der Bundeswehr
Universität München

FAKULTÄT FÜR INFORMATIK

Establishing Properties of Frameworks and their Applications: A Case Study using Alloy and Object-Z

Christian Heger und Lothar Schmitz

Bericht 2009-01
Januar 2009

Werner-Heisenberg-Weg 39 • 85577 Neubiberg • Germany

Abstract

Object-oriented frameworks are a popular means for efficiently producing quality software. We conjecture that formal proofs of a framework's properties help to prove properties of applications developed using the framework. Here, we describe a case study that was carried out in order to test this hypothesis on a chosen framework, Salespoint, and an application software, Mafiasoft, developed using this framework.

In the first part of the paper, a central portion of the framework, its referential integrity property, and an important Mafiasoft function, mergeCustomers, are formalized. Since formalization, like any human activity, tends to be error-prone, we have chosen an iterative approach where we heavily rely on feedback from the Alloy model-checking tool.

In the second part of the paper, the specification thus developed is recast in the Object-Z language, which in our opinion is easier to read and also more amenable to formal proofs. The results developed and model-checked in the first part are formally proven in the second. The referential integrity property established for the framework considerably simplifies the proof of the mergeCustomers operation. This finally supports our conjecture.

Contents

1 Introduction

The damages caused by faulty software are immense—over 100 billion euros a year in Europe alone [1]. Clearly, action must be taken to remedy this situation. One popular approach to improving software quality makes use of *object-oriented frameworks*. Such a framework consists of a "set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes" [2]. Informally, it is a skeleton solution (or "blue print") for a specific problem domain. Frameworks are explicitly intended for reuse and typically have a large user base. A framework's frequent reuse increases its quality; applications developed by using it benefit by extension.

While relying on frameworks is an accepted technique in software engineering, the use of *formal methods* is far less common. Formal methods use mathematics to describe a software's behavior and structure. This not only avoids the inconsistencies and imprecision usually inherent in natural language descriptions, it also allows for formal reasoning: System properties can be proven formally instead of merely being validated by non-exhaustive testing. Thus, formal methods help improve a system's quality by eliminating inconsistencies from its specification.

In this paper, we combine the two approaches and consider the following question: Since software development benefits from using formal methods, and application development benefits from using frameworks, does framework-based software development benefit from applying formal methods to the underlying framework? In other words: Does treating a framework with formal methods create a benefit when it comes to formal treatment of an application based on that framework?

This question was explored in the first author's diploma thesis [3] by carrying out a substantial case study. The actual pieces of software examined were Salespoint, a small framework designed to aid in the development of sales applications, and Mafiasoft, one such application. Salespoint ([4, 5]) was developed cooperatively by the TU Dresden and the UniBw München and constitutes the basis of their yearly software development lab classes. It provides many of the aspects of real-world sales applications, such as managing product catalogs, tracking stocks and conducting sale processes, while abstracting from other aspects such as physical distribution (a vital property of modern sales applications) that would overtax inexperienced student developers. Mafiasoft, an application developed with Salespoint during the Munich 2007 lab class, is an accounting software for an aging (and slightly incompetent) Mafia don whose mental shortcomings prevent him from keeping track of the criminal ploys of his clan.

Two formalisms were applied to the described software in the diploma thesis. The first is *Alloy*, an executable specification language created by Daniel Jackson ([6, 7]) that allows assertions about the model to be integrated into the specification and checked within a limited scope by the Alloy Analyzer. This encourages an incremental modelling style in which specifications are developed step by step, with each step being checked immediately. The feedback given to the modeller often exposes flaws early in the development process. This made Alloy an ideal tool for developing hypotheses about Salespoint. Alloy also helped to synthesize a set of precise formal requirements from a vague description of Mafiasoft in natural language. Finally, in order to prove properties about these specifications (and make them more amenable to human eyes), all specifications were translated into *Object-Z* [8, 9], an object-oriented extension to the popular Z notation [10].

The full description of the case study in [3] covers more than a hundred pages. Here, for brevity, we select material from both the Alloy and the Object-Z formalization in such a way that hopefully the overall strucutre of the specifications and the typical steps characteristic for either approach still remain visible.

The rest of the paper is organized as follows. Section 2 is an informal overview of the case study, introducing terms and concepts used in the remainder of the paper. Section 3 introduces Alloy, which in Section 4 is used to formally specify the main data structures of Salespoint; at the same time, some sanity checks are performed in order to avoid errors from being introduced into the formal requirements. In Sections

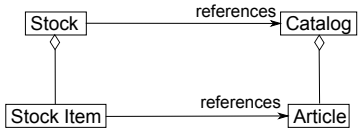

Figure 1: Relationship between catalogs and stocks

5 and 6, the model is iteratively extended with more operations and the referential integrity property. Again, the Alloy model-checking tool is used to crosscheck all additions and thus preserve the correctness of the requirements model. A simplified version of the Salespoint data structures and referential integrity property is recast in Object-Z in Section 7 (for the full treatment including transactions with data baskets see [3]). Also, a proof outline shows that the specified Salespoint operations do not violate referential integrity. Section 8 introduces the Object-Z specification of Mafiasoft (for the Alloy model of Mafiasoft, we gagain refer to [3]). The majority of this section is devoted to a proof that the mergeCustomers operation of Mafiasoft behaves as expected. This proof makes use of the referential integrity property theorem from Secion 7. Finally, in Section 9, we recapitulate what we have achieved and point out some follow-up research topics that we hope to address in the future.

2 Informal Description of Case Study

In this section, we informally introduce the concepts and properties that are central to the case study. Subsections 2.1 and 2.3 describe the terms and concepts of the Salespoint framework and its application Mafiasoft, respectively. Subsections 2.2 and 2.4 define the referential integrity property and the reachability of crimes from customer aliases in Mafiasoft.

2.1 Data Structures and Transactions in Salespoint

A central aspect of Salespoint is the management of the products to be traded. Salespoint differentiates between the products *potentially* for sale, and their actual stock numbers. **Catalogs** contain all those items (also called **articles**) potentially for sale; stock levels are kept in **stocks** containing **stock items**. A stock always references a catalog, and its stock items reference articles in the referenced catalog. These references are immutable. Generalizing this relationship, stocks can be used wherever additional information is to be stored about data. Catalogs and stocks have many similarities, so we will refer to them collectively as **containers** henceforth. Figure 1 shows this relationship.

The **shop** is another important data structure. The shop contains lists of all catalogs and stocks used by an application. Every Salespoint application has exactly one shop.

Salespoint supports manipulating containers in a transactional style through the use of so-called data baskets which act as transaction handles. This means that different data baskets will usually yield different sets of contained items for a container (called that data basket's view on the container). This has considerably complicated the case study in [3] and its description would certainly exceed the scope of this paper by far. Therefore, we only present a simplified version that omits data baskets. However, it should be kept in mind that seemingly simple matters can become exceedingly more complicated when transferred to a multi-view environment.

