Formalizing Design Patterns: A Case Study of the Iterator Pattern

Saad Alfoudari and L. J. Steggles.
Department of Computing Science,
University of Newcastle,
Great Britain.

May 13, 2002

Abstract

We investigate using algebraic methods and the support tool Maude to formally specify and reason about the well known iterator design pattern. We begin by specifying instances of the iterator pattern which can be described equationally using Maude. We then develop an abstract specification which we argue captures the essence of the iterator pattern. We conclude by specifying a possible refinement for iterator instances, a so called filter refinement, and by formally proving that this refinement is correct with respect to our abstract specification.

1 Introduction

A design pattern is a description of an established solution for a recurring problem in software engineering (see [12]). They are usually described in outline form, where the description is mostly informal, with some semi-formal descriptions and example code. The aim is to describe the pattern in a uniform way, and also to maintain the level of pattern generality to avoid confining it to narrower than need be problems.

One of the main problems in the description of a design pattern is the possible ambiguity that can arise given the informal descriptions used. In some parts, a design is intentionally vague, leaving the details to the discretion of the pattern user. In other parts, it can simply be an inadequate description, which often results from the informality of the approach used (see [8]). Another problem is understanding the relationships between patterns, which are usually not thoroughly considered. It is normally left for

the user to investigate how different design patterns relate to each other, which can be obvious for some patterns but vague for others.

To solve the problem of ambiguity in design pattern descriptions it has been proposed that so called *formal methods* (see [16]) be used to rigorously describe design patterns. Such formal methods provide specification languages based on a well–defined mathematical semantics and thus allow concise and unambiguous descriptions to be formulated. A range of different approaches for formally describing design patterns can be found in the literature and we briefly survey these now.

In [18] the DisCo method, which is based on temporal logic, is used to formalize the temporal behavior of design patterns. A version of Object Calculus [11] is used in [15] as a semantic framework in which to show that various design patterns can be formally proved to be refinement transformations. In [14] they attempt to add formalization to the intent part of the design pattern description by complementing it with a formalized visual description. Finally, one comprehensive study of formalizing design patterns is the PhD Thesis [9]. This Thesis investigates the use of meta-programming as a formalization tool, in which a design pattern is viewed as a program manipulating algorithm. It is concluded that a declarative formal specification can be more useful, where algebraic or logic formulae are more understandable and less complicated than algorithms. Based on this another approach was adopted using a monadic logic language LePUS [1] which has a textual and graphical form to exploit recurring motives as building blocks.

In this paper we propose a new approach based on using simple algebraic techniques and tools to specify patterns. We aim to add clarity and preciseness to a pattern, without loosing the advantage of the intentional gaps left in the pattern. Our approach attempts to make the formalism accessible to the average programmer or developer through the use of simple equational specifications and a simple, automatable logic for reasoning about the specifications. Indeed, we see the main advantages of our new approach as being a simple algebraic semantics coupled with good tool support. The pattern specification framework we propose should also serve as a tool for investigating the relationship between design patterns and improving our understanding of how patterns interact. The work presented here is a first step toward investigating the applicability of algebraic specifications to design patterns. For this investigation, we use conditional equational specifications and the Maude tool [2]. Maude is a tool that provides a reflective, algebraic language supporting equational and rewriting logic computation (see [7]), and we use this to specify and analyze design patterns. The focus of this paper is the iterator design pattern, which provides an interface for linear

traversal of data structures (see [12]). This simple pattern is an ideal starting case study for our investigation since its characteristics are dependent on the interaction between certain functions within a single class.

We begin by specifying a particular instance of the iterator pattern, which is formulated using Maude. Using this instance and the informal description of the pattern, we derive an abstract specification of the iterator pattern which we argue captures the essence of the pattern and the properties it should posses. Based on the abstract specification, we specify a possible refinement of the iterator pattern, which we call the *filter iterator*. This refinement uses an existing iterator as the basis for constructing a new iterator where only the elements accepted by a given property function are considered (i.e. some elements are filtered out). This refined specification is expected to respect the properties of the iterator pattern, plus some additional properties resulting from the Reification. We conclude by proving that this refinement is a correct iterator pattern with respect to our abstract specification.

The paper is organized as follows. Section 2 provides the general background that will be needed in the later sections. Section 3 develops a formal specification of the iterator pattern. Section 4 introduces a possible refinement of the iterator pattern specified formally, and Section 5 proves that the refinement implements correctly the iterator design pattern. The paper ends with some concluding remarks discussing what was done and our plans for future work.

2 Background

2.1 Algebraic specifications and Maude

Maude is a reflective algebraic language and tool supporting both equational and rewriting logic specification [7]. The system allows for a considerable freedom in defining your own notation and in executing and analyzing specifications. A maude specification is usually composed of modules, which are employed hierarchically by way of importing or including the required modules. Sorts can be defined in a module and operations can be declared over them in the usual way (see [16]). Equations are defined to capture the key attributes associated with the operations and sorts.

The equated specifications written in maude can have one of two possible semantics (see [16] and [17]):

1. Loose Semantics : class of all models (algebras) satisfying the given

equations.

2. Initial Algebra Semantics : a unique (up to isomorphism) model constructed using the possible terms and given equations.

As an example of writing equational specifications using Maude, consider the following simple parameterized specification of a stack. We start by defining a module `Elm`, which is used for specifying the parameter in the specification of `Stack`.

```
(fth Elm is
 sort elm .
 sort elm-err .
 subsort elm < elm-err .
 op error : -> elm-err [ctor].
endfth)
```

This module is a theory specification, as indicated by the keyword `fth`, where `f` signifies full maude, in which the module declares a new data sort `elm`. Another sort `elm-err` is then declared which contains `elm` as a subsort and an extra error constant. Note that this theory module has a loose semantics and will be used as the data parameter in the following parameterized stack specification.

```
(fmod Stack[X :: Elm ] is
 sort stack .

 op empty : -> stack [ctor].
 op push  : elm.X stack -> stack [ctor] .
 op pop   : stack -> stack .
 op top   : stack -> elm-err.X .

 var s s2 : stack .
 var x y : elm.X .

 eq pop(empty) = empty .
 eq pop(push(x,s)) = s .
 eq top(push(x,s)) = x .
 eq top(empty) = error .

endfm)
```

This module declares a sort stack and the four standard stack operations over this sort: empty, push, pop, and top. The empty and push operations are constructors (indicated by the keyword [ctor]). Any instance of stack is created through the constructors, which uniquely define every possible stack value. A non constructor operation like pop doesn't produce a new instance, but rather allows transitions between previously created instances. Two variables are declared which are used to help in defining the equations which are indicated by the prefix *eq*, which capture the key properties of the operations.

As noted, this module uses full maude to enable parameterization, which can usually be distinguished by the parenthesis surrounding the module in the current version of the tool. The passed argument is X which represent a module of type Elm. Throughout the Stack module, sorts from the Elm module are accessed by referencing the module using X after the desired sort, with a dot as a separator.

Notice the difference between the semantics of the two modules: the first module Elm is a theory and has a loose algebraic semantics; the second module is a parameterized functional module which is based on an initial algebra semantics (see [16]).

The following is a concrete module that can be used as an instantiation for X, followed by a view module which provides a mapping to the theory module Elm.

```
(omod INTEGER is
 protecting MACHINE-INT .
 sort errint .
 subsort MachineInt < errint .
 op error : -> errint [ctor].
endom)

(view ElmToInt from Elm to INTEGER is
 sort elm to MachineInt .
 sort elm-err to errint .
 op error to error .
endv)
```

The module INTEGER attempts to make a concrete module that can be instantiated for Elm. Note the use of protecting, which imports the MACHINE − INT module. The MACHINE − INT module provides the sort MachineInt which is used in INTEGER.

