the *Between* states.

The refinement process is as follows : all the *Between* operations refine *AbIgnore*, except *Req* which refines *AbTransfer* (that is *AbTransferOkay* or *AbTransferLost*). That is, thanks to the choice of lost transactions, the atomic transaction may be seen in *Req*. Indeed :

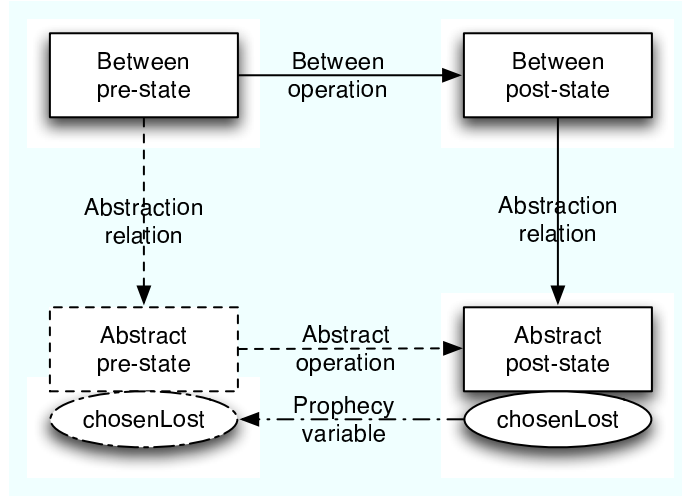


Figure 5: *Backwards refinement proof for Abstract/Between*

The dashed elements have to be proven to exist and to meet the corresponding constraints, eventually with the help of constructing auxiliary elements (dash-dotted). One can see that the prophecy variable construction goes in the way against the operations, but indicates the way of the proof.

- *Req* is the operation during which the “from” purse decreases its balance. As the concrete *balance* is greater than the abstract, *AbTransfer* cannot be refined by any of the previous operations (*StartFrom* or *StartTo*), which act like informational operations “initializing” the transaction, and thus refine *AbIgnore*. It is easier to choose lost transactions as early as possible. As *Req* is the earliest significant transaction operation, it is thus chosen to refine *AbTransfer*.
- *Abort* refines *AbIgnore* because the operation being aborted is either already definitely lost (for instance when the “from” purse aborts whereas the “to” has already aborted), or has been explicitly chosen to be lost. In the latter case, the *Abstract* pre-state is deduced from the post-states by applying the abstraction function with that transaction appended to the post-choice of lost transactions.
- *Val* refines *AbIgnore* : the balance of the “to” purse being increased means that the transaction succeeds. Thus it is not chosen lost, even for the pre-state. The abstract *balance* does not change, as the amount of the transaction moves from the ambiguous (but not chosen lost) state to the concrete *balance*.
- *Ack* refines *AbIgnore*. Indeed, this operation may even not happen if for instance the “from” purse aborts before receiving the *ack* message.
- Archive protocol operations all refine *AbIgnore* because there is no notion of global *Archive* in the abstract model.

Some of those operations (*StartFrom*, *StartTo*, *ReadExceptionLog*, *ClearExceptionLog*) require that the purse eventually first abort. But, as *Abort* refines *AbIgnore*, it does not matter.

Remark. To verify security properties requires the computation of the sum of the abstract *balance* and *lost* stores of all the abstract purses. Thus, for those sums to be well-defined, it is necessary that there be :

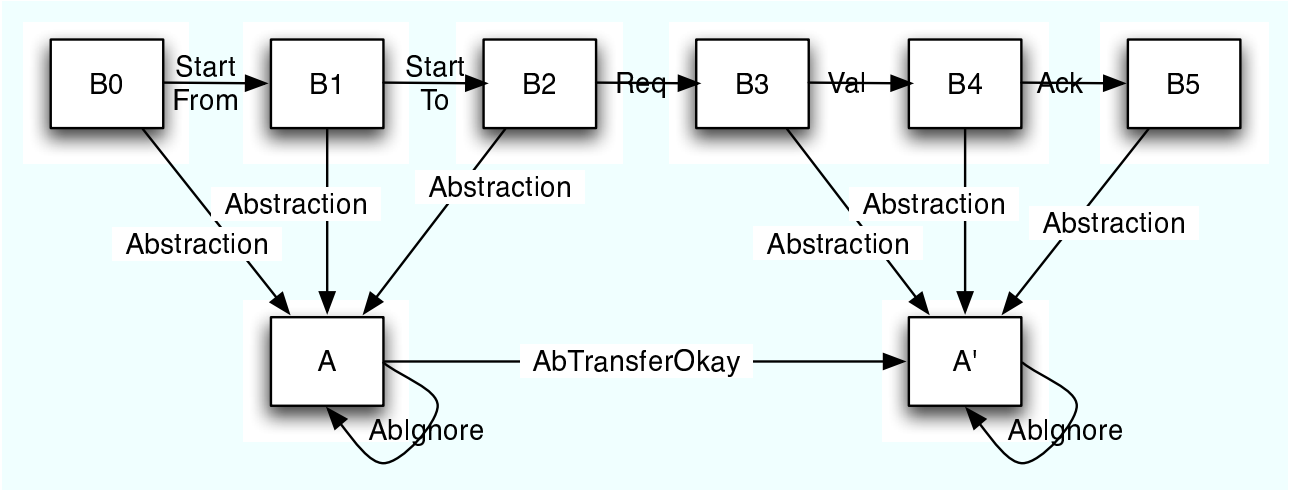


Figure 6: *Abstraction of the typical sequence of Between operations defining a transaction*

- a finite number of abstract purses, to compute the sum of *balance* stores, hence a finite number of between purses
- a finite number of transactions logged by “to” purses, to compute the sum of *lost* stores. The finiteness of transactions logged by “from” purses is not required, as such transactions, if not logged by the corresponding “to” purses, are either definitely not lost, or ambiguous. In the former case, they are *done*, so their values are accounted in Between *balance* stores ; in the latter case, they are *pending*, so they are still occurring between authentic purses, thus their number is finite.

1.2.4.2 Between/Concrete This refinement is rather an invariant check than an abstraction check. Thus, it may be led by a simple *forwards* simulation : given a *Concrete* operation and the corresponding *Between* pre-state, find a *Between* post-state such that the corresponding *Between* operation holds.

As the only structural difference between *Between* and *Concrete* states is the lossiness of the *Concrete* ether, the abstraction relation is only relevant to the *ethers* : neither the properties of the purses nor the global *archive* are changed. That is also the only difference between the *Between* and *Concrete* operations.

The abstraction relation is quite simple : a between state B abstracts a concrete C if and only if, in addition to the respective between and concrete constraints, C’s *ether* is included in B’s (), that is, C’s ether is the result of an eventual loss of messages from B’s ether.

Obviously, that abstraction relation cannot be defined as a function. But given a *Concrete* operation and a *Between* pre-state, it is possible to define a function returning a *Between* post-state abstracting the *Concrete* post-one. Of course it depends on the *Concrete* post-state, actually keeping its purse properties and its global *archive*, but it also depends on the operation, in particular its output message, and the *ether* of the *Between* pre-state, as that output message is appended to this pre-state *ether*.

1.2.4.3 Technical issues To ensure operations are *total*, they are explicitly given the choice of doing nothing, that is they are disjoined with *Ignore*. For instance, a *Val* operation may happen even if the transaction has aborted before, in that case doing nothing. The actual operation would be denoted *ValOkay*.

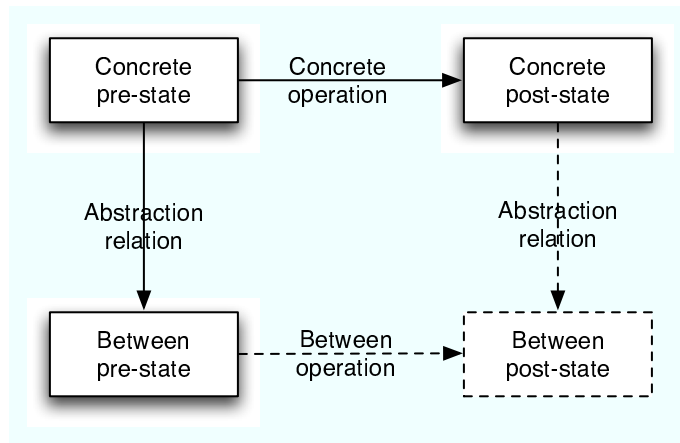


Figure 7: *Forwards refinement for Between/Concrete*

The specification also provides *initialization* and *finalization* steps. For each model, it defines an *initial* and a *final* states which are meant to be initial and final *observational* states : they bound an observation of a sequence of operations.

- An *initial state* is a special state with eventually further constraints. The refinements demand the initial states to match between models through the corresponding abstraction relations.
 - Any abstract state can be initial
 - A *Between* initial state must contain all *readExceptionLog*, *startFrom* and *startTo* messages in its *ether* (recall that messages in the *ether* are not necessarily read by the purses), and
 - Any concrete state can be initial if its correspond to a *Between* initial state having lost messages from its *ether*. Thus a concrete initial state has to meet the constraints of a *Between* state except those relevant to the *ether*.
- A *final state* is any state that is retrieved to an object, called *global world*, which has the same structure as an Abstract state. The retrieval is the identity for an *Abstract* final state, and for a *Between* or *Concrete* final state it is similar to the Abstract/*Between* abstraction relation with the constraint that any ambiguous (that is, in fact : unended) transaction is considered lost. The refinements demand that the final states be retrieved to the same *global world*.

1.3 Overview of the Z specification language through a simplified version of *Mondex*

The *Mondex* case study has been specified and proven by hand using the Z specification language, also called “the Z notation” [Spi92]. This language is based on *schema calculus* and uses the classical logic with the ZERMELO – FRAENKEL set theory (hence the “Z”). It is often extended by adding constructs for special purposes, for instance for sequential systems such as the *Mondex* case study.

In the Z notation, a theory is a set of *set parameters* along with a set of *schemas* depending on those parameters. Both objects and operations are modeled by a *schema*. A schema is

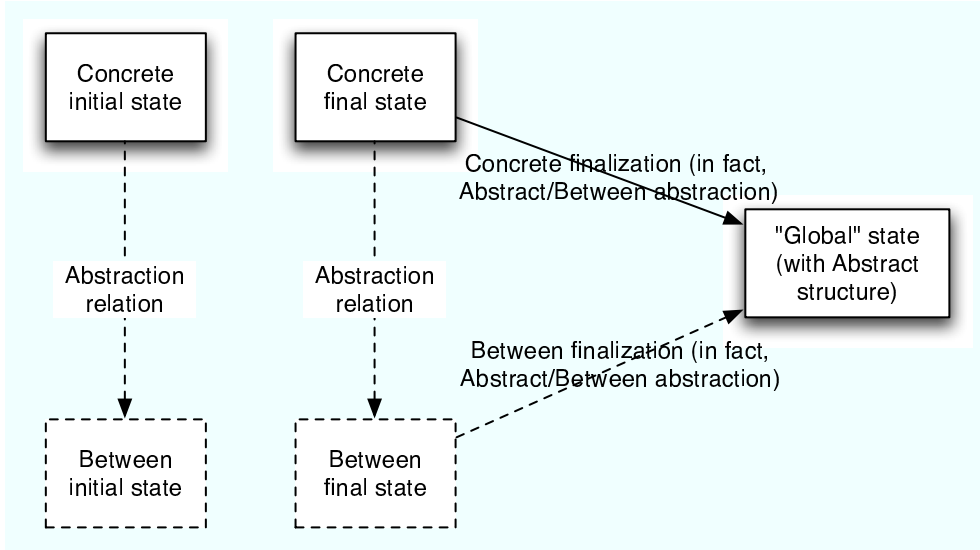


Figure 8: *Between/Concrete initialization and finalization.*

The layout is the same for the *Abstract/Between initialization and finalization*, except that the abstraction relation is different, and the *Abstract finalization* is simply the identity.

a structure gathering *fields* along with *constraints*, logical formulae involving the variables of a schema. The notion of schema comes with the *schema calculus*, a formal reduction rule involving schemas.

For instance, the following *Z* specification shows how the abstract state (*AbWorld*) with the *AbIgnore*, *AbTransferOkay* and *AbTransferLost* operations could have been modeled in a simplified way.

$\frac{AbPurse}{balance, lost : \mathbb{Z}}$
$\frac{balance \geq 0}{lost \geq 0}$

That schema defines the purse structure, *AbPurse*, which has two fields representing the *balance* and *lost* as integers constrained to be non-negative.

Then, the abstract state (*AbWorld*) is defined as follows :

[NAME]

$AbWorld$
$abAuthPurse : NAME \leftrightarrow AbPurse$

That schema defines the abstract state with a field *abAuthPurse* that is a finite functional mapping (\leftrightarrow) between names and purses, representing the set of **abstract authentic purses**. Even though the name is not directly stored within its purse, it allows distinguishing it among all the purses, for instance if two purses have the same *balance* and *lost* values. The name is picked in a set which is given as a parameter (no matter what names actually are). The

finiteness of this relation expresses the fact that there is only a finite number of authentic purses.

Then, the operation *AbIgnore*, which does nothing, is simply defined by the following schema :

$$\boxed{\begin{array}{l} \textit{AbIgnore} \\ \Xi \textit{AbWorld} \end{array}}$$

That schema uses the *invariance* construct Ξ which declares a pre-state and a post-state and constrain them to be equal. That schema could be equivalently rewritten as this one :

$$\boxed{\begin{array}{l} \textit{AbIgnore} \\ \Delta \textit{AbWorld} \\ \hline \textit{abAuthPurse}' = \textit{abAuthPurse} \end{array}}$$

That schema uses one of the most used constructs in the Z specifications for sequential systems, namely the *duplication* construct Δ which declares a pre-state and a post-state. Fields of the pre-state are accessed directly through their names used alone, whereas fields for the post-state have to be appended with a prime ($'$). So *abAuthPurse* refers to the pre-state field, whereas *abAuthPurse'* refers to the post-state field.