2.2 Referential Integrity

When introducing catalogs and stocks, we mentioned that the items contained in a stock refer to articles that must be contained in the referenced catalog. We call this property **referential integrity** and define it as follows:

Definition 1 *A shop has the **referential integrity** property (or, for short, **is RI**) if for any stock S contained in the shop*

1. *The referenced catalog of S is also contained in the shop.*
2. *The referenced targets of all stock items contained in S are contained in the catalog referenced by S.*

Referential integrity is an important property, since it guarantees that there are no "dead links" in any of the shop's stocks. However, ensuring its invariance does require some work even in this single-view context. In particular, four operations can be identified which require extra attention with respect to referential integrity:

1. Adding a stock to the shop
2. Removing a catalog from the shop
3. Adding an item to a stock
4. Removing an article from a catalog

2.3 Mafiasoft: a Salespoint Application

Mafiasoft was developed by a group of students during the Munich 2007 lab class. The sofware manages personnel and "customers" (anyone the clan has done business with) and records committed crimes. Since customers often operate under false names, Mafiasoft also keeps a list of aliases for each customer. Crimes are associated with the customers involved in them (where involvement includes the victim). Because customers usually give false names, their real identity is sometimes uncovered only after they have already been entered into the software; in particular, this may cause several entries for what is really the same person. When this becomes evident, they can be merged into one. Merging customers must retain the (indirect) association of aliases and crimes.

These requirements can be mapped to Salespoint's data structures as follows: Crimes are modelled as articles and kept in a global catalog, as are customers. Aliases are modelled as stock items; they reference the customer using the alias and are kept in a global stock. Each customer has an individual stock (his **involvement stock**) containing stock items (**involvements**) that reference those crimes the customer is involved in. Figure 2 shows the relationship between the data structures. Below each class, the type of container it is kept in is indicated in italics.

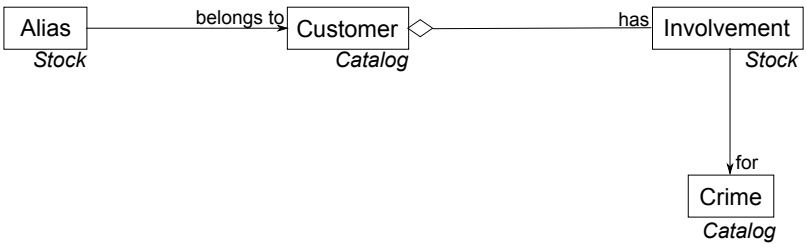

Figure 2: Relationship between catalogs and stocks

2.4 Reachability of Crimes

The requirement given above concerning preservation of the link between aliases and crimes when merging customers is far too vague to be incorporated into a formal specification. In fact, formalizing this requirement was one of the main tasks of the Alloy part of the case study.

We first define what it means if a crime is **reachable** from an alias.

Definition 2 *Let x be a crime, and let a be an alias. Let c be the customer referenced by a. x is **reachable** from a iff*

1. *c is in the catalog of customers*
2. *c's involvements include a reference to crime x*
3. *x is in the catalog of crimes*

Since references are immutable, aliases for the merged customers cannot be reused. Rather, for any alias a referencing one of the original customers, a new alias a' must be found that has the same attribute values as a but references the result of the merge instead. We then call a' a **corresponding alias** for a.

The original requirement can now be formalized along the following lines:

Requirement 3 *Let c be the result of merging two customers, c_1 and c_2. For any alias a referencing c_1 or c_2 and any corresponding alias a', all crimes reachable from a before merging are reachable from a' after merging.*

3 Alloy

Alloy is an executable declarative specification language specifically designed to give instant feedback about a specification to the modeller. It is based on set theory and first-order logic. Central to Alloy is the Alloy Analyzer tool, which searches for instances of the specified model[1]. Alloy allows expressing expectations about a model's inherent properties within the model itself and is able to check them for universal validity by searching for counterexamples. Such counterexamples are visualized graphically by the tool.

[1]We use the terms "model" and "specification" synonymously, and use the term "instance" instead when referring to models in the mathematical sense.

In order to avoid undecidability issues within first-order logic, these searches always have to be restricted to a finite universe. This is done by placing cardinality constraints on all the data types (called *signatures* in Alloy) introduced by the specification. These bounds are called the *scope of analysis*. Usually, relatively small bounds suffice to expose flaws in a model. Daniel Jackson, father of Alloy, summarizes this in the *small scope hypothesis*: "*Most bugs have small counterexamples.*" [6, p. 141] This means that sanity checks of the model can be done fast and, accordingly, often. This in turn promotes an incremental modelling style, in which specifications are developed step by step, with each step being checked immediately against the postulated assertions.

4 Specifying a Basic Model

4.1 Catalogs

We start by introducing signatures for articles and catalogs. In Alloy, primitive types like articles are modelled as empty signatures. Catalogs contain articles. We initially model catalogs as sets of items.

```
sig CITEM {}
sig Catalog {
   items: set CITEM
}
```

Next, we modify this to incorporate data baskets and their different views. We start by introducing a signature for data baskets.

```
sig DB {}
```

Using this signature, we enhance our catalog definition with data baskets and model the contained items as a binary relation between data baskets and articles: if the relation contains the pair (db, c), then article c is contained in the db view of the catalog. The new definition of Catalog is given below; the arrow symbol (–>) is Alloy's way of expressing the cartesian product.

```
sig Catalog {
   items: DB -> CITEM
}
```

We now want to model the catalog operations to add and remove articles. This means we have to incorporate the notion of change over time into our model. We follow the pattern described in [6] for explicitly modelling time. That is, instead of describing change in a relation R by relating two different relations R and R' as before and after state, we describe one relation R at different times t (before) and t' (after).

Accordingly, we introduce a signature for time atoms and append a time component to all those relations that we want to be able to change over time.

```
sig Time{}
sig Catalog {
   items: DB -> CITEM -> Time
}
```

By importing the module ordering and applying it to the Time signature, we impose a total ordering on Time.

```
open util/ordering[Time]
```

The following predicate describes the addition of an article `c` to the `db` view of catalog `C` in the time step from `t` to `t'`:

```
pred cat_add[C:Catalog, c:CITEM, db:DB, t,t':Time] {
   C.items.t' = C.items.t + (db -> c)
}
```

The dot operator used here is a special join operator introduced by Alloy. Although it is a very powerful concept, its exact definition is somewhat unintuitive. For the most part, it can simply be thought of as attribute access: `C.items` accesses the `items` relation of catalog `C`; likewise, `C.items.t` denotes the state of that relation at time `t`.