Unlike normal variable instantiations, a module instantiation needs to know which part of the module corresponds to which part in the theory. So, ElmToInt provides a mapping for example from the sort elm − err in Elm to the sort errint in INTEGER.

The following is a typical session using these modules. First, we input the modules into the system. Note that the user input comes after Maude>.

```
Maude> in Elm
rewrites: 609 in 20ms cpu (30ms real) (30450 rewrites/second)

Introduced theory: Elm
Maude> in Stack
rewrites: 1895 in 0ms cpu (10ms real) (~ rewrites/second)

Introduced module: Stack
Maude> in IntElm
rewrites: 1914 in 0ms cpu (10ms real) (~ rewrites/second)

Introduced module: INTEGER
rewrites: 332 in -10ms cpu (0ms real) (~ rewrites/second)

Introduced view: ElmToInt
```

Note that the IntElm file included two modules. The modules do not need to have a name corresponding to the file name, which only exists as a convenience. Each of the loaded files have the extension .maude, which need not be specified.

To instantiate INTEGER to Stack, we need to declare a module importing Stack with the desired instantiation.

```
Maude> (fmod prog is
protecting Stack[ElmToInt] .
endfm)
```

This will produce:

```
rewrites: 4458 in 10ms cpu (20ms real) (445800 rewrites/second)

Introduced module: prog
```

Hence, we have a new module prog with the instantiated stack. We can now try the instantiated stack with some experimental values.

```
Maude> (rew push(20,(push(30,empty))) .)
rewrites: 220 in 0ms cpu (10ms real) (~ rewrites/second)
rewrite in prog : push (20,push(30,empty)) .
result stack : push(20,push(30,empty))

Maude> (rew push(10,pop(push(20,push(30,empty)))) .)
rewrites: 256 in 0ms cpu (10ms real) (~ rewrites/second)
rewrite in prog : push(10,pop(push(20,push(30,empty)))).
result stack : push(10,push(30,empty))
```

For further information about the Maude system, the reader can consult [7] and [2].

2.2 Reasoning about specifications

The following are the four proof rules for equational logic that will be used throughout the document (see [17]). Let Σ be an arbitrary S–signature and let x be an S-sorted collection of variables.

Reflexitivity Rule for any term t over Σ and x

$$\frac{}{t = t}$$

Symmetry Rule for any terms t, t' over Σ and x

$$\frac{t = t'}{t' = t}$$

Transitivity Rule for any term t_1, t_2, t_3 over Σ and x

$$\frac{t_1 = t_2; t_2 = t_3}{t_1 = t_3}$$

Substitution Rule for any terms t, t' over Σ and x

$$\frac{t[x] = t'[x], t_1 = t_2}{t[t_1/x] = t'[t_2/x]}$$

where t/x stands for substituting t in place of x.

Note that the above equational logic is sound and complete for equational specifications (see [16]). We also have an inference rule for reasoning about conditional equations:

Conditional Equation Rule let e_1, e_2 be equations over Σ and x

$$\frac{e_1 \Rightarrow e_2; e_1}{e_2}$$

We will use the above (conditional) equational logic to reason about our design pattern specifications (see Section 5). We let $E \vdash t = t'$ denote that the equation $t = t'$ is provable from a set of (conditional) equations E using the above inference rules.

2.3 Design Patterns

A pattern [4] is a problem/solution pair, where a recurring problem requires the same basic solution, which are used and proven, though differing in details according to the context. The major aim of using patterns is to catalog known methods for solving recurring problems, and provide a terminology for talking about solutions. They also allow methods known by experts to be passed on to novices.

A distinction could be made between a pattern and a design pattern in that the latter is concerned with communicating components, and is independent of the implementation language. Other types of patterns could be language dependent, or of a more general nature (see [6]). The focus of this work is on design patterns, where the pattern is not too abstract as a notion to be formally specified. However, the approach could be generalized outside of design patterns where appropriate.

The basic parts of a design pattern is the *pattern name*, the *problem* the pattern addresses, the *solution* used for solving the problem, and the *consequences* of using the solution [12].

The pattern *name* is essential in cataloging, since a pattern could be known by different names to different parties. Without having a standardized name, a lengthy description has to be clearly made, to make sure that what the two parties are talking about is the same thing. Also, if two parties use the same name for two relatively different patterns, confusion might result, which can only be resolved by elaborating on the respective patterns.

The *problem* part describes when and where the pattern is used. This includes the type of problem being addressed, and the conditions that the solution requires to exist.

The *solution* part describes the design for solving the problem, which can include the structure and behavior that should be used, and the way the different parts collaborate for solving the problem.

The *consequences* describe the results and trade-offs of using the solution. This part is essential for determining whether the pattern is to be used or not.

A design pattern is presented as an outline that includes certain fixed parts, plus extra complementary parts that gives further elaboration and description of the pattern. The following are the part division used in [12]. Other different divisions exist (see [3] and [6]).

Name and classification: Name of the pattern, and the group it belongs to (one of creational, structural, or behavioral.)

Intent: Describes the intent and purpose of using the pattern (what does it do.)

Also known as: Other names which the pattern is known by.

Motivation: an example of a problem and how the pattern is used to sole the problem.

Applicability: Situations and conditions in which the pattern can be applied.

Structure: A graphical representation describing the solution structure.

Participants: A brief description of the responsibilities of the pattern solution elements.

Collaborations: how participants collaborate in performing their responsibilities to achieve the goal.

Consequences: The trade-offs of the solution.

Implementation: hints and notes regarding the implementation, which could address certain specific languages.

Sample Code: code fragments illustrating how the pattern might be used.

Known uses: examples of where the pattern is used in real systems.

Related Patterns: Other patterns that can have a relation to this pattern.

It could be seen that most of these parts are an expansion of the four basic parts. For example, motivation and applicability illustrate the problem the pattern addresses. Some serve two basic parts, like implementation.

3 Specifying the Iterator Pattern

In this section, we will introduce the iterator pattern, and attempt to formalize a certain implementation of it. The formalization of a concrete implementation should help in making a generalized formal specification, which should axiomatize the main and distinguishing attributes of the iterator pattern, without it being specific to a particular implementation.

3.1 The Iterator Pattern

The iterator pattern is a well-known method for traversing elements sequentially in an aggregate object. The pattern aims at creating a way to access the elements within the aggregate without exposing the internal structure of the aggregate. The pattern deals with an aggregate object, containing the elements to be traversed, and an iterator object, which is used in accessing the elements in the aggregate.

An aggregate contains a function **create()**, which is responsible for creating the iterator object to traverse its elements. The process starts by a call to **create()**, which creates the iterator, passing the necessary information for traversing the aggregate to the iterator. An aggregate can be a tree structure, a stack, a list, or any other aggregate type. The iterator has a fixed interface, which allows the sequential traversal of the aggregate elements.

After creating the iterator, a pointer will be set to the first element. The current element is fetched by a call to **current()**. To proceed to the next element, a call to **next()** is made, which sets the iterator pointer to the next element it should access, which in turn is fetched by a call to **current()**. This is done repeatedly to access succeeding elements. The client can detect whether there are further elements to be accessed by calling **isDone()**, which returns true if the iterator pointer has accessed the last element and there is no remaining elements to access, and returns false otherwise. The operator **first()** can be called to reset the pointer in the iterator to the first element. The interface is presented in Figure 1.

first() set the current item to be the first element.

next() move to the next element.

isDone() indicates if there is a current item or if the iterator reached the end of its elements.

current() returns the current element the iterator is pointing to.