The *AbTransferOkay* and *AbTransferLost* operations only differ in the fact that the former increases the *balance* of the “to” purse whereas the latter increases its *lost* store. So, we may use the *schema extension* mechanism, that consists in defining a common transfer schema which will be extended by the schemas defining respectively *AbTransferOkay* and *AbTransferLost*.

$$\boxed{\begin{array}{l} \textit{AbTransferCommon} \\ \Delta \textit{AbWorld} \\ \textit{from}, \textit{to} : \textit{NAME} \\ \textit{value} : \mathbb{N} \\ \hline \{\textit{from}, \textit{to}\} \subseteq \text{dom } \textit{abAuthPurse} \\ \textit{from} \neq \textit{to} \\ \text{dom } \textit{abAuthPurse}' = \text{dom } \textit{abAuthPurse} \\ \forall \textit{name} : \textit{NAME} \mid \textit{name} \in \text{dom } \textit{abAuthPurse} \setminus \{\textit{from}, \textit{to}\} \bullet \\ \quad \textit{abAuthPurse } \textit{name} = \textit{abAuthPurse}' \textit{name} \\ \exists \Delta \textit{AbPurse} \bullet \\ \quad \textit{abAuthPurse } \textit{from} = \theta \textit{AbPurse} \\ \quad \wedge \textit{abAuthPurse}' \textit{from} = \theta \textit{AbPurse}' \\ \quad \wedge \textit{value} \leq \textit{balance} \\ \quad \wedge \textit{balance}' = \textit{balance} - \textit{value} \\ \quad \wedge \textit{lost}' = \textit{lost} \end{array}}$$

The constraints of that schema use the *binding* construct, θ , the most frequent uses of which are to denote either the pre-state or the post-state of a previously given Δ construct : here, $\theta \textit{AbPurse}$ denotes the pre-state object of the existentially quantified $\Delta \textit{AbPurse}$ pre- and post-purses, whereas $\theta \textit{AbPurse}'$ denotes its post-state. In fact, $'$ may be replaced with any other *decoration* depending on the purpose of the specification ; then, θ may be also used in such cases. Whereas Δ is specific to sequential systems, θ is a general construct of the Z *schema calculus*.

The constraint $value \leq balance$ ensures the “from” purse to have sufficient funds for the transaction.

Given that global schema, it may be extended to get the two schemas corresponding to the *AbTransferOkay* and *AbTransferLost* operations :

<i>AbTransferOkay</i>
<i>AbTransferCommon</i>
$\exists \Delta AbPurse \bullet$ $abAuthPurse\ to = \theta AbPurse$ $\wedge abAuthPurse'\ to = \theta AbPurse'$ $\wedge balance' = balance + value$ $\wedge lost' = lost$

<i>AbTransferLost</i>
<i>AbTransferCommon</i>
$\exists \Delta AbPurse \bullet$ $abAuthPurse\ to = \theta AbPurse$ $\wedge abAuthPurse'\ to = \theta AbPurse'$ $\wedge balance' = balance$ $\wedge lost' = lost + value$

Those schemas both *extend* the *AbTransferCommon* schema : they are strictly equivalent to the schemas where *AbTransferCommon* fields and constraints would have been inlined.

<i>AbTransferOkay</i>
$\Delta AbWorld$ $from, to : NAME$ $value : \mathbb{N}$
$\{from, to\} \subseteq \text{dom } abAuthPurse$ $from \neq to$ $\text{dom } abAuthPurse' = \text{dom } abAuthPurse$ $\forall name : NAME \mid name \in \text{dom } abAuthPurse \setminus \{from, to\} \bullet$ $abAuthPurse\ name = abAuthPurse'\ name$
$\exists \Delta AbPurse \bullet$ $abAuthPurse\ from = \theta AbPurse$ $\wedge abAuthPurse'\ from = \theta AbPurse'$ $\wedge balance' = balance - value$ $\wedge lost' = lost$
$\exists \Delta AbPurse \bullet$ $abAuthPurse\ to = \theta AbPurse$ $\wedge abAuthPurse'\ to = \theta AbPurse'$ $\wedge value \leq balance$ $\wedge balance' = balance + value$ $\wedge lost' = lost$

$AbTransferLost$ <hr/> $\Delta AbWorld$ $from, to : NAME$ $value : \mathbb{N}$ <hr/> $\{from, to\} \subseteq \text{dom } abAuthPurse$ $from \neq to$ $\text{dom } abAuthPurse' = \text{dom } abAuthPurse$ $\forall name : NAME \mid name \in \text{dom } abAuthPurse \setminus \{from, to\} \bullet$ $abAuthPurse \text{ name} = abAuthPurse' \text{ name}$ $\exists \Delta AbPurse \bullet$ $abAuthPurse \text{ from} = \theta AbPurse$ $\wedge abAuthPurse' \text{ from} = \theta AbPurse'$ $\wedge value \leq balance$ $\wedge balance' = balance - value$ $\wedge lost' = lost$ $\exists \Delta AbPurse \bullet$ $abAuthPurse \text{ to} = \theta AbPurse$ $\wedge abAuthPurse' \text{ to} = \theta AbPurse'$ $\wedge balance' = balance$ $\wedge lost' = lost + value$

Then, security properties may be stated also using *schemas* :

$NoValueCreated$ <hr/> $\Delta AbWorld$ $Totals$ <hr/> $totalBalance \text{ } abAuthPurse' \leq totalBalance \text{ } abAuthPurse$
--

$AllValueAccounted$ <hr/> $\Delta AbWorld$ $Totals$ <hr/> $totalBalance \text{ } abAuthPurse' + totalLost \text{ } abAuthPurse'$ $= totalBalance \text{ } abAuthPurse + totalLost \text{ } abAuthPurse$

Those two properties use *totalBalance* and *totalLost* functions which can be defined axiomatically using the following *Totals* schema⁴ :

$Totals$ <hr/> $totalBalance, totalLost : (NAME \rightsquigarrow AbPurse) \rightarrow \mathbb{Z}$ <hr/> $totalBalance(\emptyset) = 0$ $totalLost(\emptyset) = 0$ $\forall f : NAME \rightsquigarrow AbPurse; name : NAME; AbPurse \mid$ $name \in \text{dom } f \wedge \theta AbPurse = f(name)$ <ul style="list-style-type: none"> • $totalBalance(f) = totalBalance(name \triangleleft f) + balance$ $\wedge totalLost(f) = totalLost(name \triangleleft f) + lost$

⁴In the original specification, those functions were not defined in a schema, but simply axiomatically, in order to be used more easily in a general purpose (as if they were globally defined).

Here the construct $\theta AbPurse$ refers to the previously universally quantified $AbPurse$ schema. The construct \triangleleft is a relational operation restricting the domain of a function by removing some of its elements (here *name*).

The soundness of that definitions mainly relies on the finiteness of the domain of the function f .

Finally, the theorems stating that the abstract operations make security properties hold, may be enounced as follows :

$$AbTransferOkay \vdash NoValueCreated \wedge AllValueAccounted$$

$$AbTransferLost \vdash NoValueCreated \wedge AllValueAccounted$$

The proof use the constraints of the relevant schemas and eventually the axioms of the *total-Balance* and *totalLost* functions in addition to the ZERMELO – FRAENKEL set theory. We won't detail the proofs here ; however, those theorems are quite obvious since nothing but the *value* of a given transfer may be removed from a purse, and this *value* removed from the “from” *balance* is added :

- either to the “to” *balance*, in which case the *lost* stores are constant, and so is the sum of the *balance* stores
- or to the “to” *lost*, in which case the sum of the *balance* stores decreases, the sum of the *lost* stores increases, but the difference between the pre-state and the post-state *balance* sum is the same as the difference between the post-state and the pre-state *lost* sum, that is exactly *value*
- nowhere else

2 A simplified version of the *Mondex* specification in Alloy

The *Alloy* specification language [Jac02, Jac06] has been developed within Daniel Jackson’s Software Design group at MIT CSAIL. It is a specification language based on first-order logic with relational calculus and transitive closures. It comes with *Alloy Analyzer* [AA], a piece of software using *model-finding* to check specifications written in Alloy, that is trying to find an *instance* of a model to prove its consistency, or a *counterexample* to a theorem to prove its inconsistency. The *Alloy Analyzer* reduces the problem to SAT-solving [Jac00], requiring the user to give a *finite scope*, that is the number of objects (or *atoms*) allowed in an instance or counterexample (also called *search space*).

This section explains the main guidelines I used to build the Alloy model. So the Alloy model described here is a simplified version of the actual, final model. But in the latter version, I managed to model the whole initial Z specification, addressing technical issues raised in Section 3.

2.1 Modeling with the Alloy specification language

The *Alloy* specification language relies on *first-order* logic with relational calculus and transitive closures. Historically, its conception is inspired of the Z notation.

Basically, a specification defines several *relations*. A relation represents a set of *atoms* or *tuples* (the components of which are atoms), depending on its *arity* : a relation can contain only objects of the same arity.

A *model* (or an *instance*) of the specification is the assignment of a set of *atoms* or *tuples* to each relation depending on their arities. Atoms or tuples do not directly appear in the specification, except through relations which are explicitly constrained to be singletons.

2.1.1 Defining signatures and relations

Unary relations are defined as *signatures*, and other relations are defined as the “fields” of those signatures.

The following (simplified) specification models a concrete purse :

```
sig ConPurse {
  name : NAME,
  balance : Int,
  pdAuth : PayDetails,
  exLog : set PayDetails,
  status : STATUS,
  nextSeqNo : Int
}
```

That specification defines a signature, called **ConPurse**, which represents a set of unary objects, that is *atoms*. Their “fields”, given as names along with signature names, actually define relations between the signature being defined and the signatures provided. For instance, the field `name : NAME` defines a binary relation between **ConPurse** and **NAME**. By default, a *binary*

field defined as such is a *function*, unless the user provides a *multiplicity constraint* such as `set` used to define the relation `exLog`, allowing it to match a purse with several transaction details. So, a purse is matched to exactly one name, one *balance* amount, one *status* and one *sequence number*, but can be matched to any number of transaction details in its *exLog*.

On the contrary, a ternary (or more) relation is not a function by default, despite the notation `->` :

```
sig ConWorld {
  conAuthPurse : NAME -> ConPurse,
  ether : set MESSAGE,
  archive : NAME -> PayDetails
}
```

That (simplified) specification models the concrete world. In that case, the `conAuthPurse` relation, which maps a concrete world to its set of authentic purses distinguished by their names, would be simply a ternary relation between `ConWorld`, `NAME` and `ConPurse`, which is not necessarily functional, so that a name could be associated to several purses. So this relation has to be constrained by adding a multiplicity constraint, replacing the corresponding definition with :

```
...
  conAuthPurse : NAME -> lone ConPurse,
...

```

This constraints a tuple formed by a `ConWorld` and a `NAME` to match at most one purse (that is, less than or equal than **one**) — indeed, a name may refer to no authentic purse. However, the global *archive* is defined here as a ternary relation between worlds, names, and transaction details, allowing to “tag” each archived transaction with the names of the purses having logged it. So, as a purse may archive several transaction details through several *readExceptionLog* operations, it is necessary to not define this relation as functional.

It is worth noting that the definition of a relation constraints the components of its tuples to belong to certain signatures. But by default, signatures are disjoint (to avoid this, Alloy provides a special mechanism described further down). Thus, for instance, `conAuthPurse & archive`, which is the *intersection* of the ternary relations `conAuthPurse` and `archive`, is always empty because `ConPurse` and `PayDetails` are disjoint.

2.1.2 Defining logical constraints : *facts*. The relational calculus

Although multiplicity constraints enforce the `conAuthPurse` to match at most one purse to each combination of a concrete world and a name, it is of course not enough to ensure that the name of the purse given by its `name` relation actually corresponds to the name it is associated to. That is why, *logical* constraints have to be added. In fact, they are necessary to model most axioms of the original Z theory.

A logical constraint is defined as a *fact* :

```
fact NamesMatch {
  all c : ConWorld, n : NAME | n.(c.conAuthPurse).name in n
}
```

This fact is an universally quantified formula. As Alloy is based on first-order logic, this formula quantifies over atoms. So, in the quantified formula, `c` and `n` are unary relations that contain exactly one tuple.

This fact uses one of the most used constructs of *relational calculus*, namely the *join* operator (`.`). Given two relations α and β , their join (or *composition*), $\alpha.\beta$, is defined as follows :

$$\alpha.\beta \triangleq \{(a_1, \dots, a_p, b_1, \dots, b_q) \mid \exists x, (a_1, \dots, a_p, x) \in \alpha \wedge (x, b_1, \dots, b_q) \in \beta\}$$

That is : take a tuple of α and a tuple of β such that the former's last component matches the latter's first, then *join* the two tuples by blasting the common component. The Alloy join operator is similar to the “database join” \bowtie but the latter would keep the common component.

Consider the subexpression `c.conAuthPurse`. As `c`, the concrete world quantified over, is an unary relation, $p = 0$ in the above definition, so that that join expression finally corresponds to the tuples $(name, purse)$ representing the authentic purses of the concrete world c along with their associated names. As moreover `c` is a singleton, this matches the common interpretation where the signature would be viewed as a *record* or a *class* and the join operator as a *membership* operator. Thus, we see that `c.conAuthPurse` represents a binary relation.

Then, `n.(c.conAuthPurse)` is an unary relation, representing the set of authentic purses for the concrete world `c` that are associated with the name `n`. That is, the set of the purses matched with the tuple (c, n) in the relation *conAuthPurse*. As c and n are atoms, the multiplicity constraints ensures that there is at most one such purse.

But there can be no such purse, in such case `n.(c.conAuthPurse)` is empty and, subsequently by the definition of join, so is `n.(c.conAuthPurse).name` (which is equal to `(n.(c.conAuthPurse)).name` as the join operator is defined left-associative). That is why the constraint is only `in`, that is a relation *inclusion* rather than an equality. Indeed, `n.(c.conAuthPurse).name = n` would have constrained the existence of an authentic purse for each name and each concrete world, which is too strong. So this inclusion constraint naturally holds if there is no such purse. In the other case where there is exactly one purse denoted by `n.(c.conAuthPurse)`, then, as `name` is a function matching each purse with exactly one name, `n.(c.conAuthPurse).name` denotes exactly one name. In that case, as `n` is a singleton, the inclusion implies the equality.

This constraint could even have been written without any quantification, thanks to the relational calculus :

```
fact NamesMatch {
  conAuthPurse.name in ConWorld->iden
}
```

That constraint, instead, uses the construct `->` which is simply the cartesian product. This product associates the whole signature `ConWorld`, which is indeed a unary relation gathering all the atoms corresponding to a concrete world, with `iden`, which is the *identity* relation (that is, the set of all the tuples (x, x) where x is any atom). On the left-hand side of the inclusion relation, `conAuthPurse.name` is a ternary relation between a concrete world and two names, the one associated to a purse through `conAuthPurse`, the other being the name of the *same* purse (due to the join) through the `name` relation. So the inclusion implies that for any concrete world, two such names must be equal (because of `iden`).

We see that in the general case where relations can contain more than one tuple, the join operator cannot be interpreted as a class membership operator, although that interpretation is locally interesting and helps the user understand the mechanism of signatures.

Besides those operators, more classical operators such as union (+), intersection (&), difference (-), and also restriction on both sides (left <:, right >:), are of course available in Alloy. The Alloy specification language also provides the *transitive closure* (^) and the *reflexive-transitive closure* (*) of a binary relation.

2.1.3 Signature extension and inclusion

By default, signatures are disjoint one to the other. But *Alloy* offers a mechanism to allow the user to define a family of signatures included in a “mother signature”.

For instance, as shown before, a concrete purse has a *status* among *eaFrom*, *eaTo* (expecting any from/to = “idle”), *epr* (“from” expecting request), *epv* (“to” expecting value), *epa* (“from” expecting acknowledgment). In Alloy, the status could be defined as follows :

```
sig STATUS {}
one sig eaFrom, eaTo, epr, epv, epa in STATUS {}
```

The `one` keyword is a multiplicity constraint enforcing each of the followingly defined signatures to be a singleton.