The predicate therefore specifies the state of the `items` relation of catalog `C` at time `t'` to be the same as that at time `t`, but with the addition of the pair `(db -> c)`. However, this definition *forces* the use of a data basket instead of leaving it optional. We could fix this by making the `db` parameter optional (signified by the "less-than-or-one" quantifier **lone**), giving:

```
pred cat_add[C:Catalog, c:CITEM, db:lone DB, t,t':Time] {
   some db implies
      C.items.t' = C.items.t + (db -> c) else
      C.items.t' = C.items.t + (DB -> c)
}
```

This adds `c` to all views if no data basket is used. But this approach will lead to trouble down the road: how do we determine the base view when we inspect the catalog without a data basket? Therefore, we decide to take an alternative approach and introduce a special **null data basket** that we will use from now on to signify that no data basket is to be used.

```
one sig NullDB extends DB {}
```

The operation predicate has to be changed only slightly to accomodate the new approach—we have caught this problem before it could do much harm.

```
pred cat_add[C:Catalog, c:CITEM, db:DB, t,t':Time] {
   db = NullDB implies
      C.items.t' = C.items.t + (DB -> c) else
      C.items.t' = C.items.t + (db -> c)
}
```

In much the same way, the removal of articles can be modelled:

```
pred cat_remove[C:Catalog, c:CITEM, db:DB, t,t':Time] {
   db = NullDB implies
      C.items.t' = C.items.t - (DB -> c) else
      C.items.t' = C.items.t - (db -> c)
}
```

In addition, we need an operation to test whether a view contains an article.

```
pred cat_contains[C:Catalog, c:CITEM, db:DB, t:Time] {
   (db -> c) in C.items.t
}
```

4.2 Some Sanity Checks

This concludes the initial portion of the model. We now check some of its basic properties. First, we want to make sure that articles are really contained after they have been added.

Figure 3: Counterexample for addLocal

```
assert addedVisible {
   all C:Catalog, c:CITEM, db:DB, t,t':Time |
      cat_add[C,c,db,t,t'] implies
         cat_contains[C,c,db,t']
}
```

The command **check** addedVisible directs the tool to check the assertion using the default scope of analysis three. This produces no counterexamples.

We also want to check that adding an article using a data basket (and thus modifying its view) leaves all other views unaffected.

```
assert addLocal {
   all C:Catalog, c:CITEM, db:DB, t,t':Time |
      cat_add[C,c,db,t,t'] implies all db2:DB-db |
         C.items.t[db2] = C.items.t'[db2]
}
```

Checking this assertion fails, giving the counterexample shown in Fig. 3. The article is added into both the DB view and the NullDB view of the catalog, as indicated by the arcs between the nodes (the view is identified in square brackets in the arc label). Note that Alloy gives the names of quantified variables inside the corresponding nodes, making it much easier to interpret the instance. We have also projected the instance over the Time signature, producing separate graphs for each point in time.

Evidently, the assertion fails when adding with the null data basket. This isn't an error in the specification, we were simply too sloppy in the formulation of the assertion addLocal. Addition with the null data basket is supposed to have global effects, the locality requirement therefore only holds for other data baskets (which we will henceforth call **true** data baskets). Quantifying db over DB−NullDB in the second line solves the problem.

We have also checked corresponding assertions for the remove operation, and validated that under certain conditions, remove acts as an undo operation for add.

4.3 Stocks

The definition of stocks and stock items is very similar to catalogs and articles, but includes additional attributes for the references to catalogs and articles.

```
sig SITEM {
   target: CITEM
}
sig Stock {
   target: Catalog,
   items: DB -> SITEM -> Time
}
```

Note that the target attributes are immutable (i.e. have no time component).

The add, remove and contains operations can be modelled in the same way as their catalog counterparts.

```
pred stk_add[S:Stock, s:SITEM, db:DB, t,t':Time] {
   db = NullDB implies
      S.items.t' = S.items.t + (DB -> s) else
      S.items.t' = S.items.t + (db -> s)
}
pred stk_remove[S:Stock, s:SITEM, db:DB, t,t':Time] {
   db = NullDB implies
      S.items.t' = S.items.t - (DB -> s) else
      S.items.t' = S.items.t - (db -> s)
}
pred stk_contains[S:Stock, s:SITEM, db:DB, t:Time] {
   (db -> s) in S.items.t
}
```

For later use, we add the following predicate to test whether a container has changed. The domain restriction operator <: serves as a case distinction here by restricting the argument X to catalogs and stocks, respectively, allowing us to use the same predicate for catalogs, stocks, and even (possibly mixed) sets thereof.

```
pred change[X:Catalog+Stock, t,t':Time] {
   { let C = Catalog <: X |
     let C_items = (Catalog <: items) |
         (C <: C_items).t' != (C <: C_items).t } or
   { let S = Stock <: X |
     let S_items = (Stock <: items) |
         (S <: S_items).t' != (S <: S_items).t }
}
```

5 Refining the Model to Meet Requirements

A stock item's data is meaningless without the article it refers to. However, although our model is now able to add and remove items in a transactional manner, it does not guarantee that a stock item's referenced article actually exists in the referenced catalog. Below, we will fix this unsatisfactory situation.

5.1 Developing the Referential Integrity Property

We first need to bring this rather vague requirement into the more precise form of a model property definition. Basically, we want to avoid dead links in stocks: if a stock item is contained in a stock, its referenced article should be contained in the catalog referenced by that stock. Adjusting this statement to our multi-view environment, we obtain the following definition (all introduced objects are implicitly understood to be part of $\mathcal{M}$):

Definition 4 *Let $\mathcal{M}$ be a model instance. $\mathcal{M}$ has the* ***referential integrity property*** *at time* `t` *if, at that time, for any catalog* `C`*, any data basket* `db` *and any stock* `S` *referencing* `C`*, the referenced articles of all stock items contained in the* `db` *view of* `S` *are contained in the* `db` *view of* `C`*.*

This definition can be translated to Alloy as follows.

```
pred ReferentialIntegrity[t:Time] {
   all S:Stock, db:DB, s:S.items.t[db] |
      cat_contains[S.target,s.target,db,t]
}
```

Because we have modelled references as immutable, only two operations need to be changed to meet the requirement: adding items to a stock and removing articles from a catalog.

5.2 Adding Stock Items

When adding a stock item `s` with a data basket `db`, we have to make sure that its referenced article is contained in the `db` view of the target catalog. Therefore, we add the line

```
cat_contains[S.target,s.target,db,t]
```

and obtain the following new version of stk_add:

```
pred stk_add[S:Stock, s:SITEM, db:DB, t,t':Time] {
   cat_contains[S.target,s.target,db,t]
   db = NullDB implies
      S.items.t' = S.items.t + (DB -> s) else
      S.items.t' = S.items.t + (db -> s)
}
```

We now check that our redefined add operation preserves referential integrity: for a given stock `S`, if the predicate held before the add operation, and if no container other than `S` changes in between, then the predicate should hold after the add operation.

```
assert addCorrect {
   all db:DB, S:Stock, s:SITEM, t,t':Time |
      {
         ReferentialIntegrity[S,t]
         stk_add[S,s,db,t,t']
         !change[Catalog+Stock-S,t,t']
      } implies
         ReferentialIntegrity[S,t']
}
check addCorrect
```

Alloy finds a counterexample. By reducing the scope of analysis, we obtain the minimal counterexample shown in Fig. 4.