Operator	*type*
first()	Iterator $\rightarrow$ Iterator
next()	Iterator $\rightarrow$ Iterator
isDone()	Iterator $\rightarrow$ Boolean
current()	Iterator $\rightarrow$ data

Figure 1: Iterator Interface

The iterator pattern provides a unified interface for the elements of an aggregate, and it can also enable different traversals of an aggregate by providing different iterators which share the same iterator interface.

The specification aims for defining the properties which the functions of the interface should maintain. A second aim (see section 4) is to provide a specification for which different iterators could be defined based on an existing iterator and a certain property, where the specification complies with the requirement of how an iterator should behave.

For fulfilling the first aim, which is specifying the requirements of an iterator, general properties are made for the iterator functions and how they relate to each other. This is done by the developer, who will make the specification of the iterator pattern. In this part, the aggregate structure which the iterator traverses is mostly an unknown to the developer, and the user of the pattern has to specify an initial iterator that traverses the elements of the aggregate, which he wishes to apply the pattern on.

General properties of the iterator are provided by the developer, in which the user has to make sure that his specification of the iterator over the aggregate complies with the general properties of the iterator. Some properties are not specified in the formal specification, but the user is encouraged to maintain them informally. For example, that the iterator specified spans all the elements of the aggregate, and that each element is visited only once. Such condition can not be algebraically specified, since the aggregate structure is kept a mystery for the developer, and is only known through the iterator, in which a base iterator would be provided by the user.

3.2 Specifying An Instance of the Iterator Pattern

Before attempting to build a general iterator specification, we start by specifying a concrete instance that implements the iterator. Then, we attempt to construct a general iterator specification, which should capture the general attributes without being specific to a certain implementation.

We take the stack example declared earlier and make a specification for a simple iterator over the stack. In this example, we are only interested in providing access to all the elements of a stack as plainly and straightforwardly as possible while conforming to the iterator pattern attributes, which are informal up to this point.

```
(omod StackIterator [E :: Elm] is
 protecting Stack[E] .
 sort StackIt Natural .

 op 0 : -> Natural [ctor] .
 op succ : Natural -> Natural [ctor] .

 op <__>  : stack Natural -> StackIt [ctor] .
 op _.create : stack  -> StackIt .
 op _.first  : StackIt -> StackIt .
 op _.next   : StackIt -> StackIt .
 op _.current : StackIt -> elm-err.E .
 op _.isDone : StackIt -> Bool .

 var s : stack .
 var n : Natural .
 var sit   : StackIt .
 var x : elm.E .

 eq s .create = < s 0 > .

 eq < s n > .first = < s 0 > .

 eq < s n > .next = < s succ(n) > .

 eq < push(x,s) 0 > .current = x .

 eq < empty n > .current = error .

 eq < push(x,s) succ(n) > .current = < s n > .current .

 eq < push(x,s) 0 > .isDone = false .

 eq < empty n > .isDone = true .
```

```
 eq < push(x,s) succ(n) > .isDone = < s n > .isDone .

endom)
```

In the StackIterator module, a simple data structure is used to hold the state of the iterator. This data structure is `<__>`, where in the place of the first `_` a stack would exist, paired with a Natural value in the second place. The variable `s` is a stack, and `n` is a natural member variable which represents a pointer to where the iterator is currently pointing at.

3.3 Abstract Specification

The following is an abstract specification of the iterator pattern which axiomatize its characteristic properties. The IteratorTheory specification is a loose specification with a constructor condition placed upon it (note the type Bool is automatically imported in the loose specification with its fixed initial algebra semantics.) This allows us to formalize the fixed parts of the pattern while leaving the intended "holes" in the pattern.

```
(oth IteratorTheory is

 including Elm .
 sort Agr .
 sort IT .

 op _.create : Agr -> IT [ctor].
 op _.first  : IT -> IT .
 op _.next   : IT -> IT [ctor].
 op _.current : IT -> elm-err .
 op _.isDone : IT -> Bool .

 var s : Agr .
 var it it1 it2 : IT .

 eq  s .create .first = s .create .

 eq  it .next .first = it .first .

 ceq it .next .isDone = true   if it .isDone == true  .
```

```
 ceq it .current = error if it .isDone == true .

endoth)
```

There are four equations which define the key properties of the operations defined over the iterator sort. This is a characteristic of the iterator pattern, which is that it is defined by certain operations over one sort. Other patterns might involve more sorts interacting with each other.

The first two equations axiomatize the Behavior of .first operation. The first equation establishes that .first yields the same iterator in an initial state. The second equation establishes that applying .first on a normal iterator, will unwind all .next operations, which in effect will return the iterator to its initial state.

Note that the specification is unable to define what .first means in relation to the source aggregate structure directly, since the later type is an unknown. This results from an intended hole in the pattern which the user must fill.

The third and fourth equations axiomatize aspects of the behavior of the iterator when it's at the end of its traversal. The third equation captures the property that applying .next after reaching the end of the iterator does not affect the state of the iterator of being at the end. The fourth equation enforces that the iterator will not return any valid element when at the end of the traversal.

An extra condition is placed on the loose specification when declaring .create and .next. Both operations are declared as constructors using the key word ctor, and as such, they become the generators of the sort IT (i.e. .create and .next must generate all values of type IT, see [16]). This property of IT enable us to use structural induction to prove properties about iterators (see section 5).

3.4 Proving an Instance Satisfies the Iterator Pattern

We can use Maude to help us in proving that the StackIterator satisfies the properties stated for the iterator pattern by the abstract iterator specification.

We first select the module which we want to test .

```
(select StackIterator .)
rewrites: 18 in -10ms cpu (0ms real) (~ rewrites/second)
```

For testing, we will need to use variables. We can either use the variables within the module StackIterator, in which we can issue the command `(show module .)` to know what variables are declared in the module, or simply make a small module that imports the relevant module and declares the desired variables.

```
(omod prog is
pr StackIterator .
pr Elm .
var s : stack .
var x : elm.E .
var n : Natural .
endom)
rewrites: 9650 in 20ms cpu (30ms real) (482500 rewrites/second)

Introduced module: prog
```

We can then proceed to use it for verifying the correctness of the iterator properties.

1. `eq s .create .first = s .create .`

   The test is very direct for this property. We simply apply an equality test over the desired pattern property.

   ```
   Maude> (rew s1 .create .first == s1 .create .)
   rewrites: 201 in 0ms cpu (0ms real) (~ rewrites/second)
   rewrite in prog : s1 .create .first == s1 .create .
   result Bool : true
   ```

   Maude confirms that the two sides are equivalent for any stack.

2. `eq it .next .first = it .first .`

   Similarly to the previous property, we can test the property directly by applying the equality test.

   ```
   Maude> (rew < s1 n > .next .first == < s1 n > .first .)
   rewrites: 234 in 0ms cpu (10ms real) (~ rewrites/second)
   rewrite in prog : < s1 n > .next .first == < s1 n > .first .
   result Bool : true
   ```

3. `ceq it .next .isDone = true  if it .isDone == true  .`

   We apply the test directly by replacing it with the case in which applying .isDone returns true. It is straight forward to show we need only consider terms `< empty n >` of sort StackIterator.

```
Maude> (rew < empty n > .next .isDone == true .)
rewrites: 211 in -10ms cpu (0ms real) (~ rewrites/second)
rewrite in prog : < empty n > .next .isDone == true .
result Bool : true
```

4. `ceq it .current = error  if it .isDone == true  .`

   We similarly apply the test directly on the corresponding case.

```
Maude> (rew < empty n > .current == error .)
rewrites: 203 in -10ms cpu (0ms real) (~ rewrites/second)
rewrite in prog : < empty n > .current == error .
result Bool : true
```

5. Finally, we need to show that sort StackIterator is generated by .create and .next (as stated in the abstract specification.) This is straight forward to do and for brevity is omitted here.