Through the `in` construct, those signatures are defined to be included in the `STATUS` signature. But the `in` construct does not ensure that those signatures will be disjoint. In other words, `in` signatures are simply subsets of already given signatures.

To constrain the signatures to be disjoint, we should use another construct :

```
sig STATUS {}
one sig eaFrom, eaTo, epr, epv, epa extends STATUS {}
```

Not only does the `extends` construct define the signatures to be included in `STATUS`, but it also ensures that they will be disjoint one to the other. More precisely, any two signatures which *extend* the same signature are necessarily disjoint. That is, if we had defined the statuses as follows :

```
sig STATUS {}
one sig eaFrom in STATUS {}
one sig eaTo, epr, epv, epa extends STATUS {}
```

then `eaFrom` could have been equal to one of the others.

But in both cases, the extension is not enough, as it does not enforce a `STATUS` to be necessarily included in (here, equal to) one of the defined extensions. This can be constrained by defining `STATUS` as an `abstract` signature :

```
abstract sig STATUS {}
one sig eaFrom, eaTo, epr, epv, epa extends STATUS {}
```

This specification is strong enough to model the status of a concrete purse.

Indeed, an `abstract` signature is a signature, the atoms of which necessarily belong to a signature extending it. We could even have defined the statuses as follows :

```

abstract sig STATUS {}
abstract sig idle extends STATUS {}
one sig eaFrom, eaTo extends idle {}
abstract sig busy extends STATUS {}
one sig epr, epv, epa extends STATUS {}

```

In that case, a status must necessarily belong to a signature extending either `idle` or `busy`.

In fact, `extends` and `abstract` constructs may be rewritten as `in` constructs by adding some facts. For instance, the specification above is equivalent to :

```

sig STATUS {}
sig idle in STATUS {}
sig eaFrom, eaTo in idle {}
sig busy in STATUS {}
sig epr, epv, epa in STATUS {}
fact status_extended {
  STATUS = idle + busy
  no idle & busy
}
fact idle_extended {
  idle = eaFrom + eaTo
  no eaFrom & eaTo
}
fact busy_extended {
  busy = epr + epv + epa
  no epr & epv
  no (epr + epv) & epa
}

```

where `&` is the intersection, `+` is the union, and `no` simply constraints a relation to be empty. The constraints of the form :

```
STATUS = idle + busy
```

correspond to the `abstract` construct, whereas the constraints of the form :

```
no idle & busy
```

correspond to the `extends` construct.

2.1.4 Auxiliary predicates and functions

Besides data structures, a specification has to describe operations. It is mostly not necessary to model them as objects, and very often logical formulae are enough to define them.

That is why Alloy provides predicate definition. Another purpose is to prevent formulae from growing, through defining them “piecewise”. To the latter purpose, Alloy also provides auxiliary function definition.

Whereas a predicate defines an auxiliary formula, a function returns a relation. Both take relations as arguments. Their definition constraint arguments to be included in certain fixed relations. But beyond this constraint, relations are of arbitrary arity and multiplicity. However, functions or predicates may not be carried like higher-order relations, they must be applied.

For instance, consider the *Increase* operation (a purse increases its sequence number). It can be defined by two predicates : *IncreasePurseOkay* operates on a purse first, then this predicate is *promoted* to the world level through *Increase*.

```

pred IncreasePurseOkay (p, p' : ConPurse) {
  p'.name = p.name
  p'.exLog = p.exLog
  p'.status = p.status
  p'.balance = p.balance
  p'.pdAuth = p.pdAuth
  int p'.nextSeqNo >= int p.nextSeqNo
}

pred Increase (w, w' : ConWorld) {
  w'.conAuthPurse.ConPurse = w.conAuthPurse.ConPurse
  some n : NAME {
    n in w.conAuthPurse.ConPurse
    (NAME - n) <: w'.conAuthPurse = (NAME - n) <: w.conAuthPurse
    IncreasePurseOkay (n.(w.conAuthPurse), n.(w'.conAuthPurse))
  }
  w'.ether = w.ether
  w'.archive = w.archive
}

```

It is necessary to indicate for each argument a relation in which it should be included.

The expression `w.conAuthPurse.ConPurse` denotes the *domain* of the relation `w.conAuthPurse`. Indeed, by the definition of join, it corresponds to the set of names associated with any concrete purse (hence `ConPurse`) through `w.conAuthPurse`. In other words, it corresponds to the names authentic to the world *w*. We could then have defined this as a function :

```

fun authenticNames (w : ConWorld) : NAME {
  w.conAuthPurse.ConPurse
}

```

As for a predicate, it is necessary to indicate an including relation for each argument, but also for the result of the function.

Then, the *Increase* predicate above could be rewritten as follows :

```

pred Increase (w, w' : ConWorld) {
  authenticNames (w') = authenticNames (w)
  some n : NAME {
    n in authenticNames (w)
    (NAME - n) <: w'.conAuthPurse = (NAME - n) <: w.conAuthPurse
    IncreasePurseOkay (n.(w.conAuthPurse), n.(w'.conAuthPurse))
  }
  w'.ether = w.ether
  w'.archive = w.archive
}

```

The $<$: operator corresponds to the left-hand-side restriction. It is used to constraint the invariance of any other purse than the increasing one.

Important. The arguments of predicates or functions are relations of *arbitrary multiplicity*. In particular, they could be not singletons. Thus, the definitions above of *IncreasePurseOkay* and *Increase* may not make sense if their arguments are not singletons. But in most cases, there is no worrying, as the arguments actually used when predicates or functions are being called are in fact singletons.

2.1.5 Assertions

Not only does a specification describe data structures and constraints, but it also provides theorems which the user wants to show that the specification makes hold. This is the purpose of defining *assertions*.

Those assertions are to be tackled by specification analyzers like the *Alloy Analyzer* (see below). However, besides that project, there are very few attempts of analyzing Alloy specifications. One of the most achieved is *Prioni* [AKMR03] which translates *Alloy* specifications for use with the *Athena* [ath] proof assistant.

For instance, if the concrete *Ignore* operation is defined as follows :

```

pred Ignore (w, w' : ConWorld) {
  authenticNames (w) = authenticNames (w')
  all n : NAME | n in authenticNames (w) implies {
    n.(w'.conAuthPurse).name = n.(w.conAuthPurse).name
    n.(w'.conAuthPurse).exLog = n.(w.conAuthPurse).exLog
    n.(w'.conAuthPurse).pdAuth = n.(w.conAuthPurse).pdAuth
    n.(w'.conAuthPurse).status = n.(w.conAuthPurse).status
    n.(w'.conAuthPurse).nextSeqNo = n.(w.conAuthPurse).nextSeqNo
    n.(w'.conAuthPurse).balance = n.(w.conAuthPurse).balance
  }
  w'.ether = w.ether
  w'.archive = w.archive
}

```

which states that any authentic purse keeps its properties invariant⁵, then we may ask the analyzer to show that the *Ignore* operation is a particular case of *Increase* :

```

assert IgnoreImpliesIncrease {
  all w, w' : ConWorld | Ignore (w, w') implies Increase (w, w')
}

```

In fact, this theorem appears to be true, because it corresponds the case where the inequality of the sequence number in *IncreasePurseOkay* is an equality.

Assertions use the same logic as any other formula (constraints, for instance), except that they may be *skolemized* : the user is allowed to make non-nested higher-order universal quantifications.

⁵Instead of defining $w'.conAuthPurse = w.conAuthPurse$, which might generate spurious counterexamples where different purses would have had the same properties. This is a more general issue discussed further, in Section 3.5.

For instance, consider the following *Rab* predicate, which models the refinement of the *Abstract* by the *Between*. Assume it is defined as a predicate of the form :

```
pred Rab (a : AbWorld, b : BetweenWorld, chosenLost : PayDetails) {...}
```

assuming the *Abstract* and the *Between* worlds have been modeled in Alloy by respectively *AbWorld* (with the *AbIgnore* operation) and *BetweenWorld* signatures, with *BetweenWorld* extending *ConWorld*. In this predicate, contrary to the *a* and *b* arguments, *chosenLost*, which represents the set of the details of the ambiguous transactions chosen to be considered lost, is not supposed to be a singleton. Then, the theorem stating that the *Increase* operation (defined above) refines the abstract *AbIgnore* would be defined as the following assertion :

```
assert RabIncrease {
  all a' : AbWorld, b, b' : BetweenWorld, chosenLost' : set PayDetails | {
    Rab (a', b', chosenLost')
    Increase (b, b')
  } implies some a : AbWorld {
    Rab (a, b, chosenLost')
    AbIgnore (a, a')
  }
}
```

Actually, this assertion would be equivalent to defining auxiliary signatures : either one signature with a field to represent the set of transactions chosen lost, then first-order quantifying over objects of this signature and using the field to get the set of transaction details :

```
sig ChosenLost {cl : set PayDetails}
assert RabIncrease {
  all a' : AbWorld, b, b' : BetweenWorld, chosenLost' : ChosenLost | {
    Rab (a', b', chosenLost'.cl)
    Increase (b, b')
  } implies some a : AbWorld {
    Rab (a, b, chosenLost'.cl)
    AbIgnore (a, a')
  }
}
```

or, more simply, a *in* signature directly representing the set of transactions chosen lost, then no quantification over the choice is needed :

```
sig chosenLost' in PayDetails {}
assert RabIncrease {
  all a' : AbWorld, b, b' : BetweenWorld | {
    Rab (a', b', chosenLost')
    Increase (b, b')
  } implies some a : AbWorld {
    Rab (a, b, chosenLost')
    AbIgnore (a, a')
  }
}
```

It is important to point out the fact that this only works for *unnested* and *universal* higher-order quantifications. That is why the condition over the pre-state reuses the post-*chosenLost* (*chosenLost'*). That is, the following attempt to existentially quantify over the pre-*chosenLost* would be a purely higher-order assertion :

```
assert RabIncrease {
  all a' : AbWorld, b, b' : BetweenWorld, chosenLost' : set PayDetails | {
    Rab (a', b', chosenLost')
    Increase (b, b')
  } implies some a : AbWorld, chosenLost : set PayDetails {
    Rab (a, b, chosenLost)
    AbIgnore (a, a')
  }
}
```

Even though the method enclosing the set of transaction details into a field of a **ChosenLost** signature first-order-quantified over gives a first-order assertion, we shall see later that this method would define false theorems in most cases.

2.1.6 Modules

An Alloy specification may be splitted into several *modules* in order to group signatures and predicates according to their actual dependencies. In other words, the module system allows organizing theories, for instance to point out the fact that one module is independent on another.

For instance, one could model the *Abstract* world in the module **a**, then the *Concrete* and *Between* worlds in the module **cb**. Then, the *Rab* refinement predicate would be defined in a third module, **rab**, which would *open*, that is “import”, the first two modules, through the following statements :

```
open a
open cb
```

Then, the signatures and predicates defined in both modules become available.

The module system also allows defining *parametric* modules, that is modules taking signatures as parameters to add constraints over them or even signatures extending them. For instance, the standard distribution of the *Alloy Analyzer* provides a toolkit of all-purpose modules. Among those, **util/ordering** models a total order. This module takes a signature and defines additional constraints over this signature to enforce its ordering. It begins with the following statement, declaring the signature taken as a parameter :

```
module util/ordering [X]
```

where **X** is the name used in the module to represent the signature given as a parameter. If we want to use this module instead of integers to model the sequence numbers, then we can declare a **SEQNO** signature and make it ordered through passing it as a parameter to the **util/ordering** module being imported :

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

But what should happen if we wanted to define two ordered signatures ? Opening twice would hide the definitions of the opened module. That is, assume for instance the `minimum` signature is defined in `util/ordering` to represent the least element of the total order. Then, to which order should `minimum` refer once used in the module that imported `util/ordering` ?

To solve this issue, Alloy allows the user to open a module and assign it an alias. For instance :

```
sig SEQNO, SEQNO2 {}
open util/ordering [SEQNO] as seqord1
open util/ordering [SEQNO2] as seqord2
```

Then, the module is not really opened, but it is available in two “versions” ; then, to refer to the `minimum` signature of either version, the user has to *qualify* it with the alias, in either way depending on the “version” referred to :

```
seqord1/minimum
seqord2/minimum
```

It is worth noting that this *qualification* mechanism can also be used to refer to a module that has not been imported.

However, despite this feature, the current semantics prevents the user from importing twice the same module with the same parameters. Thus it is impossible to define two different orders for the same signature using `util/ordering`. Nevertheless, the basic module features provided by *Alloy* in this section are quite useful in practice.

Actually, assuming that the *qualification* mechanism is never used in any other case than aliased imported modules, the module system allows to show that theories are independent. For instance, although I used modules to “chain” theories (which form, then, a linear dependency chain) in the naive version of the *Mondex* model, the final version of the model is organised in several groups so that the *Concrete* model does not depend on the *Abstract* one (as it was the case in the former model because of finalization, which uses a “global world” having the same structure as the *Abstract* world).

2.2 Analyzing the specification using the *Alloy Analyzer*, a model finder

The *Alloy* specification language comes with the *Alloy Analyzer* [AA], a piece of software intended to analyze specifications written in *Alloy*. It uses *model-finding* : instead of proving assertions, it tries to find counterexamples to them. But to this purpose, the model has to be assumed to be not only finite, but also bounded : the user has to specify a *scope*, that is maximal number of objects.

The *Alloy Analyzer* has been written in Java by Ilya Shlyakter, a former member of Daniel Jackson’s Software Design group. It is available on the *Alloy* website, <http://alloy.mit.edu>. Although widely used in the industry, it is no longer maintained. Currently, the SDG is developing *Kodkod*, an improved engine for the *Alloy Analyzer* but keeping the principle of model-finding.