In the stk_add operation we again failed to handle the null data basket correctly: since adding with the null data basket will add the stock item to all views, its target article should accordingly be visible in all views. Our precondition currently only checks that its target article is contained in the base view. Thus, we modify stk_add as follows:

```
pred stk_add[S:Stock, s:SITEM, db:DB, t,t':Time] {
   db = NullDB implies {
      cat_contains[S.target,s.target,DB,t]
      S.items.t' = S.items.t + (DB -> s)
   } else {
```

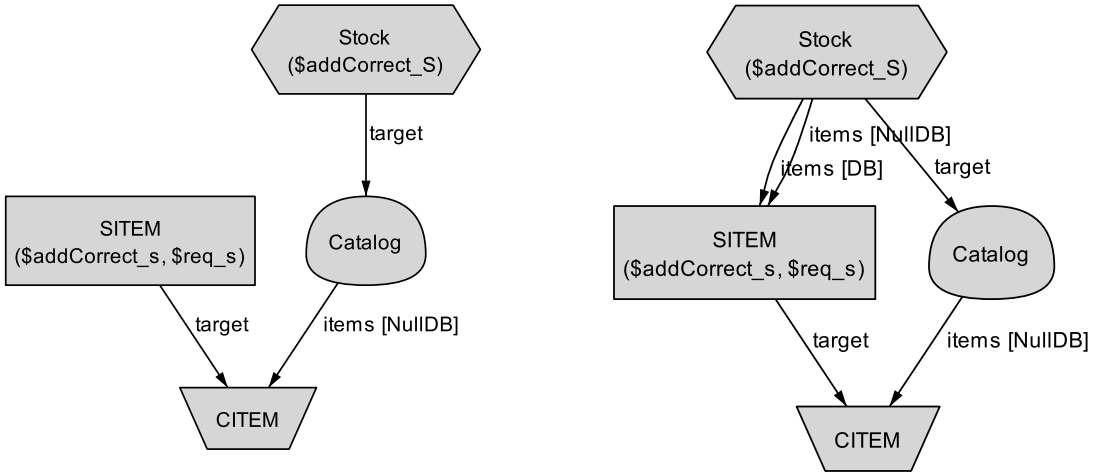

Figure 4: Counterexample for addCorrect

```
        cat_contains[S.target,s.target,db,t]
        S.items.t' = S.items.t + (db -> s)
    }
}
```

Rechecking the assertion with this version of the operation produces no more counterexamples.

5.3 Removing Articles

The second operation we have to consider is cat_remove. Before removing an article, one has to check that no stock items reference it. It is tempting to rely on universal quantification over the Stock signature here, augmenting the operation's precondition with something like

```
all S:Stock | no s:S.items.t[db] |
    s.target = c
```

for removal with a true data basket and an appropriately stronger version for the null data basket.

However, considering that we are specifying an object-oriented framework, this is not a very good approach. A check like the one above will be very hard if not impossible to translate to an object-oriented programming language. Also, it might cause trouble with very large scopes later on: if the universe contains more stocks than are actually needed by an application using the framework, the "no references" precondition should only be required for the "relevant" ones. An overpopulated universe should not interfere with a remove operation that would be valid in a universe restricted to the application context.

Therefore, we have decided to adapt the *publish-subscribe pattern* [11, p. 293] that is popular in object-oriented designs. Stocks can be registered with a catalog. When an article is removed from the catalog, all stocks registered with the catalog will be checked for a stock item referencing the article.

First, catalogs need an additional attribute for their sets of subscribers.

```
sig Catalog {
    items: DB -> CITEM -> Time,
    sub: Stock -> Time
}
```

We also provide operations to register and unregister a stock with a catalog. Stocks can only be registered with the catalog they reference. The predicate subChange tests for change in catalog's set of subscribers.

```
pred register[S:Stock, C:Catalog, t,t':Time] {
   S.target = C
   C.sub.t' = C.sub.t + S
}
pred unregister[S:Stock, C:Catalog, t,t':Time] {
   C.sub.t' = C.sub.t - S
}
pred subChange[C:Catalog, t,t':Time] {
   (C <: sub).t != (C <: sub).t'
}
```

Using the set of subscribers, we can now redefine the operation cat_remove, remembering to pay extra attention to the null data basket.

```
pred cat_remove[C:Catalog, c:CITEM, db:DB, t,t':Time] {
   db = NullDB implies {
      all S:C.sub.t | no s:S.items.t[DB] |
         s.target = c
      C.items.t' = C.items.t - (DB -> c)
   } else {
      all S:C.sub.t | no s:S.items.t[db] |
         s.target = c
      C.items.t' = C.items.t - (db -> c)
   }
}
```

Adopting this approach means that we have to adjust the referential integrity definition slightly: instead of extending the containment claims to all stocks, we now only make them for those stocks registered with a catalog, giving the following modified definition:

Definition 5 *Let $\mathcal{M}$ be a model instance. $\mathcal{M}$ has the* ***referential integrity property*** *at time* t *if, at that time, for any catalog* C*, any data basket* db *and any stock* S *registered with* C*, the referenced targets of all stock items contained in the* db *view of* S *are contained in the* db *view of* C*.*

Translation to Alloy gives the following predicate.

```
pred ReferentialIntegrity[t:Time] {
   all C:Catalog, db:DB |
      all S:C.sub.t, s:S.items.t[db] |
         cat_contains[C,s.target,db,t]
}
```

Like stk_add, we test the redefined remove operation against the referential integrity definition. If ReferentialIntegrity held before removing the article, and no stock changes, then ReferentialIntegrity should hold after the removal.

```
assert removeCorrect {
   all db:DB, C:Catalog, c:CITEM, t,t':Time {
      {
         ReferentialIntegrity[t]
         cat_remove[C,c,db,t,t']
         !change[Stock,t,t']
      } implies
         ReferentialIntegrity[t']
   }
}
check removeCorrect
```

This produces a counterexample caused by a stock that is registered with C, but does not reference C. To prevent this kind of situation, we introduce a *fact*. Facts express constraints placed on a model. When the Analyzer searches for model instances, it only takes instances into account that satisfy all facts.

```
fact SubscriberTargets {
   all C:Catalog, t:Time |
      C.sub.t.target in C
}
```

Now the removeCorrect assertion, and a recheck of the addCorrect assertion made necessary by the modification of ReferentialIntegrity, execute without counterexamples.

Changes to already defined operations make rechecks of their associated assertions necessary. In our case, this means rerunning the sanity checks from Sect. 4. These all execute without counterexamples. This is hardly surprising: our modifications have only strengthened the operations' preconditions; therefore, any assertions valid beforehand were certain to remain so.

6 Validating Property Invariance

We have now developed a model for manipulating containers in transactional style and in keeping with the referential integrity definition given above. In this section, we take referential integrity one step further, and confirm that it is an invariant of our model (i.e. holds for at all times in all possible instances).