The task of the developer is to provide the specifications for the parts which are fixed in the pattern. Some effort should be done to minimize the proving burden on the user by specifying as much fixed structure in the pattern, and only leaving the intentional gaps in a pattern to be provided by the user. Those gaps of course might have some constraints on how they should be filled.

4 Refining the Iterator Pattern

The Iterator pattern can be refined into several variations, each having more distinguishing attributes, while maintaining the original general iterator pattern requirements. For enabling the refinement to attain extra distinguishing attributes, the refinement specification would be based upon other iterators. As such, it would be assumed that an iterator which complies with the iterator requirements is provided, and based upon it, another iterator is specified which also complies with the requirements of the iterator pattern.

The developer here takes the burden of proving that the resulting iterator is compliant with the requirements. The main difference between the refined iterator and the provided one is that the new iterator can give another order sequencing of the elements the provided iterator traverses according to the following suggested possibilities:

- *Different Ordering method*, which can be done by providing an order function that tells which elements are after which other elements. A simple example is providing a function where it compares two elements and returns true if the second is greater than the first.
- *Partial Iterator*, which are elements that pass through a filter that accepts elements with certain properties. For example, only accepting the elements that are greater than 7.
- *Iterator Re-sequencing*, which can be done by providing a sequence containing the elements of the original iterator. For example, an iterator with the ordered elements [a,b,c,d,e], the sequence [2,4,2] can be used to generate another iterator that traverses the elements as follows [b,d,b], where each number in the sequence is the order of the desired element from the old traversal.

We choose to implement a refinement based on the partial iterator traversal. We achieve this by introducing a specification for a refinement, which given an iterator along with a filter property, produces a new filtered iterator, as specified by the new theory. We start by defining a new loose specification to represent an iterator and a corresponding filter property.

```
(oth IteratorPropTheory is

 including IteratorTheory .

 op _.prop : IT -> Bool .

endoth)
```

The following module specifies the filtered iterator based on an iterator and a property, which are passed to it as a combined parameter.

```
(omod it-filter [ P :: IteratorPropTheory ] is

 sort FIT[P] .
```

```
 op capsule_    : IT.P -> FIT[P] [ctor] .
 op _.create : IT.P -> FIT[P] .
 op _.gran : IT.P -> IT.P .

 op _.first   : FIT[P] -> FIT[P] .
 op _.next    : FIT[P] -> FIT[P] .
 op _.current : FIT[P] -> elm-err.P .
 op _.isDone  : FIT[P] -> Bool .

 var it : IT.P .


 eq it .create = ((capsule(it)).first) .
 eq (capsule(it)).first = (capsule(it .first .gran)).

 ceq (it) .gran = (it) if (it .prop == true)
               or (it .isDone == true) .

 ceq (it) .gran = (it .next) .gran
   if (it .prop or it .isDone) == false .

 eq (capsule(it)).next = (capsule(it .gran .next .gran)) .

 eq (capsule(it)).current = it .gran .current .

 eq (capsule(it)).isDone = it .gran .isDone .

 eq (capsule(it)) = capsule(it .gran) .

endom)
```

The it − filter specification specifies a filtered iterator, which only accepts elements with certain properties and ignores other elements. The desired property is specified through .prop in IteratorPropTheory, which should return true for acceptable elements.

A typical application can start by defining the module that includes the iterator then adding to it the desired property. The it − filter specification takes in as a parameter the iterator and the property, and defines a new

iterator based on them. This is achieved by wrapping the initial iterator with .capsule which acts in using the iterator with as a source for a new iterator with some new behavior. This new behavior is summarized in ignoring all elements that do not satisfy the property, which was taken as a parameter with the iterator, while maintaining the satisfied elements. To achieve this, an operator .gran is used to discard the undesirable elements. It can be noted that capsule constantly uses .gran when translating certain operations over it to operations over the iterator that it wraps.

In all the operations declared over capsule, the operation .gran is used to guarantee the behavior of the filtered iterator. The last equation reasserts this relationship and ensures that all intermediate filter iterator states between desired elements are combined into one. This prevents multiple instances representing the same iterator state.

```
(omod StackIterator1 is
 pr StackIterator[ElmToInt] .
 op _.highvalue : StackIt -> Bool .
 var x : StackIt .
 ceq x .highvalue = x .current > 5 if x .isDone == false .
 ceq x .highvalue = false if x .isDone == true .
endom)
```

The above module combines the specification of an instantiated stack iterator with an extra operation that works as a selector operation. Such a module would require a view which maps the elements of IteratorPropTheory to appropriate counterparts.

```
(view Prop1 from IteratorPropTheory
to StackIterator1 is
--- Elm Theory .
 sort elm to MachineInt .
 sort elm-err to errint .

--- Iterator Theory .
 sort Agr to stack .
 sort IT to StackIt .
 op _.create to _.create .
 op _.first to _.first .
 op _.next to _.next .
 op _.current to _.current .
```

```
 op _.isDone to _.isDone .

--- Prop Theory .
 op _.prop to _.highvalue .
endv)
```

Now, the view only needs to be passed to $\mathsf{it-filter}$ to compose a filtered iterator.

These are not the only iterators that could be specified, for example, we might want to specify an iterator that has a different order of traversal, but it shows a major category of iterators that can be specified. The purpose of this paper is limited to investigating such a category of iterators through the suggested specification, and not to provide a full coverage of the specification of iterator pattern categories.

Given the previous specification of $\mathsf{it-filter}$, if the user of the specification assures that the input iterator to $\mathsf{it-filter}$ complies with the abstract iterator specification, and the developer insures that based on a valid iterator, the resulting refined iterator is also a valid iterator, then it can be guaranteed that all resulting iterators thereof using this filter are valid iterators (see Figure 2).

5 Correctness Proof for the Refinement

To prove the correctness of the refinement, we will first extract the basic axioms of the iterator theory. It should be proven that these axioms hold for the filter iterator implementation.

For ease of discussion. we extract the axioms from the abstract iterator specification `IteratorTheory`.

Definition 1 The iterator theory has the following axioms. Let $\mathrm{x} \in \mathcal{A}\mathrm{gr}$, and $\mathsf{it} \mathrm{in} \mathcal{X}_{\mathrm{IT}}$ be variables.
Axiom 1.1 x .create .first = x .create
Axiom 1.2 it .next .first = it .first
Axiom 1.3 it .isDone = true $\Rightarrow$ it .next .isDone = true
Axiom 1.4 it .isDone = true $\Rightarrow$ it .current = error

□

In order to use the extra constructor semantics information for iterators, captured in iterator theory, we use the following structural induction infer-

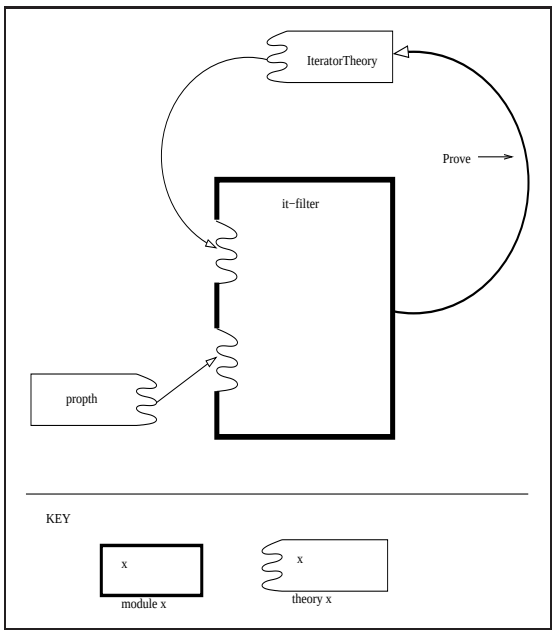

Figure 2: Overview of Filter Refinement

ence rule.