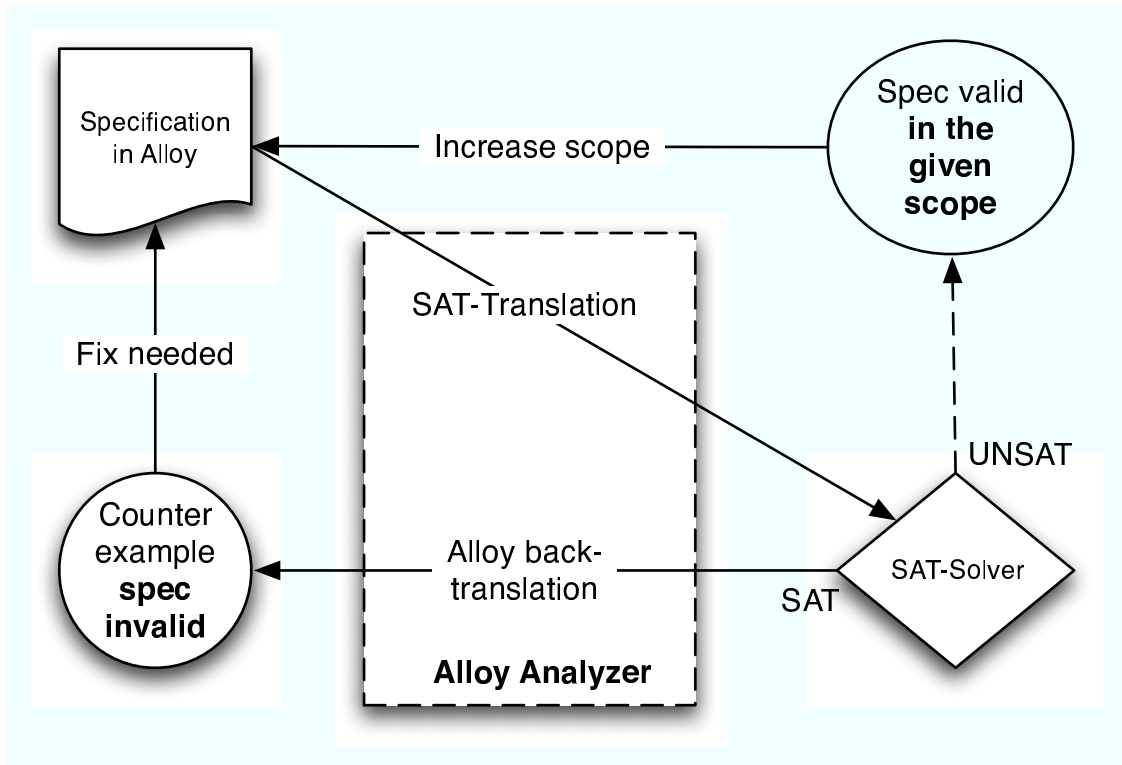


Figure 9: *Principle of model-finding through SAT-solving with the Alloy Analyzer*

2.2.1 Type checking

Strictly speaking, there are no types in *Alloy*. But, as shown before, relation definitions constrain the components of their tuples to belong to certain signatures. Thus, it is possible to write relational expressions that are *always* empty, for instance `conAuthPurse & archive`. Mostly such expressions are *user errors*. That is why, even though they may be strictly well-defined, the *Alloy Analyzer* rejects them as such.

Of course the *Alloy Analyzer* also rejects truly malformed relational expressions such as the join of two unary relations⁶, or the union of two relations of different arities, as Alloy requires a relation to contain only tuples of the same arity.

The *Alloy Analyzer* also rejects higher-order quantifications in predicates or assertions when they are not skolemizable.

2.2.2 Checking assertions. Notion of finite scope.

The *Alloy Analyzer* does not prove assertions. Instead, it tries to *check* them, by finding a *counterexample* to them, that is, a model satisfying all the constraints but not the assertion being checked.

To do that, it translates the Alloy specification, namely the constraints and the *negation* of the assertion, to a (often huge) boolean formula [Jac00] that it asks a SAT-solver to try to satisfy it. If it is satisfied, then the assignation of boolean variables is translated back to an assignation

⁶Rigorously speaking, in such a join expression $\alpha.\beta$ where α and β are unary, we would have $p = q = 0$ in the definition of join, then the result would be a relation of arity 0 : if $\exists x, x \in \alpha \wedge x \in \beta$ (following the definition), then $\alpha.\beta$ would contain the *empty tuple*, else the relation would contain *no tuple*. But Alloy does not handle relations of arity 0.

of atoms to signatures and tuples to relations, which gives a model satisfying the negation of the assertion, that is a *counterexample* to the assertion.

But that translation requires a *finite* number of atoms. More precisely, the analyzer must know in advance a *bound* on the number of atoms in each signature, that is called a *scope*. The user has to provide the scope for each assertion being checked. If the analyzer gives a counterexample, then the assertion is invalid for *any* scope. But if the analyzer does not, the assertion might still be false in a higher scope. Increasing the scope only raises the *confidence level*.

To check the `IgnoreImpliesIncrease` assertion, we just have to provide the scope through a `check` statement like this one :

```
check IgnoreImpliesIncrease for 5
```

This bounds the number of objects to 5 for each signature other than `abstract` or `in` or signatures constrained to be singletons through `one`. It is also possible to define a scope for *some* signatures, leaving a default scope for the others. For instance, as only two concrete worlds are considered (because there are no constraints relating different concrete worlds outside of an operation), it is sound to bound the scope to 2 concrete worlds :

```
check IgnoreImpliesIncrease for 5 but 2 ConWorld
```

It is also possible to define scope for *each* signature :

```
check IgnoreImpliesIncrease for 5 ConPurse, 2 ConWorld, 5 PayDetails, 5 Int
```

By default, a scope defines an *upper bound* on the number of atoms in a signature. But specifying the `exactly` keyword enforces this bound to be reached, and the signature to have precisely the specified number of atoms :

```
check IgnoreImpliesIncrease for 5 but exactly 2 ConWorld
```

In all those cases, the *Alloy Analyzer* finds no counterexample. But this bounding method could not be sound if applied to the purses, because for instance the *Between* constraints could relate them.

Again, this does not show that the assertion holds for any scope. If we want to dramatically increase our confidence level :

```
check IgnoreImpliesIncrease for 100 but 2 ConWorld
```

then in that case, the *Alloy Analyzer* crashes (out of memory), namely while translating to SAT-formula.

Nevertheless, the scope should be high enough to make sense.

For instance, suppose we define the following reciprocal assertion :

```
assert IncreaseImpliesIgnore {  
  all w, w' : ConWorld | Increase (w, w') implies Ignore (w, w')  
}
```

We know that this assertion should not hold, as the sequence number could change. But if we check the assertion for a too small scope :

```
check IncreaseImpliesIgnore for 5 but 1 ConPurse
```

the *Alloy Analyzer* will find no counterexample. Indeed, with only one purse, its sequence number cannot change (because this would require another purse to exist, the post-purse with

the new sequence number), and the assertion is true. But as expected, if we increase the scope :

```
check IgnoreImpliesIncrease for 5 but 2 ConPurse
```

then the *Alloy Analyzer* finds a counterexample, as expected.

2.2.3 “Running” predicates : sanity-check simulation

The *Alloy Analyzer* can also “run” a predicate : it tries to make it satisfiable. That is, the *Alloy Analyzer* translates the specification and the predicate — not its negation as for an assertion — to a boolean formula, and by translating back an eventual assignation of boolean variables to an assignation of atoms to signatures and tuples to relations, it finds an *example* making the predicate hold. This is a *simulation* of the model, showing that it is *consistent* : it is a *sanity-check*.

To put it in a nutshell, “running” a predicate is equivalent to checking the assertion corresponding to its negation.

The predicate may have arguments ; in that case, the *Alloy Analyzer* will existentially quantify over them, that is the analyzer will try to find an assignation of tuples to relations being passed to the predicate being “run”.

As for assertions, it is also necessary to provide a scope for signatures. For instance, to show that a *Increase* operation may happen, we may provide the scope by a `run` statement like this one :

```
run Increase for 5 but 2 ConPurse
```

Then, the *Alloy Analyzer* will find an example to the predicate, showing that it is satisfiable.

But although that relations passed as arguments to a predicate have an *arbitrary multiplicity*, the implicit existential quantification made by the *Alloy Analyzer* when “running” a predicate is only first-order. To enforce skolemisation, the user has to explicitly specifying the existential higher-order in the predicate. Then this predicate may be “run” in a standalone simulation. But if this predicate is referred to by an assertion, then the *Alloy Analyzer* will reject the specification as this existential quantifier will be “inlined” in the assertion and there becomes not skolemizable.

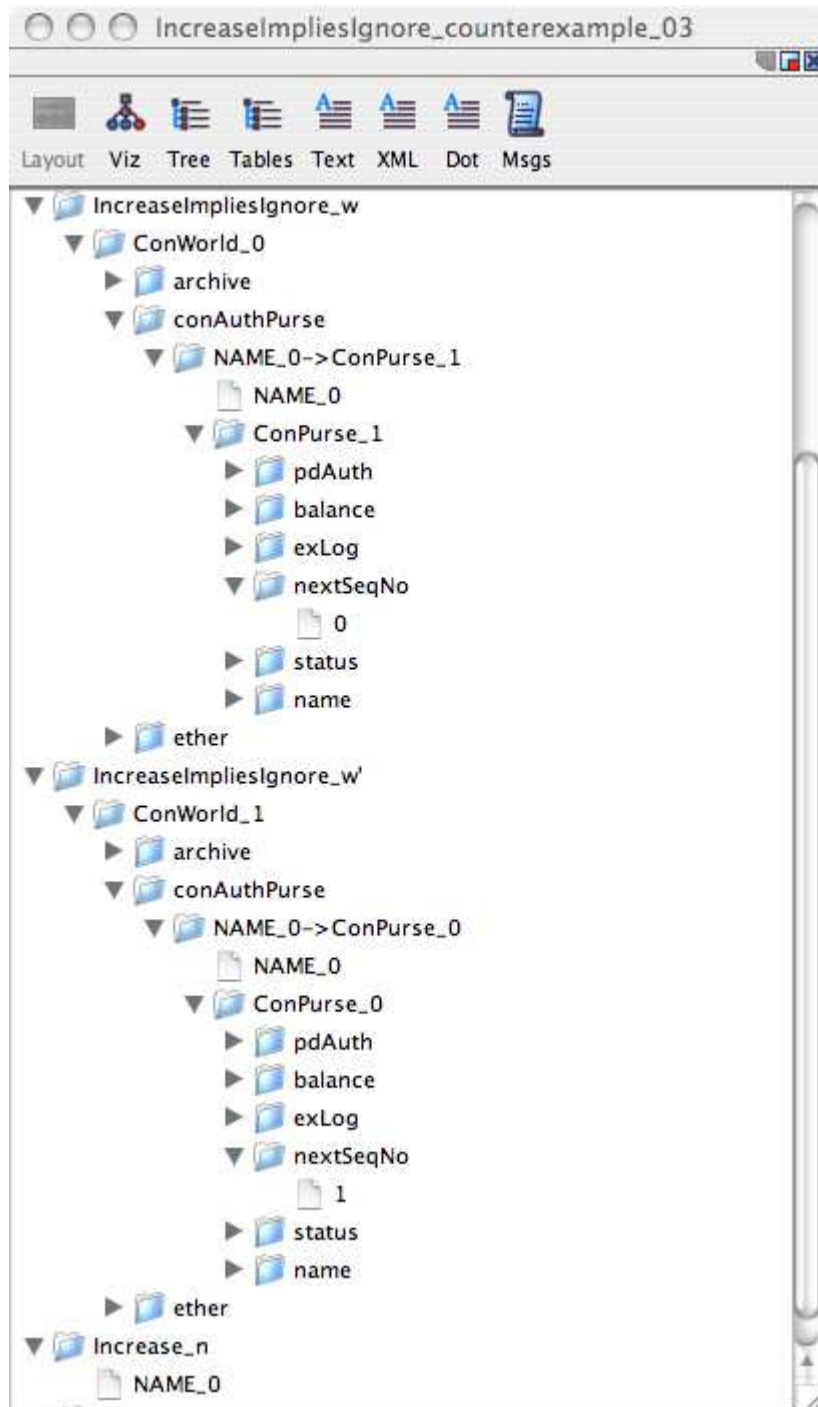


Figure 10: Counterexample to the *IncreaseImpliesIgnore* assertion

The counterexample shows a *Increase* operation between two Concrete worlds, where a purse, the name of which is indicated by *Increase_n* on the bottom, gets its sequence number strictly increased.

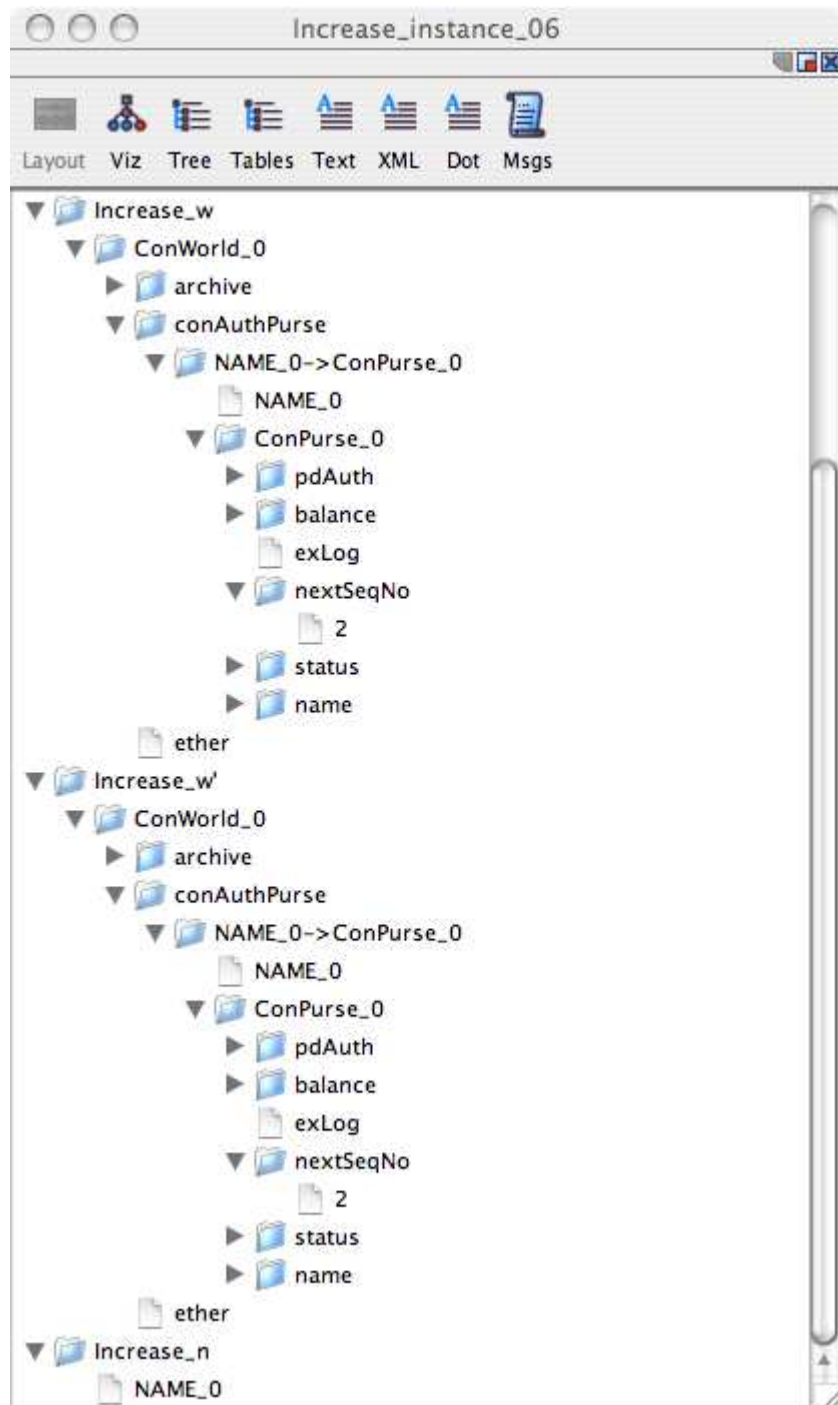


Figure 11: *Example to the Increase predicate*

*The example shows a Increase operation between two Concrete worlds, where the increasing purse, the name of which is indicated by **Increase_n** on the bottom, gets its sequence number actually unchanged.*

3 Modeling the *Mondex* case study in *Alloy* : technical issues encountered

The use of the *Alloy* method (the *Alloy* specification language and the *Alloy Analyzer*) raised some technical issues at different levels, either due to the logical conception of the *Alloy* specification language, or to the current implementation of the *Alloy Analyzer*. I solve them in two steps : first, I wrote a first naive version of the *Mondex* specification in *Alloy* that I presented on May 26, 2006 at the 3rd VSR/NET workshop. That version used to follow the Z specification as close as possible. But since I came back to MIT on June 6th, I optimized the model in order to make it more conform to the Alloy “idiom”. This optimisation has eventually pointed out some errors in the former model.