6.1 Constraining Instance Transitions

Before we can expect referential integrity to be a model invariant, we have to constrain the model instances' allowed transitions to those operations we defined in Sects. 4 and 5. For this purpose, we again use Alloy's fact construct. Our fact needs to express the following conditions:

1. In the beginning, all catalogs and stocks are empty, and no stocks are registered with any catalog.
2. Changes to a container's items relation are restricted to the previously defined operations.
3. At most one container can change in a time step.
4. Changes in a catalog's set of subscribers are restricted to the operations register and unregister (note that this will make the fact SubscriberTargets obsolete).
5. Changes in a catalog's set of subscribers and any container's items relation are mutually exclusive, that is, if a stock is registered with a catalog in a time step, no items can be added to or removed from any container in that time step, and vice versa.

```
fact traces {
   // condition 1
   no (Catalog.items+Stock.items).first
   no Catalog.sub.first
   all t:Time-last | let t' = next[t] {
      // condition 2
      all C:Catalog | change[C,t,t'] implies
         some c:CITEM, db:DB |
            cat_add[C,c,db,t,t'] or
            cat_remove[C,c,db,t,t']
      all S:Stock | change[S,t,t'] implies
         some s:SITEM, db:DB |
```

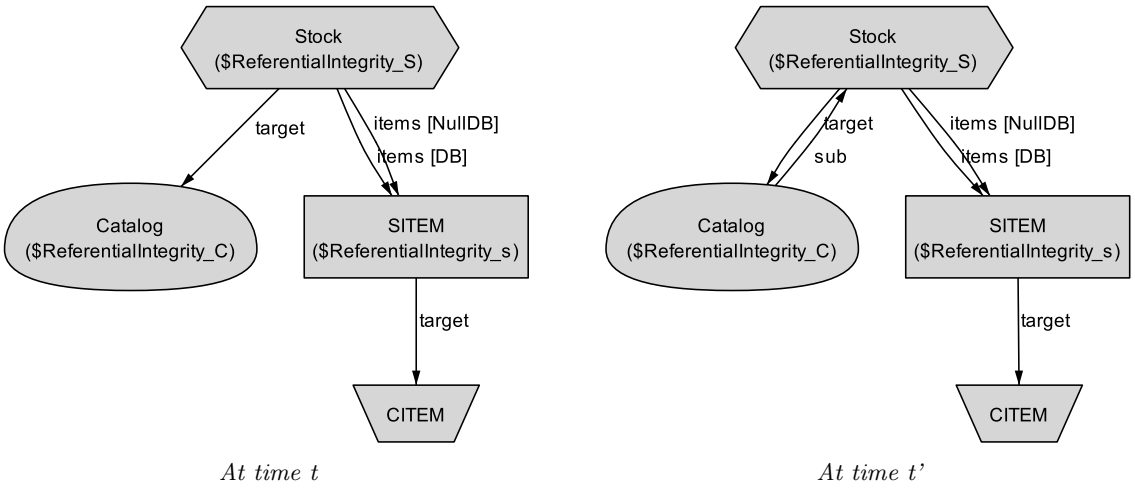

Figure 5: Counterexample for AlwaysReferentialIntegrity

```
            stk_add[S,s,db,t,t'] or
            stk_remove[S,s,db,t,t']
        // condition 3
        lone X: Catalog+Stock | change[X,t,t']
        // condition 4
        all C:Catalog | subChange[C,t,t'] implies
           some S:Stock |
              register[S,C,t,t'] or
              unregister[S,C,t,t']
        // condition 5
        subChange[Catalog,t,t'] implies
           !change[Catalog+Stock,t,t']
    }
}
```

6.2 Checking Property Invariance

Having thus constrained the allowed transitions, we expect referential integrity to be a model invariant.

```
assert AlwaysReferentialIntegrity {
   all t:Time | ReferentialIntegrity[t]
}
check AlwaysReferentialIntegrity for 3 but 10 Time
```

However, the Analyzer finds a counterexample. For lack of space, we cannot show the entire trace here, but Fig. 5 shows the crucial time step that causes the violation.

At time t', the referential integrity property is violated: although the stock is registered with the catalog, it contains a stock item for which the referenced article is not contained in the target catalog. Leading up to this situation, the stock item was added to the stock, after its referenced article had been added to the target catalog. Then the article was removed from the catalog. This is valid because at that point, the stock was not registered with the catalog. In the final step (shown in Fig. 5), the stock is registered with the catalog, bringing it into the scope of referential integrity and causing the violation. We can fix the problem by slightly modifying the register operation.

```
pred register[S:Stock, C:Catalog, t,t':Time] {
   all db:DB, s:S.items.t[db] |
      cat_contains[C,s.target,db,t]
   S.target = C
   C.sub.t' = C.sub.t + S
}
```

Now, Alloy finds no more counterexamples and we conclude that referential integrity probably is an invariant of the specification.

Treatment of Salespoint with Alloy is now complete. The version we have developed here actually deviates slightly from the structure described in Section 2 by replacing the shop with the publish-subscribe-pattern. A version that follows the structure originally outlined more closely can be found in [3]. We have chosen to display this alternate approach, however, because it is the notationally and conceptually clearer of the two versions. As can be seen in [3], the two versions are actually not all that different: most of the operations are very similar, often even identical. As we are looking for a lower level of abstraction in our Object-Z specification (that is, a specification resembling actual code more closely), we will switch back to the shop based approach now.

7 Formal Specification of Salespoint

In this section, those Salespoint concepts outlined in 2.1 and the RI property introduced in 2.2 are specified formally using Object-Z.

7.1 Catalogs

Catalogs contain articles, so we introduce a class *CITEM*. Articles are a hot spot (short for those points where the framework has to be adapted in an application) of the Salespoint framework and carry no inherent meaning on the framework level. Therefore, the class body remains empty here.

CITEM

Reducing containers to only one view simplifies the modelling of the contained items to an (initially empty) set.

Catalog

$items : \mathbb{P} \downarrow CITEM$

Init

$items = \varnothing$

Articles can be added to and removed from a catalog.

add

$\Delta(items)$
$c? : \downarrow CITEM$

$items' = items \cup \{c?\}$

remove

$\Delta(items)$
$c? : \downarrow CITEM$

$items' = items \setminus \{c?\}$

7.2 Stocks

Structurally, stocks are very similar to catalogs. Like articles, stock items are modelled as an empty class.

SITEM

To model stock items' references, we introduce a global total function $\blacktriangleright$ that maps stock items to their referenced articles.

$\blacktriangleright : \downarrow SITEM \to \downarrow CITEM$

The following line allows us to use $\blacktriangleright$ as a postfix operator.

$\_^{\blacktriangleright} == \blacktriangleright \_$

Apart from the *add* operation, all definitions made above for catalogs can be reused here with only slight modifications. Stocks' references to catalogs are modelled by the class constant *target*.