Definition 2 (Structural Induction) Let E be an (conditional) equational theory over the iterator signature and let t, t' be two terms of the same sort. Let $x \in X_{IT}$, $s \in X_{Agr}$ and c be a new constant symbol of sort IT not already occurring in the given signature.

$$\frac{\begin{array}{c}\mathsf{E} \vdash t[x/s.create] = t'[x/s.create] \\ \mathsf{E} \cup \{t[x/c] = t'[x/c]\} \vdash t[x/c.next] = t'[x/c.next]\end{array}}{\mathsf{E} \vdash_{ind} t = t'}$$

It is understood that a fresh constant c of sort IT is introduced each time this rule is used. □

The above rule captures structural induction proofs for iterators based on their construction. It is straight forward to show this rule is sound for the abstract iterator specification since the additional constructor constraint ensures that .create and .next generate the sort IT. We will use this rule to prove the following lemma, which states that applying the operation .first

twice yields the same result as applying it once.

Lemma 3 For $\mathsf{it} \in X_{\mathrm{IT}}$, we have that

$$\mathsf{IteratorTheory} \vdash_{\mathrm{ind}} \mathsf{it.first.first} = \mathsf{it.first}$$

is provable using iterator theory, equational logic and structural induction.

Proof : Let $\mathsf{it} \in X_{\mathrm{IT}}$, and $\mathsf{x} \in X_{\mathcal{A}\mathrm{gr}}$ be variables.

1 - **from**

1.1 - $\mathsf{it.first} = \mathsf{it.first}$

by reflection

1.2 - $\mathsf{x.create.first} = \mathsf{x.create}$

by axiom 1.1

infer
$\mathsf{x.create.first.first} = \mathsf{x.create.first}$

by substitution(1.1, 1.2)

2 - **from** $\mathsf{c.first.first} = \mathsf{c.first}$ (Induction hypothesis)

2.1 - $\mathsf{c} = \mathsf{c}$

by reflection

2.2 - $\mathsf{c.next.first} = \mathsf{c.first}$

by substitution(Axiom 1.2, 2.1)

2.3 - $\mathsf{c.first} = \mathsf{c.first}$

by reflection

2.4 - $\mathsf{c.first.next.first} = \mathsf{c.first.first}$

by substitution(Axiom 1.2, 2.3)

2.5 - $\mathsf{c.next.first} = \mathsf{c.next.first}$

by reflection

2.6 - $\mathsf{c.next.first.next.first} = \mathsf{c.next.first.first}$

by substitution(Axiom 1.2, 2.5)

2.7 - $\mathsf{it.next.first} = \mathsf{it.next.first}$

by reflection

2.8 - $\mathsf{c.next.first.next.first} = \mathsf{c.first.next.first}$

by substitution(2.7, 2.2)

2.9 - $\mathsf{c.first} = \mathsf{c.first.first}$

by symmetry(assumption 2)

2.10 - $\mathsf{c.next.first} = \mathsf{c.first.first}$

by transitivity(2.2, 2.9)

2.11 - $\mathsf{c.first.first} = \mathsf{c.first.next.first}$

by symmetry(2.4)

2.12 - $c.next.first = c.first.next.first$
by transitivity(2.10, 2.11)
2.13 - $c.first.next.first = c.next.first.next.first$
by symmetry(2.8)
2.14 - $c.next.first = c.next.first.next.first$
by transitivity(2.12, 2.13)
2.15 - $c.next.first = c.next.first.first$
by transitivity(2.14, 2.6)

infer
$c.next.first.first = c.next.first$
by symmetry(2.15)

infer
$it.first.first = it.first$
by structural induction(1,2)
□

For convenience, we now extract the axioms of the filter iterator which we then prove correctly implement the abstract iterator specification.

Definition 4 The Filter iterator specification has the following axioms. Let $it \in X_{IT}$ be a variable.

Axiom 4.1 it .create = capsule(it) .first
Axiom 4.2 capsule(it) .first = capsule(it .first .gran)
Axiom 4.3 (it .prop or it .isDone)=true ⇒ it .gran = it
Axiom 4.4 (it .prop or it .isDone)=false ⇒ it .gran = it .next .gran
Axiom 4.5 capsule(it).next = capsule(it .gran .next .gran)
Axiom 4.6 capsule(it).current = it .gran .current
Axiom 4.7 capsule(it).isDone = it .gran .isDone
Axiom 4.8 capsule(it) = capsule(it .gran) □

Each axiom of the abstract iterator theory must now be investigated to examine whether it is valid for filter iterator specification or not. In this way, we can check that the filter specification always results in a valid iterator for any given parameter.

In the filter iterator specification, we introduced .gran over an existing iterator, which serves in adding a property to the new iterator to "filter out" elements. The operator .gran keeps introducing .next to the iterator until the iterator reaches an element which satisfies .prop or until it reaches the

end of the iterator, (applying .isDone evaluates to true.) It is assumed that the iterator does terminate at some element and that there is no infinite loop within the iterator. As such, it.gran can be viewed as $it.next^n$, where the n^{th} .next applied on it would either satisfy .prop or .isDone, where applying .prop or .isDone would return false if applied on $it.next^k$, $0 \leq k < n$. If the iterator does not terminate, then .gran could test for a property which is not satisfied by any of the succeeding elements of the iterator, and hence result in the infinite application of .next over the iterator.

The following defines a short hand notation for sequences of applications of .next to an iterator term $it : IT$.

Definition 5 Let $i \in \mathbb{N}$, $i \geq 0$, and let $it : IT$ be an iterator term. Then

$$it.next^0 = it,$$
$$it.next^{i+1} = it.next^i.next$$

□

The following lemma shows that applying .first on an iterator yields the same iterator state, regardless of how many .next operations are applied to the iterator.

Lemma 6 Let $i \geq 0$ and $it \in X_{IT}$ be an iterator variable. Then:

$$IteratorTheory \vdash it.next^i.first = it.first$$

Proof : By induction on $i \in \mathbb{N}$.

Base Case Let $i = 0$
Holds trivially since $it.next^0 = it$.
Inductive Case Let $i = k+1$, for some $k \in \mathbb{N}$, and suppose $it.next^k.first = it.first$ (Inductive hypothesis).

1 - $x.next.first = x.first$
by Axiom 1.2

2 - $it.next^k = it.next^k$
by reflection

3 - $it.next^k.next.first = it.next^k.first$
by substitution(1, 2)

infer

$it.next^{k+1}.first = it.first$

by transitivity(3, induction assumption)

□

The following captures what it means to "unwind" the $.gran$ operator into applications of $.next$ in order to skip unwanted elements.

Lemma 7 Let $it \in X_{IT}$, and suppose there exists $i \in \mathbb{N}$ such that for all $j \leq i$, we have ($it.next^j.isDone$ or $it.next^j.prop$) = $false$(initial assumption). Then we have

$$it - filter \vdash it.gran = it.next^{i+1}.gran$$

Proof : By induction on $i \in \mathbb{N}$.