3.1 Finiteness

Although there might be an infinite number of abstract or concrete worlds, or even transactions, the *Mondex* case study requires finiteness properties to be shown :

- for any abstract, between or concrete world, there must be only a *finite* number of authentic purses. This is required to compute the sum of *balance* values when abstracting a Between world to an Abstract one.
- for any between or concrete world, the set of the transaction details logged by “to” purses has to be finite. This is required to compute the sum of *lost* values when abstracting a Between world to an Abstract one.

As *Alloy* is based on first-order with transitive closures, it is possible to constrain a given set to be finite. It is enough to define an “immediate successor” function, the induced order of which (by transitive closure) is bounded. For instance, we could redefine the concrete world by adding, for each concrete world, such a function over authentic names, as follows :

```
sig ConWorld {
  conAuthPurse : NAME -> ConPurse,
  finiteAuthPurse : NAME -> lone NAME, -- 1
  ether : set MESSAGE,
  archive : NAME -> PayDetails
}
fun authenticNames (c : ConWorld) : NAME {
  c.conAuthPurse.ConPurse
}
fact finiteAuthPurseAxioms {
  all c : ConWorld {
    c.finiteAuthPurse in authenticNames(c)->authenticNames(c) -- 2
    no iden & ^(c.finiteAuthPurse) -- 3
    one authenticNames(c) - authenticNames(c).(c.finiteAuthPurse) -- 4
    one authenticNames(c) - c.finiteAuthPurse.authenticNames(c) -- 5
  }
}
```

The character sequence `--` introduces a *comment* in Alloy. There are the meanings of the constraints denoted by the numbers indicated beneath the comments.

1. A concrete world and a name are matched with at most one name.
2. For a given world `c`, any tuple in `c.finiteAuthPurse` has both components in the set of names authentic for `c`.
3. For a given world `c`, there is no tuple with equal components in the transitive closure of `c.finiteAuthPurse`. This means that `c.finiteAuthPurse` is acyclic.
4. For a given world `c`, there is exactly one authentic name having no “pre-image” through `c.finiteAuthPurse`. This name is actually the “maximum”. Indeed, `authenticNames(c).(c.finiteAuthPurse)` represents the set of all the “images” obtained from authentic names through `c.finiteAuthPurse` (by definition of join).
5. For a given world `c`, there is exactly one authentic name having no “image” through `c.finiteAuthPurse`. This name is actually the “minimum”. Indeed, `(c.finiteAuthPurse).authenticNames(c)` represents the set of all the “pre-images” obtained from authentic names through `c.finiteAuthPurse` (by definition of join).

But it still remains impossible in *Alloy* to directly *show* the finiteness of any set or relation. Indeed, it would be necessary to at least existentially quantify over a relation, as any attempt to express finiteness involves an “external” relation. In fact, transitive closure only expresses *reachability* in a finite number of steps ; this does not imply that the underlying set is finite.

Thus, to be able to analyze a specification using the *Alloy* specification language (with or without the *Alloy Analyzer*), it is necessary to *drop* any property about finiteness.

Moreover, because of the finite scope, the analysis with the *Alloy Analyzer* assumes that every model is finite, that is there is necessary a finite number of atoms (thus, every signature or relation is finite).

Remark. In fact, it is worth noting that the finiteness of the set of *Abstract* (or *Concrete*) authentic purses can be shown *indirectly*. Indeed, the set of *Abstract* authentic purses does not change through an operation. In the same way, the finiteness of the set of the transaction details logged by “to” purses can be shown indirectly, as at most one transaction is logged by a purse during an operation. So, as regards the *Mondex* case study, finiteness properties may be skipped without worrying. More globally, it could be also an interesting idea to abstract a whole model by taking only authentic purses, which would necessarily give a finite number of purses.

3.2 Integers

The Alloy specification language provides support for integers, namely comparison, sum, and sum of sets of values, provided the model is assumed *finite*. Product, however, is not available, as the model has to be kept *first-order*.

So, the Abstract world and the security properties could have been modeled in *Alloy* as follows :

```

sig AbPurse { balance, lost : Int }
sig AbWorld {
  abAuthPurse : NAME -> lone AbPurse,
}
pred NoValueCreated (w, w' : AbWorld) {
  sum NAME.(w'.abAuthPurse).balance <= sum NAME.(w.abAuthPurse).balance
}
pred AllValueAccounted (w, w' : AbWorld) {
  sum NAME.(w'.abAuthPurse).balance + sum NAME.(w'.abAuthPurse).lost
  = sum NAME.(w.abAuthPurse).balance + sum NAME.(w.abAuthPurse).lost
}

```

Those predicates use the `sum` construct, which computes the sum of a set of values. This construct only works if the model is assumed *finite*, which is automatically the case if the *Alloy Analyzer* is used.

However, the current implementation of integers in the *Alloy Analyzer* prevents efficient analyses. Indeed, the translation of integers and their operations into boolean formulae consume a lot of time and space, and dramatically reduce 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 :

- Sequence numbers do not use any arithmetical property of \mathbb{N} . They only use comparison.
- Surprisingly, even amounts may be represented without integers, although they seem to use all the properties of \mathbb{N} .

3.2.1 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 a total order to model them.

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

```

sig SEQNO {}
open util/ordering [SEQNO]

```

Moreover, an obscure “hack” allows the *Alloy Analyzer* to treat this module in an optimized way. In particular, when a scope is given to `SEQNO`, it is *exact* : whereas by default a scope is an *upper bound* on the number of atoms in a signature, this bound is reached for any signature using `util/ordering`, as if the `exactly` flag was specified along with its scope in a `check` (assertion check) or `run` (predicate simulation) statement.

3.2.2 Amounts

It is somewhat surprising that amounts may be expressed without using integers. But the point is that, even though all the first-order properties of integers are used, 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 ! So, with this approach, operations are redefined as follows :

- The sum of two values is the union of the corresponding sets of coins
- The difference of two values is the (set) difference of the corresponding sets of coins
- 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, lost : set Coin }
sig AbWorld { abAuthPurse : NAME -> lone AbPurse }
```

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.

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)
    no p : AbPurse {
      p in NAME.(w.abAuthPurse)
      some p.balance & p.lost
    }
  }
}
```

The `no disj` quantification expresses that there are no distinct (“disjoint” singletons) purses verifying the property, namely the fact that there is a coin common to the union of their *balance* and *lost*.

These constraints only apply to abstract authentic purses (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
}
```

Equivalent constraints to avoid coin sharing have to be added to the *Concrete* world. First, in the former model, I 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.log)
      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 abstract
3. A purse has no coin common to its balance and a transaction that has been logged in the global *archive*

As I came back from England in June, trying to optimize the specification, I figured out the fact that although constraint 2 makes sense, constraints 1 and 3 were too strong.

Indeed, as regards constraint 3 :

- the “to” purse has received the money and sends the acknowledgment message, but 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*, as the “to” *balance* contains the coins corresponding to its value.
- the “to” purse has just send the request message but aborts, then logging . 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*.

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, which precisely defines . This relation relies on the definition of two functions :

- *definitelyLost* corresponds to the set of details referring to transactions definitely lost, that is 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, where the “to” purse expects the value while the “from” purse has already sent it and either expects the acknowledgment or has logged the transaction before the “to” received the value. In this case, the value is stored in no other

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 alone guarantees that the equivalent *balance* and *lost* stores are disjoint. This constraint, indeed, prevents a coin from being in a *balance* and a *lost* store at the same time, even if the purses are distinct. However,

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 optimize the Alloy specification after I came back from England. Indeed, whereas I first defined those constraints within the *Concrete* world, I then moved them to the *Between* world. Then, I figured out the fact that the *Between/Concrete* refinements did not hold, as some operations were impossible in some situations, precisely those situations due to too strong constraints in the *Between* that are not necessarily kept through operations. Thus, the new constraints defined here are actually in the *Between* world, not the *Concrete*.

However, it does make sense to set some additional constraints over the *Between* world, even though no counterexamples are yielded without them :

- before the *Val* operation, the balance of the “to” purse must not have coins common the value being transferred to it. In other words, a coin cannot be received twice by the purse. There are no counterexamples without this constraint, because *Val* refines *AbIgnore* but occurs after *Req* that actually refines *AbTransfer*. Thus the value is already considered transferred in the Abstract world. However, adding the constraint implies to also add a similar constraint in the *StartTo* operation.
- In the same way, each coin must correspond to at most one lost transaction. There are no counterexamples without this constraint, because *Abort* refines *AbIgnore*, that is the coin is already considered lost, at least thanks to the *chosenLost* prophecy variable.

Remark 1. It is worth noting that the introduction of coins does not require the model to be finite as would integers (because of the `sum` construct). But the drawback is that arbitrary infinite values may be defined, so a purse may have an arbitrary, even infinitely, high value...

Remark 2. Using coins has another interesting effect, namely in the *Abstract/Between* refinement relation : given a *Between* world and an *Abstract* world refining it, it is worth noting that the *chosenLost* set of ambiguous transaction details chosen lost used to build the *Abstract* set is uniquely known. Indeed, thanks to the constraint preventing a coin to belong 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, because for the former, the “to” purse has to have logged the transaction, whereas for the latter, the “to” purse has to be still in *epv*, 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 a *maybeLost*, but not chosen lost : then, it will be accounted into the *balance* of the “from” purse of this transaction
- the coin is a *maybeLost*, but 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))
}

```

This relational definition does not seem clear. Actually it is quite simple, from left to right : take the names, match them to `a.abAuthPurse` get the abstract authentic purses, then get their *lost* coins. `~value` represents the “revolved” *value* relation : it matches a coin to any transaction details having this coin in its value. Then, starting from the *lost* coins of the abstract authentic purses, get their transaction details but only those *maybe lost*.

To put it in a nutshell, using coins instead of integers allows to better *track* the amounts through operations.

3.3 Clear codes

Recall that a clear code is meant to *represent* a set of transaction details, as if it was computed through a *hash function*. Naively we could define a signature to represent this hash function and the clear codes :

```

sig CLEAR {imageRecip : set PayDetails}
fact clearDef {
  no disj c1, c2 : CLEAR | c1.imageRecip = c2.imageRecip
}
sig exceptionLogClear extends MESSAGE {
  name : ConPurse,
  clear : CLEAR
}

```

Actually, the `imageRecip` is the *reciprocal relation* to the hash function. The constraint ensures the functionality of the hash function ; its injectivity is ensured by the *Alloy* relational calculus itself : indeed, a clear code is mapped to exactly one set of transaction details, the one defined by its join with `imageRecip`.

Then, the `exceptionLogClear` message is defined, carrying the [name of the] clearing purse and the clear code.

But there is an easier way to represent clear codes. Indeed, we could also consider that the *exceptionLogClear* message *is itself* the clear code. Although this would not make sense in *Z* (as the message is, then, simply a record), it does in *Alloy*, since a message is an atom. Then, it could be redefined as follows :

```

sig exceptionLogClear extends MESSAGE {
  name : ConPurse,
  pds : set PayDetails
}

```

Then, the `pds` relation represents the reciprocal hash function.

It is worth noting that, when a set of transaction details is quantified over by the *Z* specification in order to compute its clear code, it is wise to quantify over the clear code instead. Then, the use of the reciprocal injection allows finding back the set of transaction details. Nevertheless, it is true that in practice, this reciprocal function is not calculable (as we start from a hash function). But for the needs of the model, we may assume its existence, regardless of how to actually construct it.

3.4 Existential quantification and constraints

The backwards (resp. forwards) simulation proofs require to show that for any *Between* (resp. *Concrete*) operation and *Abstract* post-state (resp. *Between* pre-state), there exists an *Abstract* pre-state (resp. a *Between* post-state) such that the *Abstract* (resp. *Between*) operation holds.

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 an object with the right field values may be *constructed*. But in *Alloy*, existence is the *actual* existence of the corresponding atoms in the model. That is why, in the *Abstract/Between* refinement, if we tried 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 *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, and 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) {...}

```

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': ConWorld, a, a' : AbWorld, cl' : set PayDetails | {
    Between (b)
    Between (b')
    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.

3.5 The identity of objects

The immediate question yielded by the issue of existential quantification is : can an object be defined only through its properties ?

First, the notion of a *property* has to be made more clear. Indeed, it is important to understand that *signatures do not define records*. That is why, for instance, the following assertion fails :

```

pred AbIgnore (w, w' : AbWorld) {
  w'.abAuthPurse = w.abAuthPurse
}
assert AbIgnoreIsIdent {
  all w, w' : AbWorld | AbIgnore (w, w') implies w = w'
}

```

This assertion fails : it gives a counterexample with two different abstract worlds. Indeed, abstract worlds are not records, but *atoms*, and they are simply related to the same objects by the *abAuthPurse* relation. This case raises the issue of the *identity* of objects in Alloy compared to the *Z* idiom.

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 :

```

fact canonAbPurse {
  no disj p, p' : AbWorld {
    p.balance = p'.balance
  }
}

```

```

    p.lost = p'.lost
  }
}
fact canonAbstract {
  no disj w, w' : AbWorld {
    w'.abAuthPurse = w.abAuthPurse
  }
}

```

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

But actually, the problem is deeper. Indeed, in *Z*, an abstract purse is only a *record* with two fields, *balance* and *lost*. So, when two abstract purses have the same *balance* values and the same *lost* values, then it would be impossible to distinguish them if we only considered that an *AbWorld* could be a simple set of purses. That is why names were intended : to be able to define several distinct purses with the same properties.

In Alloy, there is no such notion of field, and two distinct objects may be related to the same values by the relations. In fact, we can get rid of names, confusing them with the purses they represent. In some way, whereas *Z* represents records by default and needs names to make them individual objects, it is the converse in Alloy, which represents objects by default and needs canonicalization to pretend they are records.

Moreover, in *Z*, a purse in two different *states* may be represented by different records in different abstract worlds but still associated to the same name. Then, confusing a name with a purse leads to the fact that the *balance* of a purse not only depends on the purse, but also on the abstract world which represents not only the set of authentic purses at a given time, but also the state in which those purses are, along with their values. So, abstract purses and worlds can be redefined as follows :