Stock

$target : Catalog$

$items : \mathbb{P} \downarrow SITEM$

INIT

$items = \varnothing$

remove

$\Delta(items)$
$s? : \downarrow SITEM$

$items' = items \setminus \{s?\}$

When adding a stock item $s?$ to a stock $S?$, we first have to check that this would not violate referential integrity. That is, $s?^{\blacktriangleright}$ must exist in the catalog referenced by $S?$ as captured by the schema *TargetExists*.

TargetExists

$s? : \downarrow SITEM$

$s?^{\blacktriangleright} \in target.items$

This becomes part of the operation's precondition.

add

$\Delta(items)$
$s? : \downarrow SITEM$

TargetExists
$items' = items \cup \{s?\}$

7.3 The Shop

The shop contains sets of all containers used by the application. It provides operations to add and remove items from these sets. When adding a stock or removing a catalog, care must be taken so as not to violate referential integrity.

Shop

$$catalogs : \mathbb{P}\, Catalog;\ stocks : \mathbb{P}\, Stock$$

INIT

$$catalogs = \varnothing \land stocks = \varnothing$$

addCatalog

$$\Delta(catalogs)$$
$$C? : Catalog$$
$$catalogs' = catalogs \cup \{C?\}$$

removeCatalog

$$\Delta(catalogs)$$
$$C? : Catalog$$
$$\nexists S : stocks \bullet S.target = C?$$
$$catalogs' = catalogs \setminus \{C?\}$$

addStock

$$\Delta(stocks)$$
$$S? : Stock$$
$$S?.target \in catalogs$$
$$\forall s : S?.items \bullet s^{\blacktriangleright} \in S?.target.items$$
$$stocks' = stocks \cup \{S?\}$$

removeStock

$$\Delta(stocks)$$
$$S? : Stock$$
$$stocks' = stocks \setminus \{S?\}$$

selectStock

$$S? : stocks$$

selectCatalog

$$C? : catalogs$$

The *remove* operation on catalogs is not yet defined in a way that will preserve referential integrity: Before removing, we must check if any stock items reference the article in question. One might be tempted to add the following to the *remove* precondition:

$$\forall S : Stock \nexists s : S.items \bullet s^{\blacktriangleright} = c?$$

However, this quantifies over all conceivable bindings of the *Stock* class schema, and among those there are certainly some that will not pass the test. The real intention is to quantify over only those stocks that are actually relevant to the application. This information is not available at the catalog level. Therefore, we have to promote the operation to the shop level.

The schema *Referenced* formalizes what it means if an article is referenced within the shop.

Referenced

$$c? : \downarrow CITEM$$

$$\exists S : stocks, s : S.items \bullet s^{\blacktriangleright} = c?$$

We use this schema to enhance the precondition of the (promoted) *remove* operation.

$$cat\_remove \mathrel{\widehat=} selectCatalog \bullet \left(\neg Referenced \land C?.remove\right)$$

Promotion of the remaining container operations is straightforward.

$$stk\_add \mathrel{\widehat=} selectStock \bullet S?.add$$
$$stk\_remove \mathrel{\widehat=} selectStock \bullet S?.remove$$
$$cat\_add \mathrel{\widehat=} selectCatalog \bullet C?.add$$

7.4 Referential Integrity

Here we outline the referential integrity invariance proof. A series of lemmas shows that the initial state is RI, and each of the operations listed in 2.2 preserves it. The property's invariance is then shown via structural induction.

Referential Integrity, as defined in 1 in Section 2.2 can be translated to Z as follows:

ReferentialIntegrity

$$\forall S : stocks \bullet S.target \in catalogs$$
$$\forall S : stocks, s : S.items \bullet s^{\blacktriangleright} \in S.target.items$$

For lack of space and since most of the introductory lemmas are quite straightforward, we will omit their proofs here except for one example.

Lemma 6 *In its initial state, the shop is RI.*

Lemma 7 *Removing catalogs from the shop preserves referential integrity.*

Lemma 8 *Adding stocks to the shop preserves referential integrity.*

Lemma 9 *Adding stock items to a stock preserves referential integrity.*

Lemma 10 *Removing articles preserves referential integrity.*

Proof

We present an argument by contradiction. Suppose article c is removed from catalog C. Let Σ be the state of the shop before the removal, and let Σ' be the state after removal.
Assume Σ is RI, but Σ' isn't. Then in Σ', there is a stock S and a stock item $s \in S.items$ such that $s^{\blacktriangleright} \notin S.target.items$. Since Σ is RI, the violation can only have been caused by the removal of c from C; it follows that $S.target = C$ and $s^{\blacktriangleright} = c$. Moreover, $S \in \Sigma.stocks$.
Because Σ is RI, c must be contained in C before the removal. Therefore, the predicate of *Referenced* is satisfied, which means the precondition of the remove operation is dissatisfied. Therefore, $C.items = C.items'$, which implies $c \in C.items'$. Contradiction.

Having proved that the shop's initial state satisfies referential integrity, and that all operations preserve it, we can conclude by structural induction that referential integrity is an invariant of the shop data structure.

Theorem 11 *If manipulation of containers is restricted to the operations cat_add, cat_remove, stk_add and stk_remove, referential integrity is invariant.*

Proof

Let Σ be a shop state. If Σ is the initial state *INIT*, Σ is RI according to lemma 6. Otherwise, there is an operation *op* and a shop state $\widetilde{\Sigma}$ such that $\Sigma = \widetilde{\Sigma}.op$. Assuming that $\widetilde{\Sigma}$ is RI, we can conclude Σ's referential integrity from lemmas 7 through 10.

8 Mafiasoft

Applying Salespoint to Mafiasoft consists of three stages: First, the data needed by Mafiasoft has to be organized in such a way as to be conformant with Salespoint, as was outlined in 2.3. This includes specializing the *Shop* class to allow for easy access to particular containers. Then a specification of the operation for merging customers is developed, followed by a formal proof that this specification preserves the reachability of crimes.

8.1 Specification of Data Structures

Applying the structure developed in 2.3, we obtain the following definitions. First, we introduce a new type for names (this is really a technicality).

$[NAME]$

As described in Section 2.3, crimes are modelled as articles.

Crime

$CITEM$

Similarly, aliases are modelled as stock items. They reference customers (which will be established after the definition of the *Customer* class) and have the given false name as an attribute.

Alias

$SITEM$

$name : NAME$

Customers are also modelled as articles. Each customer has an individual involvement stock. The class invariant states that any items contained in that stock reference crimes. When *Customer* is in its initial state, its involvement stock is also in the initial state.

Customer

$CITEM$

$involved : Stock$

$\blacktriangleright \ (\!|\ involved.items\ |\!) \subseteq Crime$

Init

$involved.INIT$

The following line ensures that aliases only reference customers.