Base Case Let i=0

$it.gran = it.next^1.gran$

by CondEq(Axiom 4.4, initial assumption)

Inductive Case Suppose $i = j+1$, for some $j \in \mathbb{N}$ and $it.gran = it.next^i.gran$ (induction assumption)

1 - $it.next^i = it.next^i$

by reflection

2 - $it.gran = it.next.gran$

by CondEq(Axiom 4.4, initial assumption)

3 - $it.next^i.gran = it.next^i.next.gran$

by substitution(2, 1)

infer

$it.gran = it.next^{i+1}.gran$

by transitivity(induction assumption, 3)

□

The next lemma uses the previous two lemmas to show that applying $.first$ is unaffected by preceding it with $.gran$.

Lemma 8 Let $it \in X_{IT}$, and suppose there exists $n \in \mathbb{N}$, such that $it.next^n.isDone = true$ or $it(.next)^n.prop = true$, then

$$it - filter \vdash it.gran.first = it.first$$

Proof : By cases on $n \in \mathbb{N}$.

Case 1 Let $n = 0$, then $it.next^n = it.next^0 = it$, and suppose either $it.isDone = true$ or $it.prop = true$. Let $x \in X_{IT}$ be a variable.

1 - $it.gran = it$
by CondEq(Axiom 4.3, assumption)

2 - $x.first = x.first$
by reflection

infer

3 - $it.gran.first = it.first$
by substitution(2, 1)

Case 2 Let $n > 0$, $n = l+1$ where ($it.next^n.isDone = true$ or $it.next^n.prop = true$)(assumption H1). Then we can assume without loss of generality that ($it.next^m.isDone = false$ and $it.next^m.prop = false$,) for $0 \leq m < n$.

1 - $it.gran = it.next^{l+1}.gran$
by Lemma 7 using case 2 assumption

2 - $it.next^n = it.next^n$
by reflection

3 - $(it.next^n.proporit.next^n.isDone) = true \Rightarrow it.next^n.gran = it.next^n$
by substitution(Axiom 4.3, 2)

4 - $it.next^n.gran = it.next^n$
by CondEq(3, assumption H1)

5 - $it.gran = it.next^n$
by transitivity(1, 3)

6 - $x.first = x.first$
by reflection

7 - $it.gran.first = it.next^n.first$
by substitution(6, 5)

8 - $it.next^n.first = it.first$
by Lemma 6

infer

$it.gran.first = it.first$
by transitivity(7, 8)

□

To prove that the abstract Iterator axioms hold for the filter iterator, we require to show that all terms of sort FIT are represented through the

operator $\mathsf{capsule}$ (i.e. $\mathsf{capsule}$ can be seen as a constructor for sort FIT). This is necessary since all operations over FIT are defined using the $\mathsf{capsule}$ operator. In the sequel we refer to any term $\mathsf{fit} : \mathsf{FIT}$ which does not contain a variable of type FIT as a *FIT ground term*.

Lemma 9 For any FIT ground term $\mathsf{fit} : \mathsf{FIT}$, there exists a term $\mathsf{it} : \mathsf{IT}$ such that

$$\mathsf{it} - \mathsf{filter} \vdash \mathsf{fit} = \mathsf{capsule}(\mathsf{it})$$

Proof : By structural induction on the construction of terms of sort FIT.
Base Cases : $\mathsf{IT} \rightarrow \mathsf{FIT}$.
Case 1: Consider $\mathsf{x.create}$: Let $\mathsf{x} \in X_{\mathsf{IT}}$ be a variable.

1 - $\mathsf{x} = \mathsf{x}$
by reflection

2 - $\mathsf{it.create} = \mathsf{capsule}(\mathsf{it.first.gran})$
by transitivity(Axiom 4.1, Axiom 4.2)

infer
$\mathsf{x.create} = \mathsf{capsule}(\mathsf{x.first.gran})$
by substitution(2, 1)

Case 2: Consider $\mathsf{capsule}$:
$\mathsf{capsule}(\mathsf{it}) = \mathsf{capsule}(\mathsf{it})$
by reflection

Inductive Cases : $\mathsf{FIT} \rightarrow \mathsf{FIT}$.
Case 1: Consider $\mathsf{fit.next}$ for some term $\mathsf{fit} : \mathsf{IT}$:
from $\mathsf{fit} = \mathsf{capsule}(\mathsf{it})$, for some term $\mathsf{i} : \mathsf{IT}$(Hypothesis I1)

1 - $\mathsf{x.next} = \mathsf{x.next}$
by reflection

2 - $\mathsf{fit.next} = \mathsf{capsule}(\mathsf{it}).\mathsf{next}$
by substitution(1, hypothesis I1)

infer
$\mathsf{fit.next} = \mathsf{capsule}(\mathsf{it.gran.next.gran})$
by transitivity(2, Axiom 4.5)

Case 2: Consider $\mathsf{fit.first}$ for some term $\mathsf{it2} : \mathsf{IT}$:
from $\mathsf{fit} = \mathsf{capsule}(\mathsf{it})$, for some term $\mathsf{it} : \mathsf{IT}$

1 - $\mathsf{x.first} = \mathsf{x.first}$
by reflection

2 - $fit.first = capsule(it).first$

by substitution(1, assumption)

infer
$fit.first = capsule(it.first.gran)$

by transitivity(2, Axiom 4.2)

□

Now, we need to show that the Filter Iterator specification satisfies the generator restriction placed on iterators by the abstract iterator specification (i.e. that .create and .next are constructors.)

Theorem 10 For any FIT ground term $fit : FIT$, there exists $i \in \mathbb{N}$ and a term $it : IT$ such that

$$it - filter \vdash fit = it.create.next^i$$

Proof : By Lemma 9, we need only consider terms of the form $capsule(it)$, for $it : IT$. We now use structural induction on $it : IT$ to prove the result.

Base Case Let $it = x.create$, for $x \in X_{Agr}$, and let $y \in X_{IT}$

1 - $capsule(x) = capsule(x)$

by reflection

2 - $capsule(x.create.first) = capsule(x.create)$

by substitution(1, Axiom 1.1)

3 - $capsule(x.create) = capsule(x.create.first)$

by symmetry(2)

4 - $x.create.first = x.create.first$

by reflection

5 - $capsule(x.create.first) = capsule(x.create.first.gran)$

by substitution(Axiom 4.8, 4)

6 - $capsule(x.create) = capsule(x.create.first.gran)$

by transitivity(3, 5)

7 - $x.create = x.create$

by reflection

8 - $capsule(x.create).first = capsule(x.create.first.gran)$

by substitution(Axiom 4.2, 7)

9 - $x.create.create = capsule(x.create).first$

by substitution(Axiom 4.1, 7)

10 - $x.create.create = capsule(x.create.first.gran)$

by transitivity(9, 8)

11 - $capsule(x.create.first.gran) = x.create.create$
by symmetry(10)

infer
$capsule(x.create) = x.create.create$
by transitivity(6, 11)

Inductive Case Assuming $capsule(it) = it.create.next^i$, prove that $capsule(it.next) = it.create.next^j$, for some $i, j \in \aleph$
Case 1 - **from** $prop(it.current) = false$ and $it.isDone = false$