```

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

```

An abstract world is a state which is mapped to the set of its authentic purses through the *abAuthPurse* relation. There are no more names. On the other hand, *balance* and *lost* are now *ternary* relations, the tuples of which consist in an abstract purse, coins, and the abstract world representing the state where we can consider that the coin belongs to the purse for the relevant relation. Then, the *AbTransfer* operation could be defined as follows :

```

sig TransferDetails {
  from, to : AbPurse,
  details : set Coin
}
pred XiAbPurse (w, w' : AbWorld, p : AbPurse) {

```

```

  p <: balance.w' = p <: balance.w
  p <: lost.w' = p <: lost.w
}
pred AbTransferOkay (w, w' : AbWorld) {
  some td : TransferDetails {
    td.from + td.to in w.abAuthPurse
    w'.abAuthPurse = w.abAuthPurse -- 1
    XiAbPurse (w, w', w.abAuthPurse - from - to) -- 2
    td.value in td.from.balance.w -- 3
    td.from.balance.w' = td.from.balance.w - td.value
    td.to.balance.w' = td.to.balance.w' + td.value
  }
}

```

Then, formula 1 would not mean that the purses do not change their *balance* or *lost*. It only means that the set of authentic purses does not change, regardless of their “properties”. To ensure that the purses other than the “from” and the “to” purse won’t change their stores during the transaction, formula 2 is required. It uses the predicate *XiAbPurse* defined above, which claims that the binary relations *balance.w'* and *balance.w* restricted to the authentic purses are the same : the authentic purses except “from” and “to” are matched to the same coins. Then, constraint 3 ensures that the “from” purse has sufficient funds to make the transaction.

The only problem is that, once names disappeared, there seems to be no more link between abstract purses and concrete purses. But thanks to the signature mechanism of *Alloy*, this problem may be solved by defining a *Purse* signature and *AbPurse* and *ConPurse* as *subsets* of it :

```

sig Purse {}
sig AbPurse in Purse {
  balance, lost : Coin -> AbWorld
}
sig ConPurse in Purse {
  balance : Coin -> ConWorld,
  payDetails : PayDetails -> ConWorld,
  exLog : PayDetails -> ConWorld,
  status : STATUS -> ConWorld,
  nextSeqNo : SEQNO -> ConWorld
}

```

Note that the same method of getting rid of names has been used for the *Concrete* purse. We can see that the *name* relation disappeared.

It is important to not define those signatures as *extending Purse*. Indeed, in that case we would have that the sets of abstract purses and concrete purses are disjoint, then again without names there would be no relation between them, and finally defining a *Purse* “mother signature” would be useless. On the contrary, if *AbPurse* and *ConPurse* are defined as subsets of *Purse*, then they may meet, and actually the abstraction relation will include this particular clause, where *a* is an *Abstract* world and *b* is a *Between* world :

```

b.conAuthPurse = a.abAuthPurse

```

This clause actually states that the abstract purses are refined by themselves. Actually, what is interesting in the abstraction relation is not the purses themselves, but their “properties”, that is the way they appear in *Abstract* and in *Between* relations.

However, not all the objects of the *Mondex* case study can be treated this way. There are still “true” records, for instance *TransferDetails* and *PayDetails* which represent respectively abstract and concrete transaction details.

```
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
  }
}
```

The canonicalization property is necessary because the *StartTo* and the *StartFrom* are independent : during those operations, the “from” and the “to” purses have to define their *pdAuth*, the data relevant to the transaction. It is necessary that the *pdAuth* of the “from” and the *pdAuth* of the “to” be equal, as they are intended to be carried through the *ether* by *req*, *val*, *ack* messages, and maybe logged into the *archive*.

4 Summary of the final model layout

This section describes the layout of the final *Mondex* specification in *Alloy*. It is divided into several modules.

It is available on my website : <http://www.eleves.ens.fr/home/ramanana/work/mondex/> .
So is the former model, which is not detailed here.

4.1 The Common module

This module defines the signatures that are going to be used by every module, namely the purses (without properties) and the coins. It also defines the *TransferDetails*, because this signature is *extended* by *PayDetails* in the Concrete.

```
module common
sig Purse {}
sig Coin {}
sig TransferDetails {
  from, to : Purse,
  value : set Coin
}
```

4.2 The Abstract module

This module imports the Common module.

It defines the abstract purses and the abstract world. It only structurally defines them, without constraining them. Constraints, which are actually only coin sharing constraints, are defined as predicates rather than facts, to be used later, in assertions themselves.

```
module a
sig AbPurse in Purse {
  abBalance, abLost : Coin -> AbWorld
}
sig AbWorld {
  abAuthPurse : set AbPurse
}
pred Abstract (a : AbWorld) {
  no a.abAuthPurse.balance & a.abAuthPurse.lost
  a.abAuthPurse <: (balance + lost) : lone AbPurse -> Coin
}
```

This module also defines the abstract operations (*AbIgnore*, *AbTransfer*).

It also defines the sanity-check predicate simulations for the *Abstract* model, and also the assertions for security properties.

It also defines assertions to show the totality of the *Abstract* properties, and also to show that if the pre-state of an *Abstract* operation is an *Abstract* world, then so is the post-state.

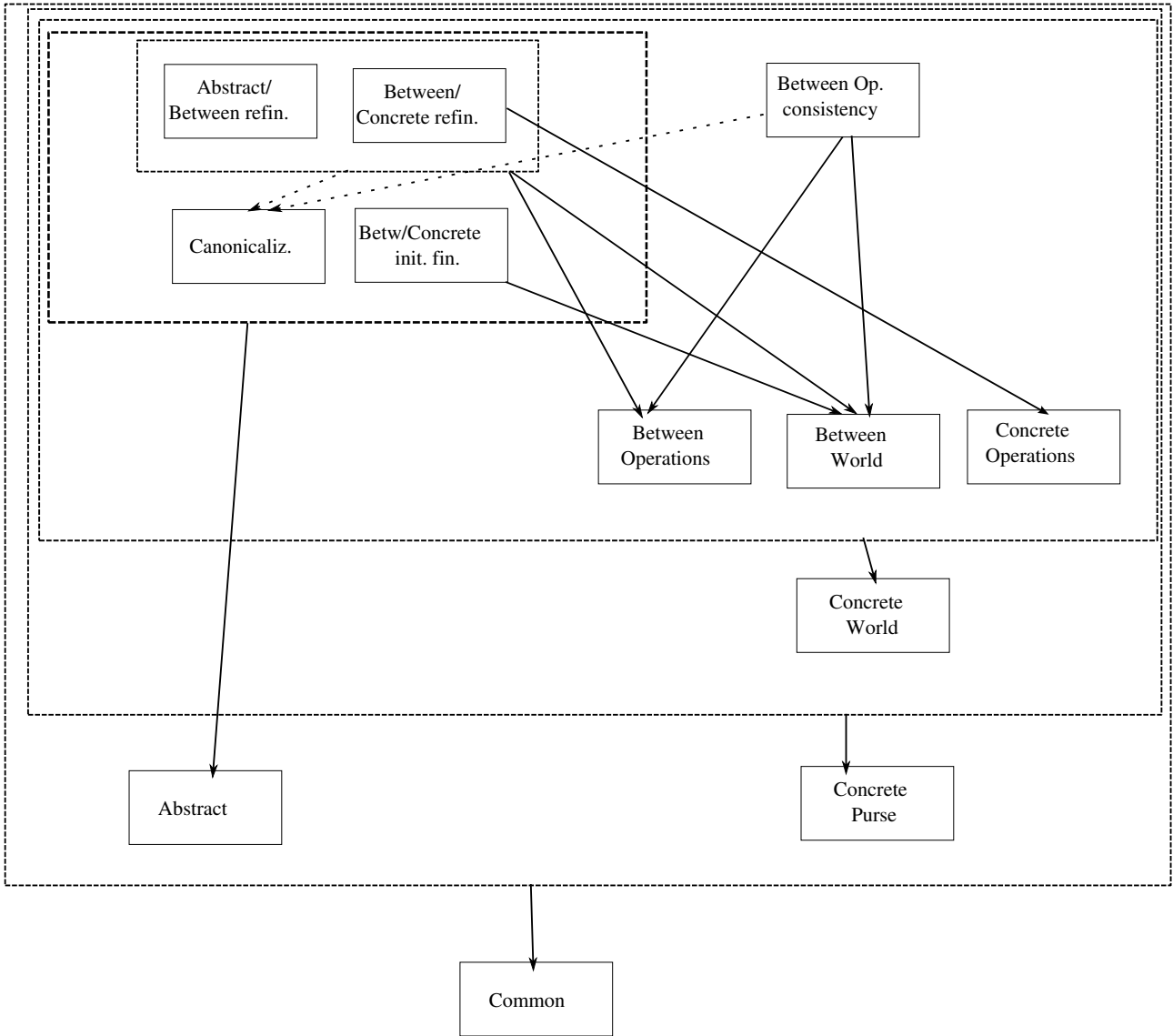


Figure 12: *Module dependencies of the final model*

The arrows correspond to the “imports” relationship. It is worth noting that Between operations do not import Between world : they are defined only structurally, independently on the constraints.

Modules are grouped to improve graph lisibility.

4.3 The Concrete Purse module

This module imports the Common module. It also imports `util/ordering` to define the sequence numbers along with their ordering relation.

It defines the concrete purses, hence also the sequence numbers, the statuses (confusing the two “idle” statuses *eaFrom* and *eaTo*) and the transaction details, which have to extend `TransferDetails` (defined in the Common module). It only structurally defines them, without constraining them. Constraints are given in further modules, within worlds.

But the Concrete world is not defined yet. Though, a state must be defined to allow the purse having different values for its properties depending on the state. To show that those values are independent on the *global* properties on a Concrete world (*ether*, *archive*), we define here a `ConState` signature (without properties) which will be extended by the definition of the concrete world later on.

```
module c

sig SEQNO {}
open util/ordering [SEQNO]

abstract sig STATUS {}
one sig eaFrom, epr, epv, epa extends STATUS {}

sig PayDetails extends TransferDetails {
  fromSeqNo, toSeqNo : SEQNO
}

sig ConState {}

sig ConPurse {
  balance : Coin -> ConState,
  pdAuth : PayDetails -> ConState,
  exLog : PayDetails -> ConState,
  nextSeqNo : SEQNO -> ConState,
  status : STATUS -> ConState
}
```

This module also defines concrete operations at the purse level, which will be used in both Concrete and Between promotions to the world level. To this purpose, this module defines messages : relevant to the transfer protocol (*startFrom*, *startTo*, *req*, *val*, *ack*) or the *exLog* clearing protocol (*readExceptionLog*, *exceptionLogResult*, *exceptionLogClear*).

```
abstract sig MESSAGE {}

abstract sig CounterPartyDetails extends MESSAGE {
  counterParty : ConPurse,
  value : set Coin,
  next : SEQNO
}
sig startFrom, startTo extends CounterPartyDetails {}
```

```

sig req extends MESSAGE {details : PayDetails}
sig val extends MESSAGE {details : PayDetails}
sig ack extends MESSAGE {details : PayDetails}

one sig readExceptionLog extends MESSAGE {}
sig exceptionLogResult extends MESSAGE {
  name : ConPurse,
  details : PayDetails
}
sig exceptionLogClear extends MESSAGE {
  name : ConPurse,
  pds : set PayDetails
}

```

4.4 The Concrete World module

This module imports the Common and the Concrete Purse modules.

It defines the Concrete World, and its “constraints” as predicates. The concrete world signature extends `ConState` defined as the state signature for the concrete purses. The “constraints” include the original constraints defined in the *Z* specification, and also multiplicity constraints for the relations involving concrete purses. Indeed, one has to explicitly constraint the concrete purse to have exactly one sequence number, one status, and one pending transaction (*pdAuth* — except if the purse is idle) for each concrete world where it is authentic. However, they do not include coin sharing constraints. The latter constraints are considered *Between* constraints.

```

module cw
open common
open c

sig ConWorld extends ConState {
  conAuthPurse : set ConPurse,
  ether : set MESSAGE,
  archive : ConPurse -> PayDetails
}

pred Concrete (c : ConWorld) {...}

```

This module also defines a simulation predicate to show that there may exist a concrete world satisfying `Concrete`.

4.5 The Between World module

This module (*b*) imports the Common, the Concrete Purse and the Concrete World modules.

It defines the Between World constraints, as predicates. To this purpose, it also defines auxiliary functions such as *definitelyLost*, *maybeLost*.

In fact, as *Between* and *Concrete* world have the same structure, a *Between* world is not represented by a signature ; such an object is only a *Concrete* world verifying the constraints of the *Between* defined as predicates.

4.6 The Between World operations module and the Concrete World operations module

These modules (*bop*, *cop*) import the Common, the Concrete Purse and the Concrete World modules.

bop defines the operations at the Between World level. However, it does not import the Between World module, to show that operation definitions are independent on the constraints of *Between* : they are defined only *structurally*. These operations work with a complete *ether*, losing no messages.

cop defines the operations at the Concrete World level. These operations work with a lossy *ether*. It additionally defines an assertion to show the totality of the *Concrete* operations (for any *Concrete* pre-world and *Concrete* operation, there exists a *Concrete* post-world).

4.7 The Between and Concrete initialization and finalization module

This module (*bcif*) imports the Common module, the Concrete Purse module, the Concrete World module, the Between World module, and also the Abstract World module.

It defines the initialization and finalization predicates. To the latter purpose, it needs the Abstract World module because of the “global world”, which is simply a specific Abstract World.

It also defines simulation predicates to show that there exist initial states.

4.8 The Between operation consistency module

This module (*bopc*) imports the Common, the Concrete Purse, the Concrete World, the Between World and the Between World operations modules.

It defines an assertion to show the totality of the *Between* operations (for any *Concrete* pre-world and *Concrete* operation, there exists a *Concrete* post-world).

It also defines assertions to show that for any *Between* pre-state and any *Between* operation, the post-state is a *Between* world.

It also defines assertions to show that the operations that first abort (*StartFrom*, *StartTo*, *readExceptionLog*, *clearExceptionLog*) may be decomposed into *Abort* followed by an elementary operation. Then, further operations about those theorems only involve more “atomic” operations, that is assume that *Abort* has already been tackled.

4.9 The Abstract/Between refinement module

This module imports the Common, the Concrete Purse, the Concrete World, the Between World, the Between World operations and the Abstract World modules.

It defines the *Abstract/Between* abstraction relation, splitting it into two parts : a “construction” part (defining the abstract world), and a “condition” part (the applicability preconditions for the abstraction relation).