$\blacktriangleright \ (\!|\ Alias\ |\!) \subseteq Customer$

Shop is specialized as follows:

Mafia

Shop [**redef** *Init*]

$customers : Catalog;\ crimes : Catalog$
$aliases : Stock$

$customers \in catalogs \land customers.items \subseteq Customer$
$crimes \in catalogs \land crimes.items \subseteq Crime$
$aliases \in stocks \land aliases.items \subseteq Alias$
$\forall\, c : customers.items \bullet c.involved.target = crimes \land c.involved \in stocks$

Init

$catalogs = \{customers, crimes\} \land stocks = \{aliases\}$
$customers.INIT \land crimes.INIT \land aliases.INIT$

Note that the class invariant—besides placing some purely technical type safety constraints—requires customers' involvement stocks to be contained in the shop.

8.2 Operation to Merge Customers

Suppose we want to merge the customers $c_1?$ and $c_2?$, giving the new customer $c!$ as a result. For this purpose, we have to perform the following steps:

1. Add $c!$'s involvement stock to the Mafia shop's set of stocks
2. Copy all involvements from $c_1?$ and $c_2?$ to $c!$
3. Add $c!$ to *customers* and create corresponding aliases for all aliases of $c_1?$ and $c_2?$
4. Remove all aliases of $c_1?$ and $c_2?$
5. Remove $c_1?$ and $c_2?$ and their involvement stocks

In order to remove $c_1?$ and $c_2?$ they must not be referenced by any stock items besides their aliases.

$$\forall S : stocks \setminus aliases \mid S.target = customers \bullet \blacktriangleright (\!| S.items |\!) \cap \{c_1?, c_2?\} = \varnothing$$

The following purely technical requirement ensures we can add $c!$ to the catalog of customers.

$$c!.involved.target = customers$$

We begin by adding the involvement stock of $c!$ to the Mafia shop and copying the appropriate involvements.

$$addStock[c!.involved/S?]_{⨾}^{⨾}$$
$$_{⨾}^{⨾}\ (s? : c_1?.involved.items \cup c_2?.involved.items \bullet stk\_add[c!.involved/S?])$$

After adding $c!$ to the catalog of customers

$$cat\_add[c!/c?, customers/C?]\ ,$$

we have to create corresponding aliases for all aliases of $c_1?$ and $c_2?$. We therefore first determine the set A of their aliases.

$$A == \{a : aliases.items \mid a^{\blacktriangleright} \in \{c_1?, c_2?\}\}$$

From A, we create the set B of corresponding aliases.

$$B == \{a : A \bullet \mu\, b : Alias \mid b.name = a.name \land b^{\blacktriangleright} = c!\}$$

All the aliases in A have to be removed from the stock of aliases and all the aliases in B added to it.

$$_{⨾}^{⨾}\ (s? : A \bullet stk\_remove[aliases/S?])\ _{⨾}^{⨾}$$
$$_{⨾}^{⨾}\ (s? : B \bullet stk\_add[aliases/S?])$$

Finally, we remove $c_1?$ and $c_2?$ from the catalog of customers and remove their involvement stocks from the Mafia shop.

$$cat\_remove[customers/C?, c_1?/c?]\ _{⨾}^{⨾}\ cat\_remove[customers/C?, c_2?/c?]_{⨾}^{⨾}$$
$$removeStock[c_1?.involved/S?]\ _{⨾}^{⨾}\ removeStock[c_2?.involved/S?]$$

This results in the following operation schema for merging customers:

mergeCustomers

$c_1?, c_2? : Customer;\ c! : Customer$

$\forall S : stocks \setminus aliases \mid S.target = customers \bullet \blacktriangleright (\!|\ S.items\ |\!) \cap \{c_1?, c_2?\} = \varnothing$
$c!.involved.target = customers$

let $A == \{a : aliases.items \mid a^{\blacktriangleright} \in \{c_1?, c_2?\}\}$,
$B == \{a : A \bullet \mu\, b : Alias \mid b.name = a.name \wedge b^{\blacktriangleright} = c!\} \bullet$

$addStock[c!.involved/S?]^{\circ}_{9}$
$^{\circ}_{9}\ (s? : c_1?.involved.items \cup c_2?.involved.items \bullet stk\_add[c!.involved/S?])\ ^{\circ}_{9}$
$cat\_add[c!/c?, customers/C?]^{\circ}_{9}$
$^{\circ}_{9}\ (s? : A \bullet stk\_remove[aliases/S?])\ ^{\circ}_{9}$
$^{\circ}_{9}\ (s? : B \bullet stk\_add[aliases/S?])\ ^{\circ}_{9}$
$cat\_remove[customers/C?, c_1?/c?]\ ^{\circ}_{9}\ cat\_remove[customers/C?, c_2?/c?]^{\circ}_{9}$
$removeStock[c_1?.involved/S?]\ ^{\circ}_{9}\ removeStock[c_2?.involved/S?]$

8.3 Properties of *mergeCustomers*: Preservation of Reachability

After these preparations, we now formally prove that *mergeCustomers* preserves the reachability of crimes. Some introductory lemmas are necessary. In the following, let Σ be a Mafia shop state and let Σ' stand for $\Sigma.mergeCustomers$.

Lemma 12 *$c!$ is involved in all crimes that $c_1?$ or $c_2?$ were involved in, that is*

$$c!.involved.items \subseteq c_1?.involved.items \cup c_2?.involved.items\ .$$

Lemma 13 *The catalog of crimes does not change.*

The proofs of these lemmas are omitted due to their triviality.

Lemma 14 *For every alias $a \in aliases.items$ with $a^{\blacktriangleright} \in \{c_1?, c_2?\}$ there is a corresponding alias $b \in aliases.items'$ (that is, after merging).*

Proof

Let a be an alias referencing $c_1?$ or $c_2?$ before merging. By definition $a \in A$. Then B contains an alias b with $b.name = a.name \wedge b^{\blacktriangleright} = c!$. b is a corresponding alias to a. Because $c!$ is contained in *customers*, addition of b to *aliases* is valid (the precondition is satisfied), so b is contained in *aliases* after merging.

Lemma 15 *Σ' is RI.*

Proof

mergeCustomers relies only on shop operations. It follows from theorem 11 that Σ' is RI.

We can now show that *mergeCustomers* preserves the reachability of crimes.

Theorem 16 *Let x be a crime. For any alias a in Σ with $a^\blacktriangleright \in \{c_1?, c_2?\}$, there is a corresponding alias b in Σ' such that if x is reachable from a in Σ, x is reachable from b in Σ'.*

Proof

Existence and containment of b has already been established by lemma 14. Lemma 15 shows that Σ' is RI; it follows that $c! = b^\blacktriangleright$ is in $\Sigma'.customers$. Since x is reachable from a in Σ, there has to be an involvement i referencing x in $c_1?.involved$ or $c_2?.involved$. It follows from lemma 12 that $i \in c!.involved.items$. Because of the *Mafia* class invariant, $c!.involved \in \Sigma'.stocks$, that is, the involvement stock of $c!$ falls within the scope of referential integrity. Referential integrity then requires $x \in crimes.items$. Thus, x is reachable from b in Σ'.