1.1 - $it.gran = it.next.gran$
by CondEq(Axiom 4.4, assumption)
1.2 - $capsule(it) = capsule(it)$
by reflection
1.3 - $capsule(it.gran) = capsule(it.next.gran)$
by substitution(1.2, 1.1)
1.4 - $capsule(it) = capsule(it.next.gran)$
by transitivity(Axiom 4.8, 1.3)
1.5 - $it.next = it.next$
by reflection
1.6 - $capsule(it.next) = capsule(it.next.gran)$
by substitution(Axiom 4.8, 1.5)
1.7 - $capsule(it.next.gran) = capsule(it)$
by symmetry(1.4)
1.8 - $capsule(it.next) = capsule(it)$
by transitivity(1.6, 1.7)

infer
$capsule(it.next) = it.create.next^i$
by transitivity(1.8, Induction assumption)
Case 2 - **from** $prop(it.current) = true$ or $it.isDone = true$

2.1 - $it.gran = it$
by CondEq(Axiom 4.3, assumption)
2.2 - $it.gran.next = it.gran.next$
by reflection
2.3 - $capsule(it.gran.next) = capsule(it.gran.next.gran)$
by substitution(Axiom 4.8, 2.2)
2.4 - $capsule(it.gran.next.gran) = capsule(it.gran.next)$
by symmetry(2.3)
2.5 - $capsule(it).next = capsule(it.gran.next)$
by transitivity(Axiom 4.5, 2.4)

2.6 - $x.next = x.next$

by reflection

2.7 - $it.gran.next = it.next$

by substitution(2.6, 2.1)

2.8 - $capsule(it) = capsule(it)$

by reflection

2.9 - $capsule(it.gran.next) = capsule(it.next)$

by substitution(2.8, 2.7)

2.10 - $capsule(it).next = capsule(it.next)$

by transitivity(2.5, 2.9)

2.11 - $capsule(it.next) = capsule(it).next$

by symmetry(2.10)

2.12 - $capsule(it).next = it.create.next^{i}.next$

by substitution(2.6, Induction assumption)

infer

$capsule(it.next) = it.create.next^{i}.next$

by transitivity(2.11, 2.12)

□

In the sequel, let $it \in X_{IT}$. We may now show that $it - filter$ satisfies the iterator axioms. In order to apply the iterator axioms to the filter specification, we need to identify which sorts in the theory $it - filter$ will act as the sorts Agr and IT. Clearly we will take sort $IT.P$ as the Agr and sort $FIT.P$ as IT in Filter-Iterator and Iterator respectively.

Definition 11 Let A be any model of the abstract iterator specification. For any $it \in A_{IT}$ we say it is finite if, and only if, there exists an $i \in \mathbb{N}$ such that

$$it.next^{i}.isDone = True$$

The iterator A is said to be a finite iterator if, and only if, every $it \in A_{IT}$ is finite. □

In the following propositions, we show that $it - filter$ specification satisfies the axioms of the iterator specification. Each proposition shows an axiom is satisfied. The notation $A \models e$ (A models e) means that the model A satisfies the formula e.

Proposition 12 *Axiom 1.1 holds on the filter iterator specification with an arbitrary parameter P:*

$$\mathrm{it} - \mathrm{filter}(\mathsf{P}) \models \mathrm{it.create} = \mathrm{it.create.first}.$$

Proof :

1 - $\mathrm{it.create} = \mathrm{capsule}(\mathrm{it.first.gran})$
by transitivity(Axiom 4.1, Axiom 4.2)

2 - $\mathrm{it.first} = \mathrm{it.first}$
by reflection

3 - $\mathrm{it.create.first} = \mathrm{capsule}(\mathrm{it.first.gran}).\mathrm{first}$
by substitution(2, 1)

4 - $\mathrm{it.first.gran} = \mathrm{it.first.gran}$
by reflection

5 - $\mathrm{capsule}(\mathrm{it.first.gran}).\mathrm{first} = \mathrm{capsule}(\mathrm{it.first.gran.first.gran})$
by substitution(Axiom 4.2, 4)

6 - $\mathrm{it.gran.first} = \mathrm{it.first}$
by Lemma 8

7 - $\mathrm{it.first} = \mathrm{it.first}$
by reflection

8 - $\mathrm{it.first.gran.first} = \mathrm{it.first.first}$
by substitution(6, 7)

9 - $\mathrm{it.gran} = \mathrm{it.gran}$
by reflection

10 - $\mathrm{it.first.gran.first.gran} = \mathrm{it.first.first.gran}$
by substitution(9, 8)

11 - $\mathrm{capsule}(\mathrm{it}) = \mathrm{capsule}(\mathrm{it})$
by reflection

12 - $\mathrm{capsule}(\mathrm{it.first.gran.first.gran}) = \mathrm{capsule}(\mathrm{it.first.first.gran})$
by substitution(11, 10)

13 - $\mathrm{capsule}(\mathrm{it.first.gran}).\mathrm{first} = \mathrm{capsule}(\mathrm{it.first.first.gran})$
by transitivity(5, 12)

14 - $\mathrm{it.create.first} = \mathrm{capsule}(\mathrm{it.first.first.gran})$
by transitivity(3, 13)

15 - $\mathrm{it.first.first} = \mathrm{it.first}$
by Lemma 3

16 - $\mathrm{it.first.first.gran} = \mathrm{it.first.gran}$
by substitution(9, 15)

17 - $\mathrm{capsule}(\mathrm{it.first.first.gran}) = \mathrm{capsule}(\mathrm{it.first.gran})$
by substitution(11, 16)

18 - $\mathrm{it.create.first} = \mathrm{capsule}(\mathrm{it.first.gran})$
by transitivity(14, 17)

19 - $capsule(it.first.gran) = it.create$

by symmetry(1)

infer
$it.create.first = it.create$

by transitivity(18, 19)
□

We now consider the validity of the second iterator axiom. Again we require the finiteness condition similar to the previous proposition.

Fact 13 We note that in order to show that

$$it - filter(P) \models t[fit] = t'[fit]$$

where $t[fit] = t'[fit]$ is an equation involving $fit \in X_{FIT}$, it suffices to show by Lemma 9 that

$$it - filter(P) \models t[fit/capsule(it)] = t'[fit/capsule(it)]$$

for $it \in X_{IT.P}$. □

We use this fact to prove that the remaining iterator axioms are valid in $it - filter$ specification, given an arbitrary parameter model P of $iteratorPropTheory$.

Proposition 14 *Axiom 1.2 holds on Filter Iterator Specification with an arbitrary parameter P assumed to be finite.*

Proof : By Fact 13, it suffices to consider only FIT terms of the form $capsule(it)$, i.e.

$$it - filter(P) \models capsule(it).next.first = capsule(it).first$$

from

1 - $x.first = x.first$

by reflection

2 - $capsule(it).next.first = capsule(it.gran.next.gran).first$

by substitution(1, Axiom 4.5)

3 - $it.gran.next.gran = it.gran.next.gran$

by reflection

4 - $capsule(it.gran.next.gran).first = capsule(it.gran.next.gran.first.gran)$
by substitution(Axiom 4.2, 3)

5 - $capsule(it).next.first = capsule(it.gran.next.gran.first.gran)$
by transitivity(2, 4)

6 - $it.gran.first = it.first$
by Lemma 8

7 - $it.gran.next = it.gran.next$
by reflection

8 - $it.gran.next.gran.first = it.gran.next.first$
by substitution(6, 7)

9 - $it.gran = it.gran$
by reflection

10 - $it.gran.next.first = it.gran.first$
by substitution(axiom 1.2, 9)

11 - $it.gran.next.gran.first = it.gran.first$
by transitivity(8, 10)

12 - $it.gran.next.gran.first.gran = it.gran.first.gran$
by substitution(9, 11)

13 - $capsule(x) = capsule(x)$
by reflection

14 - $capsule(it.gran.next.gran.first.gran) = capsule(it.gran.first.gran)$
by substitution(13, 12)

15 - $capsule(it).next.first = capsule(it.gran.first.gran)$
by transitivity(5, 14)

16 - $it.gran.first.gran = it.first.gran$
by substitution(9, 6)

17 - $capsule(it.gran.first.gran) = capsule(it.first.gran)$
by substitution(13, 16)

18 - $capsule(it).next.first = capsule(it.first.gran)$
by transitivity(15, 17)

19 - $capsule(it.first.gran) = capsule(it).first$
by symmetry(Axiom 4.2)

infer
$capsule(it).next.first = capsule(it).first$
by transitivity(18, 19)
□

We now come to showing the validity of the third iterator Axiom. Note that the proposition does not require that the iterator be finite since the

axiom is guarded.