```
module rab
open common
```

```

open c
open cw
open b
open bop
open a

pred RabCl_constr (a : AbWorld, b : ConWorld, cl : set PayDetails) {
  a.abAuthPurse = b.conAuthPurse
  abLost.a = ((b.conAuthPurse->(definitelyLost(b) + cl)) & ~from).value
  abBalance.a = ((b.conAuthPurse->(maybeLost(b) - cl)) & ~to).value + balance.b
}
pred RabCl_cond (b : ConWorld, cl : set PayDetails) {
  cl in maybeLost (b)
}
pred RabCl (a : AbWorld, b : ConWorld, cl : set PayDetails) {
  RabCl_cond (b, cl)
  RabCl_constr (a, b, cl)
}

```

Starting from those predicates, this module defines an assertion showing that any “abstract object” defined through this relation is actually an *Abstract* world (verifying namely the abstract coin sharing constraints) :

```

assert Rab_ex {
  all b : ConWorld, cl : set PayDetails, a : AbWorld | {
    RabCl (a, b, cl)
    Between (b)
  } implies Abstract (a)
}

```

This module also defines the variant *Abstract/Between* relation using coins. Then, it defines two assertions to show that the definition is equivalent to the regular abstraction relation.

Then, the module defines the refinement assertions for each *Between* operation, using the variant relation.

4.10 The *Between/Concrete* refinement module

This module imports the Common, the Concrete Purse, the Concrete World, the Between World, the Concrete World operations and the Between World operations modules.

It defines the *Between/Concrete* abstraction relation, splitting it into two parts : a “generic” part (defining the relation as the lossiness of the Concrete ether), and a “constructive” part (actually constructing a *Between* post-state given the *Concrete* post-state, the ether of the pre-*Between* and the output message of the operation.

```

module rbc
open common
open c
open cw

```

```

open b
open cop
open bop
open bcif
open a

pred Rbc (b, c : ConWorld) {
  b.conAuthPurse = c.conAuthPurse
  XiConPurse (b, c, b.conAuthPurse)
  c.ether in b.ether
  b.archive = c.archive
}
pred Rbc_constr (b', c' : ConWorld, eth : set MESSAGE, m_out : MESSAGE) {
  b'.conAuthPurse = c'.conAuthPurse
  XiConPurse (b', c', b'.conAuthPurse)
  b'.ether = eth + m_out
  b'.archive = c'.archive
}

```

Then, this module defines assertions to check the refinement relation for each operation.

It is not useful to check whether the structure computed by `Rbc_constr` is actually a *Between* world. Indeed, thanks to the *Between World* consistency module, we already know that if the pre-state of a *Between* operation is a *Between* world, then so is the post-state.

4.11 The Canonicalization module

Both *Abstract/Between* and *Between/Concrete* refinements, and also the *Between* world consistency, may import the *Canonicalization* module (*canon*). This module defines axioms to canonicalize *abstract* and *concrete* worlds in order to try to reduce the search space when checking theorems. This importation is “optional” as it can be disabled by commenting the corresponding `open` statement, without any structural effect (as the module only defines facts).

```
fact ax_AbWorld_canon {
  no disj a1, a2 : AbWorld {
    a1.abAuthPurse = a2.abAuthPurse
    XiAbPurse (a1, a2, a1.abAuthPurse)
  } }