9 Conclusions and Future Work

In the Alloy part of this paper, we have systematically engineered a formal specification for portions of the Salespoint framework. A simple initial model was iteratively refined to meet additional requirements we had identified. Throughout the process, we continuously relied on Alloy's constraint solving abilities to find errors in the specification that would otherwise have gone unnoticed. The requirement model thus derived has already been tested for consistency.

The Alloy language and Analyzer Tool turned out to be particularly well suited for our task. This may be due to the following facts: Alloy allows for underspecification which is indispensable in the context of frameworks, i.e., incomplete programs. Alloy supports systematic tests with a complete test coverage for small models. The Alloy Analyzer is robust and fast; all tests described in this paper were completed within seconds (however, some of the bulky test cases in [3] took up to five hours). Alloy presents examples and counterexamples in a graphical format which can even be customized (using, say, projections in order not to become overwhelmed by the amount of detail).

In the Object-Z part, the engineered specification is used as the basis for formal treatment of Salespoint with a notation that is more amenable to the human eye and also somewhat more expressive. This part of the case study was simplified significantly by omitting data baskets. In particular, the merging of customers is considerably more complicated with multi-view containers and made the development of a pattern called *injection* necessary to allow for addition of items into arbitrary views of a container. Application of this pattern greatly increased the operation's complexity, making it necessary to spell out proofs that could be left out here due to their triviality. Most of these proofs, again, relied heavily on referential integrity.

However, even in this simplified version, reasoning in the application context profited from the existence of proven framework properties. Reachability of crimes is comprised of a series of containment claims within a chain of references. Referential integrity meant that we only had to prove containment at the beginning of that chain. The rest then followed directly from referential integrity in conjunction with the *Mafia* class invariant.

It could be argued that the application might just as well have been constructed without using the framework. In fact, it is often true that for a given problem within a framework's problem domain, a

slimmer version specifically tailored to the problem can be found without using the framework—provided one is willing to invest enough work. Whether or not relying on frameworks is sensible is not a subject of this paper, although in light of the quality improvement gained by repeated reuse, the authors would argue against hand-crafting solutions where components or frameworks could be reused instead. As our case study illustrates, reuse mechanisms can be expected to work just as well on the formal specification level as they do on the code level.

With Object-Z, reuse at the class level was introduced to the Z notation. This made it necessary to hide certain aspects of a class through complementary visibility [8, 9] and invisibility [12] lists. Here, we extend reuse to the specification level. It should not be surprising that similar visibility restrictions become necessary at a larger scale. However, no appropriate constructs are available in either Z or Object-Z. In fact, the notation is not really geared toward working with multiple specifications at the same time. At most, specifications are in a "vertical" relationship where one is a refinement of the other. Here, we have a rather "horizontal" relationship in which one specification instantiates and specializes the other. Therefore, [3] informally suggests constructs for naming and importing specifications, and for restricting visibility of imported features. It uses these to offer a stronger, tautological version of the referential integrity invariance theorem instead of the conditional version developed here. These constructs will have to be formalized if Z is to be applied to the application/framework relationship constructively.

The ability to automatically translate specifications from Alloy to other notations and vice versa would be very valuable. Because of its prominence and popularity, Z would be a good starting point for this. Furthermore, Alloy itself is by design fairly similar to Z [6, p. xii]; in fact, the Alloy modelling language can roughly be seen as a subset of Z. As demonstrated in [3], translation from one formalism to the other is often straightforward; in places, the specifications are even identical.

However, some difficulties remain, particularly with the use of existential quantification in underpopularized universes, and higher-order expressions, which cannot always be skolemized. For automatic or at least tool-based translation between Alloy and Z, future work has to identify problematic modelling patterns and find possible modelling idioms to meet them. Ideally, at the end of this process, specification engineering will combine the notational elegance of Z with the explorative powers of Alloy, letting the modeller switch between both at the press of a button. Some results concerning translation from Z to Alloy have been established in [13] and [14], but many issues remain yet unresolved.

The results of the case study support our working hypothesis, i.e. that proving framework properties helps proving properties of applications developed using that framework. Assuming that we have not picked a pathological example (for which there is no indication), further research in the same vein is therefore warranted. For example, the relationship between proofs at the application and the framework level should be examined. In particular, the following questions should be addressed: Do application proofs differ fundamentally from framework proofs? Does proof reuse resemble code reuse structurally, for instance, does the inversion of control caused by code reuse extend to the proof level? Is it possible to apply the hot spot/frozen spot concept to this relationship? Addressing these questions, and further examining the relationship between formal treatment of frameworks and their applications, will hopefully lead to "'formal frameworks"' in which much of an application's proof structure is already provided by the framework, waiting to be appropriately extended and instantiated by the appication modeller—much as is already done with code.

References

[1] Bundesministerium für Bildung und Forschung: Press release 148/2007 (July 2007)

[2] Johnson, R.E., Foote, B.: Designing reuseable classes. Journal of Object-Oriented Programming (June/July 1988) `http://www.laputan.org/pub/foote/DRC.pdf`.

[3] Heger, C.: Anwendung formaler Methoden auf Frameworks: Fallstudien zu Salespoint. Diploma thesis, UniBw München (2008) http://www.unibw.de/inf2/Personen-en/Wissen_Mitarbeiter/lothar/publprog.html.

[4] Demuth, B., Schmitz, L., Hussmann, H., Zschaler, S.: Using a framework to teach OOT to beginners. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, Educators' Symposium. (1998)

[5] Demuth, B., Schmitz, L., Hussmann, H., Zschaler, S.: A framework-based approach to teaching OOT: Aims, implementation, and experience. In: 13th Conference on Software Engineering Education and Training. (2000)

[6] Jackson, D.: Software abstractions: logic, language and analysis. 1 edn. MIT Press (2006)

[7] Alloy Homepage: http://alloy.mit.edu/

[8] Duke, R., King, P., Rose, G.A., Smith, G.: The Object-Z specification language: Version 1. Technical Report 91-1, University of Queensland, Department of Computer Science, St. Lucia 4072, Australia (1991) citeseer.ist.psu.edu/duke91objectz.html.

[9] Duke, R., Rose, G., Smith, G.: Object-Z: A specification language advocated for the description of standards. Technical Report 5–6 (1995) citeseer.ist.psu.edu/duke95objectz.html.

[10] Spivey, J.M.: The Z Notation: A Reference Manual. Programming Research Group, University of Oxford. 2 edn. (1992) http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. 1 edn. Addison-Wesley (1995)

[12] Dong, J.S., Duke, R.: Exclusive control within object oriented systems. In: The 18th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'Pacific95), Prentice-Hall (1995) 123–132

[13] Bolton, C.: Using the alloy analyzer to verify data refinement in Z. Electronic Notes in Theoretical Computer Science **137**(2) (2005) 23–44

[14] Estler, H.C., Wehrheim, H.: Alloy as a refactoring checker? Electronic Notes in Theoretical Computer Science **214** (2008)