fact ax_ConWorld_canon {
  no disj c1, c2 : ConWorld {
    c1.conAuthPurse = c2.conAuthPurse
    XiConPurse (c1, c2, c1.conAuthPurse)
    c1.ether = c2.ether
    c1.archive = c2.archive
  } }
```

5 Results

5.1 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.

5.1.1 Abort proof schema

Mostly, the *Alloy* method allows to directly check the specification without going through intermediate lemmas. But for the *Abort/AbIgnore* refinement, as a check on a scope of 8 did not terminate after 2 days of computation, it was necessary to tackle a lemma. So, we had to go into the details of the proof for this theorem.

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 : if it is *eaFrom*, then 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 in *epv*. 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 in *epa*. 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 I presented it at the workshop in May 2006.

5.1.2 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 on the former specification 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 is in *epv* (which is a “to” state)
- the other if the purse holds a *pdAuth* indicating that it is actually the *to* purse, but is in *epa* (which is a “from” state).

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 the *pdAuth*.

Adding the corresponding constraints 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].

5.1.3 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 : *StartFrom*, *StartTo*, *ReadExceptionLog*, *ClearExceptionLog*) 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 ! Indeed, whereas *StartTo* and *readExceptionLog* output specific messages (*req* and *exceptionLogResult*), *Abort* outputs a generic message called \perp .

So there is necessary a bug in the Z specification. Actually, it can be found without a further check with the *Alloy Analyzer*. The Z proof argues that those operations are defined through a *framing schema* Φ , that is through a definition of the form :

$$\exists \Delta ConPurse \bullet \Phi \wedge (AbortPurseOkay; StartFromPurseEafromOkay)$$

where $;$ is the *composition* operation.

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

$$(\exists \Delta ConPurse \bullet \Phi \wedge AbortPurseOkay); (\exists \Delta ConPurse \bullet \Phi \wedge StartFromPurseEafromOkay)$$

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

$\frac{ConWorld}{conAuthPurse : NAME \leftrightarrow ConPurse}$
$\frac{\Phi}{\begin{array}{l} \Delta ConWorld \\ \Delta ConPurse \\ n? : NAME \end{array}}$ <hr style="width: 80%; margin-left: 0;"/> $\begin{array}{l} n? \in \text{dom } conAuthPurse \\ conAuthPurse \ n? = \theta ConPurse \\ conAuthPurse' = conAuthPurse \oplus \{n? \mapsto \theta ConPurse'\} \end{array}$

But, even though the lemma itself might be true, actually the *process* is wrong because Φ is *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 the two “elementary operations” output different messages, so that it is impossible to compose them.

The 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*. The lemma would 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.

5.2 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 :

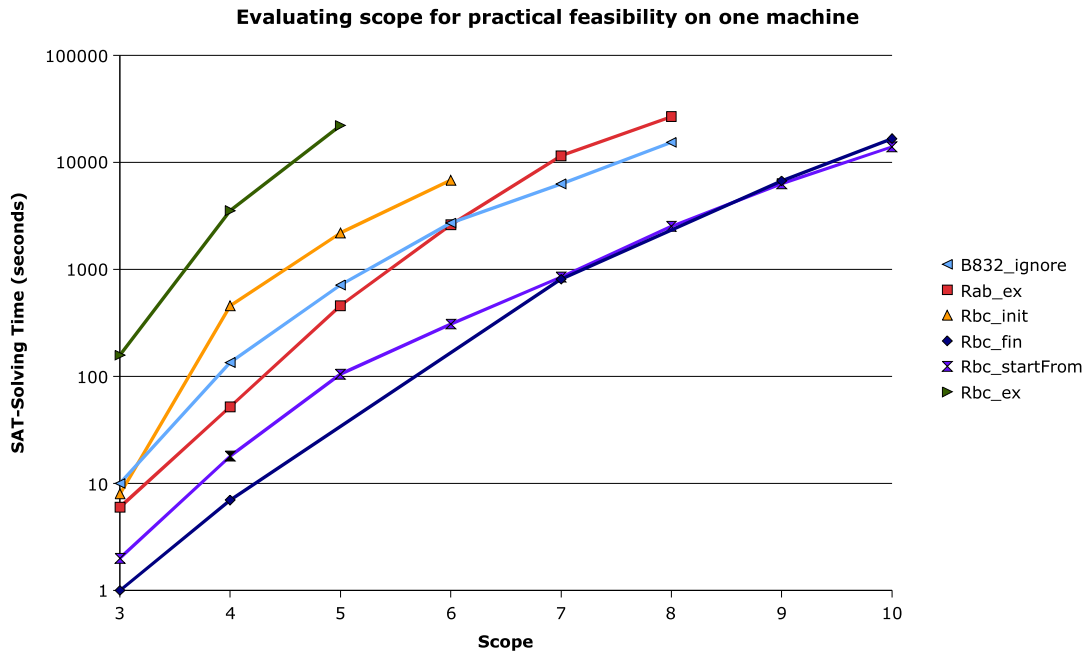


Figure 13: *Time exponentially increases with the scope*

This graph was obtained with the former model.

- 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 unauthentic purses.
- Obviously, 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 former 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 former model. It is worth noting that in that case, the times are sensitively

Scope →	5	6	7	8	9	10	Notes
↓Theorem							
Abstract Security Properties						1:47	berkmin
with AbTransferOkay only				0:09	0:17	0:21	berkmin
Totality of Abstract Operations						0:24	minisat
Abstract constr. & operations							
AbIgnore						0:50	minisat
AbTransfer						8:41	minisat
Totality of Between operations						3:47	minisat
Between constraints & operations							
Ignore	5:42:14						minisat
	5:43						minisat, but 2 ConState
	4:13			1:40:22			minisat, but 2 ConState, canon.
Increase				20:35:33			minisat, but 2 ConState, canon.
Abort		20:08:46					minisat
	4:22:24						minisat, but 2 ConWorld
				19:10:12			siege_v4, but 2 ConWorld, canon.
StartFrom				13:16:13			siege_v4, but 2 ConWorld, canon.
StartTo				5:10:13			siege_v4, but 2 ConWorld, canon.
Req				20:26:45			siege_v4, but 2 ConWorld, canon.
Val				8:38:24			siege_v4, but 2 ConWorld, canon.
Ack				14:42:13			siege_v4, but 2 ConWorld, canon.
ReadExceptionLog				28:13:00			siege_v4, but 2 ConWorld, canon.
ClearExceptionLog				6:49:54			siege_v4, but 2 ConWorld, canon.
AuthorizeExLogClear				26:36:53			siege_v4, but 2 ConWorld, canon.
Archive				57:49			siege_v4, but 2 ConWorld, canon.
Decomp. of operations first aborting							
StartFrom						2:36	minisat
StartTo						1:15	minisat
ReadExceptionLog						1:51	minisat
ClearExceptionLog						1:28	minisat

Figure 14: *Scopes and times for the final model*

longer for the final model than for the former model. On the one hand, this is due to the constraints, which were too strong in the former 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.

In some way, the time of computation shows how important theorems are, but not always how difficult they are. Indeed, some obvious-looking theorems such as the *Between/Concrete* initialization refinement take quite a long time to be checked.

5.3 Limits to the use of the *Alloy Analyzer*

Because *Alloy* is based on first-order, even despite transitive closures, finiteness properties have to be dropped. But we saw that, as regards the *Mondex* case study, finiteness properties may

Scope →	5	6	7	8	9	10	Notes
↓ Theorem							
Abstract/Between ref. and coins							
chosenLost can be found by coins						52:41	siege_v4
coins can be found if chosenLost exists						57:34	siege_v4
	2:55	22:30					minisat
Abstraction relation sound	1:45	9:19	9:48	31:28	10:25:45		minisat
Abstract/Between refinement							
Ignore	0:46	5:15	27:40	1:47:03			minisat, with chosenLost
		17:09					minisat, but 2 AbWorld, 2 ConState with coins
		7:20		38:14			siege_v4, but 2 AbWorld, 2 Constate, with coins, canonicalized
Increase		14:01		58:23			idem
		42:12					idem but not canonicalized
Abort						17:04:03	siege_v4
		36:07					siege_v4, but 2 AbWorld, 2 ConState
		16:00		1:20:34			idem canonicalized
StartFrom				48:05			idem as above
StartTo				1:01:23			idem
Req				42:26			idem
Val				25:35			idem
Ack				40:44			idem
ReadExceptionLog				38:06			idem
ClearExceptionLog				35:01			idem
AuthorizeExLogClear				18:34			idem
Archive				29:01			idem
Between/Concrete refinement							
Initialization						2:46:47	siege_v4, but 2 ConState, 1 AbWorld
Finalization						14:01	idem
Ignore						1:26	minisat
Increase						2:32	minisat
Abort						2:11	minisat
StartFrom						3:27:36	siege_v4, but 5 ConState
StartTo						1:43:28	siege_v4, but 5 ConState
Req						2:18	berkmin
Val						2:06	berkmin
Ack						1:59	berkmin
ReadExceptionLog						2:44	berkmin
ClearExceptionLog						2:46	berkmin
AuthorizeExLogClear						1:56	berkmin
Archive						1:53	berkmin

Figure 15: *Scopes and times for the final model (continued)*

be shown *indirectly* by showing, for instance, that the symmetric difference between a post-set and a pre-set is a bounded finite set. As for the finiteness of abstract (or concrete) authentic purses, or This is true by the definition of 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.

A first attempt could be to try to increase scopes by improving ambient conditions (machines, etc.), or even by using the *Kodkod* tool 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 such as the quantification over a set of transaction details to compute its clear code, can be turned into first-order, for instance by considering the clear code itself ; moreover, transitive closures are not useful as operations are considered individually : there are no theorems about sequences of operations.

6 Using first-order theorem provers with Alloy

As the *Mondex* case study (minus finiteness properties) can be entirely written in first-order logic, even without transitive closures, it could be interesting to really *prove* theorems using an automated first-order theorem prover, instead of simply checking it with a model-finder.

There are two possible approaches :

- the usual approach, where Alloy atoms are directly mapped to FOL atoms
- the “lifted” approach, where FOL atoms are Alloy *relations*. This requires axiomatizing a first-order relational theory

6.1 The usual approach : Alloy atoms as FOL atoms

A first approach is to consider *Alloy* atoms as atoms of the first-order theory. Then, relations are expressible using predicates telling whether their argument is an element of the relation.

I wrote an Alloy parser, in Objective Caml [oca]. It parses part of the Alloy syntax (but does not handle transitive closures), and outputs a first-order theory in either TPTP format, a format developed to appreciate, through competitions, the capabilities of theorem provers such as Vampire, E [e] or Theo, or DFG format, for use with theorem provers such as SPASS [spa]. It is available on my website, <http://www.eleves.ens.fr/home/ramanana/work/mondex/>.

6.1.1 Principle

Signatures and relations are defined as predicate symbols. For instance, for the *Abstract* world, we obtain the following predicate symbols :

- *Coin*, *Purse*; *AbPurse*; *AbWorld* unary predicates,
- *balance*, *lost*, *abAuthPurse* binary predicates,

These predicate symbols are defined along with constraints over them, to model the signature inclusion and extension mechanisms, and also the “typing” constraints for relations :

- $\forall x : \neg (Coin(x) \wedge Purse(x)) \wedge \neg ((Coin(x) \vee Purse(x)) \wedge AbWorld(x))$: signatures *Coin*, *Purse*, *AbWorld* are disjoint.
- $\forall x : AbPurse(x) \Rightarrow Purse(x)$: signature *AbPurse* included in *Purse*.
- $\forall x, y : balance(x, y) \Rightarrow (AbPurse(x) \wedge Coin(y))$: relation *balance* associates only abstract purses with coins. Similar constraints are generated for *lost* and *abAuthPurse*.

Then, each additional fact is added as an axiom of the FOL theory, whereas each assertion is added as a conjecture.

Relations are mapped to the predicates representing them as follows, dropping transitive closures (impossible to express in FOL) :

- $[S](\vec{x}) \equiv S(\vec{x})$ if S is a signature or a relation

- $[y](x) \equiv x = y$ if y is a variable (necessarily representing an atom)
- $[U + V](\vec{x}) \equiv [U](\vec{x}) \vee [V](\vec{x})$ (union)
- $[U \& V](\vec{x}) \equiv [U](\vec{x}) \wedge [V](\vec{x})$ (intersection)
- $[U \rightarrow V](\vec{x}, \vec{y}) \equiv [U](\vec{x}) \wedge [V](\vec{y})$ (Cartesian product) if the tuples have the right arities
- $[U.V](\vec{x}, \vec{y}) \equiv \exists t : [U](\vec{x}, t) \wedge [V](t, \vec{y})$ (join)

Logical formulae are taken quite “as is”, except that there is no skolemisation : quantifications must be first-order.

But Alloy auxiliary predicates and functions have to be inlined, as they take relations, not necessarily atoms, as arguments. This makes the formulae dramatically grow.

6.1.2 Simplifications

Inlinings make the formulae grow. But in particular, each join operator introduces an existential quantifier : the target formula may have up to 22000 existential quantifiers (*Abstract/Between* refinement) !

Fortunately most of them generate formulae of the form $\exists t : t = y \wedge P(\vec{z})$. In that case, the obvious simplification to $P(\vec{z} [t \leftarrow y])$ allows deleting, roughly speaking, half of those quantifiers. In the same way, such a simplification applies to formulae of the form $\forall t : t = y \Rightarrow P(\vec{z})$.

It is possible to increase the simplification level through introducing functions, that is taking into account the fact that some relations are defined functional. Only binary functional relations are implemented. Then, *terms* appear in the FOL-formula. Thanks to this simplification, roughly speaking the two thirds of the existential quantifiers disappear.

Thanks to those simplifications, some theorems about the *Abstract* model, such as the trivial *AbIgnore_inv* (stating that the post-state of an *AbIgnore* operation is an *Abstract* world) or the totality of transfer operations, can be proved with Vampire or SPASS, in a couple of seconds. Security properties also get proved, but need variable amount of time, up to 20 minutes. However, splitting the latter theorem into *Okay* (achieved) and *Lost* transactions yields two theorems provable in a couple of seconds each.

Thus comes up the idea of trying to break the formulae in several parts. The following cases would be breakable :

- $P \wedge Q$ is breakable into the two parts P and Q.
- $\forall \vec{x} : P(\vec{x}) \wedge Q(\vec{x})$ is breakable into the two parts $\forall \vec{x} : P(\vec{x})$ and $\forall \vec{x} : Q(\vec{x})$.
- If \vec{y} and \vec{x} are independent on each other, then $\exists \vec{x} : P(\vec{x}) \wedge Q(\vec{y})$ is breakable into the two parts $\exists \vec{x} : P(\vec{x})$ and $Q(\vec{y})$.

As it is possible to put quantifiers as close as possible to the variables they quantify over (in other words, on the contrary, extract subformulae under quantifiers that are independent on the quantified variables), the idea is to break a formula into several parts of the form $(\forall \forall \exists \wedge) *$

But this form requires to use the distributivity of \vee through \wedge . Subsequently, contrary to what is expected of a simplification algorithm, the formula explodes, despite adding an algorithm of subsumption detection. Worse : the “simplification” process does not terminate in a reasonable amount of time. So this “simplification” has to be abandoned.

More generally, it is necessary to find another FOL interpretation of *Alloy* which could use less variables and less quantifiers.

6.2 The “lifted” way : Alloy relations as FOL atoms

To prevent formulae from growing, the idea could be to axiomatize a first-order relational theory where the FOL atoms would be *Alloy* relations themselves :

- Relations are of a given arity : they may only contain tuples of certain arity
- Given a specification in *Alloy*, it is possible to compute an upper bound over the arity of the relations considered in the specification

From those main ideas, we can axiomatize the relation theory in a *finite* number of axioms given the maximal arity of our relations. The basis is a binary predicate symbol representing \in , meant to be a restriction of \subseteq to singletons.

“General” set theory is defined as follows :

- $\forall r : \text{singl}(r) \equiv \forall x : x \in r \Leftrightarrow x = r$: a relation is defined to be a singleton if and only if the only relation that belongs (\in) to it is exactly itself.
- $\forall x, r : x \in r \Rightarrow \text{singl}(x)$: only singletons can belong to relations.
- $\forall x : x \notin \text{none}$: the empty relation has no elements.
- $\forall a, b : a \subseteq b \equiv \forall x : x \in a \Rightarrow x \in b$: definition of inclusion
- $\forall a, b : (a \subseteq b \wedge b \subseteq a) \Leftrightarrow a = b$: inclusion is reflexive and antisymmetric (extensionality)

Then, relations are partitioned into almost disjoint classes that are the *arities* :

- $\bigwedge_j \text{arity}_j(\text{none})$: the empty relation is said to be of any arity (hence the *almost disjoint* arity classes)
- $\forall x : \left(\text{arity}_i(x) \wedge \bigvee_{j < i} \text{arity}_j(x) \right) \Rightarrow x = \text{none}$: a relation of two distinct arities is empty
- $\forall r : \text{arity}_i(r) \Leftrightarrow (\forall s : s \subseteq r \Rightarrow \text{arity}_i(s))$: any subset of a relation has the same arity as the relation

Then, we define tuples as a predicate symbol to model the decomposition of a singleton of arity i into a tuple of i singletons of arity 1. For each arity, there are 5 axioms to define the tuple predicate :

- $\forall x : (singl(x) \wedge arity_i(x)) \Rightarrow \exists (y_j)_{j < i} : tuple_i(x, \vec{y})$: any singleton of arity i is decomposable into a tuple of i elements.
- $\forall (y_j)_{j < i} : \bigwedge_{j < i} (singl(y_j) \wedge arity_1(y_j)) \Rightarrow \exists x : tuple_i(x, \vec{y})$: for any i singletons of arity 1, their tuple may be constructed.
- $\forall x, (y_j)_{j < i} : tuple_i(x, \vec{y}) \Rightarrow \left(singl(x) \wedge arity_i(x) \wedge \bigwedge_{j < i} (singl(y_j) \wedge arity_1(y_j)) \right)$: for any decomposition, the decomposed tuple is a singleton of arity i , and each of the components is a singleton of arity 1.
- $\forall x, (y_j)_{j < i}, (z_j)_{j < i} : (tuple_i(x, \vec{y}) \wedge tuple_i(x, \vec{z})) \Rightarrow \vec{y} = \vec{z}$: the decomposition of a tuple is unique
- $\forall x, w, (y_j)_{j < i} : (tuple_i(x, \vec{y}) \wedge tuple_i(w, \vec{y})) \Rightarrow x = w$: a tuple construction defines a unique tuple

The case $i = 1$ requires a further axiom : $\forall x, y : tuple_1(x, y) \Rightarrow x = y$ (the decomposition of a “tuple” of arity 1 is necessary itself)

Given a specification in *Alloy*, it is possible to *tag* each relational construct (join, union, etc.) by the arities of its arguments. Then, such a construct is constrained to define empty relations if the arities of its arguments do not match.

Even though this theory is enough to model the relational theory in *Alloy* (without the transitive closure), it still remains difficult to prove even basic theorems such as “the cartesian product of two singletons equals the tuple they form”. Such a proof is achieved through defining lemmas by hand. This highly jeopardizes the hope of trying to automatically prove arbitrarily complicated theorems in this theory, in particular the *Mondex* case study, despite the fact that *Mondex* relations are of arity at most 3.

6.3 Results and limits

Thanks to the first method, theorems about the *Abstract* model only have been proven : totality of the *Abstract* operations, security properties and invariance of the *Abstract* constraints through the *Abstract* operations. If the theorems about *Transfer* operations are splitted into the two parts *Okay* (achieved transaction) and *Lost*, then each theorem takes a couple of seconds to be proved with SPASS or Vampire. However, other theorem provers such as E or Theo failed : E crashed each time, by lack of memory, whereas Theo did not terminate after several days — almost one week — of computation.

Unfortunately, the second method did not even achieve to prove those theorems. This was mainly due to the fact that lemmas would have been necessary to carry on the proof. But the use of lemmas completely breaks the automation, unless those lemmas are themselves automatically generated.

Beyond the problem of lemmas, the logical expressiveness of those theorem provers is limited to FOL : it is even less powerful than the *Alloy* specification language. Even transitive closures are not available. But as the *Mondex* case study (without finiteness properties) does not need them, the FOL method could have been interesting insofar as it allowed to rigorously prove theorems beyond the limitations of scope imposed by the *Alloy Analyzer*.

7 Conclusion and future work

This internship allowed me to learn how to design a specification, and more generally to discover another aspect of formal methods, namely raising automation as a main issue. When I came at MIT in March, I had to learn Z and Alloy quite from scratch. The way of designing specifications with Alloy is very user-friendly, as the language is easily understandable and has no graphical issues such as Z's Δ , θ or Ξ : Alloy's notations are quite intuitive. But intuition can also deceive the user through the join operator, which is *not* a field dereference operator, hence all the conceptual problems between Z and Alloy about representing “records” and, more generally, the issue of the identity of objects. However, thanks to the point of view, I have been able to design a more rigorous *Mondex* model based on Alloy's “idiom” rather than Z's.

The *Alloy* formal method, based on first-order 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. Some those bugs such as the authenticity bug have also been found by other methods such as Z/Eves or KIV, but not all of them, such as the *AbIgnore/Abort* refinement proof structure. 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, 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, does not fit the *Mondex* case study. So, other formal methods have to complete the use of the *Alloy Analyzer*. But, besides *Prioni* [AKMR03] which intends to use *Alloy* specifications with the *Athena* proof assistant, which is not fully automatic, trying to handle *Alloy* models in first-order logic could be also interesting. For more general cases than *Mondex* which might use transitive closures, Tal Lev Ami's work could represent an interesting first-order logic complement to *Alloy*, as it moreover tries to handle transitive closures [LAIR⁺05]. But to be able to practically use theorem provers, it is necessary to improve the conception of automated theorem provers, which is the concern raised by competitions such as TPTP [tpt].

It could be also interesting to develop syntactic analysis of Alloy specifications, to automatize relational calculus and reasoning directly at the formula level, which would make the constraint of finite scope irrelevant. Integrating such an idea could dramatically increase the fame of Alloy, which is already very popular in the industry thanks to its automated *Alloy Analyzer*. But, with the coming *Kodkod*, an improved engine for the *Alloy Analyzer*, so still based on *model-finding* through SAT-solving, the SDG group would like to extend the capabilities of the Alloy specification language through adding scripting features. So, one could fear from the

fact that such features would become an issue for groups wishing to use the Alloy specification language with other methods than *model-finding*. However, according to the SDG members, those scripting features will keep the expressivity of the Alloy specification language invariant.

References

- [AA] The Alloy Analyzer. <http://alloy.mit.edu>.
- [Abr96a] Jean-René Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [Abr96b] Jean-René Abrial. Extending B without changing it (for developing distributed systems). In *Proceedings of the 1st Conference on the B method*, Putting into Practice methods and tools for information system design, pages 169–191. Habrias, 1996.
- [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>.
- [BY06] Michael Butler and Divakar Yadav. Applying Event-B to Mondex. Slides for the 3rd VSR-net workshop, 5 2006.
- [CC04] David Crocker and Judith Carlton. Perfect Developer: what it is and what it does. *FACS Facts: Newsletter of the BCS Formal Aspects of Computer Science special interest group*, 11 2004.
- [Com] UK Computing Research Committee. Grand Challenges in Computer Research. http://www.ukcsrc.org.uk/grand_challenges/index.cfm.
- [Coq] The Coq proof assistant. <http://coq.inria.fr>.
- [Cro06] David Crocker. Mondex Revisited with Perfect Developer. Slides for the 2nd VSR-net Workshop, 1 2006.
- [e] The E Equational Theorem Prover. <http://www4.in.tum.de/~schulz/WORK/e prover.html>.
- [FW06] Leo Freitas and Jim Woodcock. Mondex in Z/Eves. Slides for the 3rd VSR-net workshop, 5 2006.
- [GC6] Grand Challenge 6 : Dependable Systems Evolution. <http://www.fmnet.info/gc6>.
- [GH06] Chris George and Anne Haxthausen. Specification and Proof of the Mondex Electronic Purse. Slides of the 3rd VSR-net Workshop, 5 2006.
- [GHH⁺95] Chris George, Anne E. Haxthausen, Steven Hughes, Robert Milae, Soren Prehn, and Jan Storbak Pedersen. *The RAISE Development Method*. Prentice Hall, 1995.
- [Gog06] Martin Gogolla. Use OCL for Mondex. Slides for the 3rd VSR-net workshop, 5 2006.
- [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.
- [Jon90] Cliff Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.
- [Jon06] Cliff Jones. VSR Mondex Meeting. Slides of the 2nd VSR-net Workshop, 1 2006.
- [JOW06] Cliff Jones, Peter O’Hearn, and Jim Woodcock. Verified Software: A Grand Challenge. *Computer*, 39(4):93–95, 4 2006.
- [KIV] KIV, the Karlsruhe Interactive Verifier. <http://i11www.iti.uni-karlsruhe.de/~kiv>.
- [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>.
- [Mno] The Moneo electronic purse system. <http://www.moneo.net>.
- [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>.
- [oca] The Objective Caml functional programming language. <http://caml.inria.fr/ocaml>.
- [PD] Perfect Developer. <http://www.eschertech.com>.
- [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.
- [spa] SPASS: An Automated Theorem Prover for First-Order Logic with Equality. <http://spass.mpi-sb.mpg.de>.
- [Spi92] J. Michael Spivey. *The Z notation: a Reference Manual*. Prentice Hall, 2nd edition edition, 1992.
- [tpt] Thousands of Problems for Theorem Provers. <http://www.cs.miami.edu/~tptp>.
- [VSR] VSR-net: A Network for the Verified Software Repository. <http://www.fmnet.info/gc6>.

- [WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof*. Prentice Hall, 1996.
- [WK99] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12:10–13, 3 1999.
- [ZE] The Z/Eves System. <http://nexp.cs.pdx.edu/bart/omse/omse522-winter2002/nfp/sw/z-eves/z-eves.html>.

List of Figures

1	<i>Abstract (atomic) transactions : successful, lost.</i>	9
2	<i>Concrete 5-step protocol, with the statuses of the purses depending on the operations.</i>	11
3	<i>Abort operation : cases when money is lost or not.</i>	14
4	<i>Clearing process</i>	15
5	<i>Backwards refinement proof for Abstract/Between</i>	18
6	<i>Abstraction of the typical sequence of Between operations defining a transaction</i>	19
7	<i>Forwards refinement for Between/Concrete</i>	20
8	<i>Between/Concrete initialization and finalization.</i>	21
9	<i>Principle of model-finding through SAT-solving with the Alloy Analyzer</i>	36
10	<i>Counterexample to the IncreaseImpliesIgnore assertion</i>	39
11	<i>Example to the Increase predicate</i>	40
12	<i>Module dependencies of the final model</i>	56
13	<i>Time exponentially increases with the scope</i>	65
14	<i>Scopes and times for the final model</i>	66
15	<i>Scopes and times for the final model (continued)</i>	67