Proposition 15 *Axiom 1.3 holds on Filter Iterator Specification with an arbitrary parameter P.*
Proof : By fact 13, it suffices to consider only FIT terms of the form capsule(it), i.e.

$$\mathrm{it} - \mathrm{filter}(\mathsf{P}) \models$$

$$\mathrm{capsule}(\mathrm{it}).\mathrm{isDone} = \mathrm{true} \Rightarrow \mathrm{capsule}(\mathrm{it}).\mathrm{next}.\mathrm{isDone} = \mathrm{true}$$

from $\mathrm{capsule}(\mathrm{it}).\mathrm{isDone} = \mathrm{true}$

1 - $\mathrm{it.gran.isDone} = \mathrm{capsule}(\mathrm{it}).\mathrm{isDone}$
by symmetry(Axiom 4.7)

2 - $\mathrm{it.gran.isDone} = \mathrm{true}$
by transitivity(1, assumption)

3 - $\mathrm{x.isDone} = \mathrm{x.isDone}$
by reflection

4 - $\mathrm{it.gran} = \mathrm{it.gran}$
by reflection

5 - $\mathrm{it.gran.isDone} = \mathrm{true} \Rightarrow \mathrm{it.gran.next.isDone} = \mathrm{true}$
by substitution(Axiom 1.3, 4)

6 - $\mathrm{it.gran.next.isDone} = \mathrm{true}$
by CondEq(5, 2)

7 - $\mathrm{it.gran.next} = \mathrm{it.gran.next}$
by reflection

8 - $\mathrm{capsule}(\mathrm{it.gran.next}) = \mathrm{capsule}(\mathrm{it.gran.next.gran})$
by substitution(Axiom 4.8, 7)

9 - $\mathrm{capsule}(\mathrm{it.gran.next.gran}) = \mathrm{capsule}(\mathrm{it.gran.next})$
by symmetry(8)

10 - $\mathrm{capsule}(\mathrm{it}).\mathrm{next} = \mathrm{capsule}(\mathrm{it.gran.next})$
by transitivity(Axiom 4.5, 9)

11 - $\mathrm{capsule}(\mathrm{it.gran.next}).\mathrm{isDone} = \mathrm{it.gran.next.gran.isDone}$
by substitution(Axiom 4.7, 7)

12 - $\mathrm{capsule}(\mathrm{it}).\mathrm{next.isDone} = \mathrm{capsule}(\mathrm{it.gran.next}).\mathrm{isDone}$
by substitution(3, 10)

13 - $\mathrm{capsule}(\mathrm{it}).\mathrm{next.isDone} = \mathrm{it.gran.next.gran.isDone}$
by transitivity(12, 11)

14 - $(\mathrm{it.gran.next.prop}$ or $\mathrm{it.gran.next.isDone}) = \mathrm{true}$
$\Rightarrow \mathrm{it.gran.next.gran} = \mathrm{it.gran.next}$
by substitution(Axiom 4.3, 7)

15 - $\mathrm{it.gran.next.gran} = \mathrm{it.gran.next}$
by CondEq(14, 6)

16 - $it.gran.next.gran.isDone = it.gran.next.isDone$
by substitution(3, 15)
17 - $capsule(it).next.isDone = it.gran.next.isDone$
by transitivity(13, 16)

infer
$capsule(it).next.isDone = true$
by transitivity(17, 6)
□

The following proposition is similar to the previous one in being guarded.

Proposition 16 *Axiom 1.4 holds on Filter Iterator Specification with an arbitrary parameter P.*
Proof : By fact 13, it suffices to consider only FIT terms of the form capsule(it), i.e.

$it{-}filter(P) \models capsule(it).isDone = true \Rightarrow capsule(it).current = error$

from $capsule(it).isDone = true$

1 - $it.gran.isDone = capsule(it).isDone$
by symmetry(Axiom 4.7)
2 - $it.gran.isDone = true$
by transitivity(1, assumption)
3 - $it.gran = it.gran$
by reflection
4 - $it.gran.isDone = true \Rightarrow it.gran.current = error$
by substitution(Axiom 1.4, 3)
5 - $it.gran.current = error$
by CondEq(4, 2)

infer
$capsule(it).current = error$
by transitivity(Axiom 4.6, 5)
□

6 Conclusions

In this paper, we have shown that a simple pattern can be formally specified using algebraic techniques, and in particular, a combination of loose

and initial semantics together with constructor constraints. Such algebraic specification serve to document the main attributes of a design pattern: loose semantics used to capture intentional "gaps" in the pattern, while initial semantics and constructor constraints are used to axiomatise the fixed structure of the pattern. We illustrated our approach by constructing an abstract specification of the iterator design pattern. We considered how this abstract specification could be used to verify an implementation of the iterator pattern and in particular, how the Maude tool can automate such verifications. Finally we considered in detail specifying a refinement, the so called filter refinement, for the iterator pattern. We formally proved this refinement correct with respect to the patterns abstract specification.

The formal specification aimed mainly to provide a clarity of what the pattern is, and how it could be implemented, though at a high level of abstraction. The formal treatment of the specification allowed us to see some weakness in the refinement, which appears in the case of a non-terminating iterator that could result in an infinite loop. Such weaknesses might not have been observed if not for the formal treatment of the refinement. Such insights are one of the main motivations for formalizing design patterns. The algebraic method used was practical for discussing the attributes of the pattern. In our opinion, our approach so far was very beneficial in not only describing, but also analyzing a design pattern to form a solid abstract specification, and hence an implementation less prone to errors. It was also successful in maintaining the simplicity of reasoning about the pattern specification and a relatively easy to use tool support.

The specified refinement made in this paper was only one possible reification of the iterator pattern. Specification for other refinements could be made to try and capture a range of possible refinements, and hence provide a detailed treatment of the pattern, while still maintaining the general nature of the pattern. Each refinement could be considered a sub-pattern of the iterator pattern.

Proofs for instances of an iterator tend to be straight forward and automatable proofs (see Section 3.4), while validating refinements (see Section 5) are more complex and difficult to automate. This fits well in practice, since instance proofs tend to be done numerous times, while refinement proofs are "one off" proofs.

Work on formalizing other patterns with a different nature is ongoing and will be reported else where. In further work, we intend to investigate and clarify the formal relationships between different patterns.

7 Acknowledgement

We would like to thank Cliff Jones for his help and advice while writing this paper. We would also like to thank Kuwait University for their sponsorship during this work.

References

[1] Lepus: Language for patterns uniform specification. http://www.cs.concordia.ca/ faculty/eden/lepus/.

[2] Maude, http://maude.csl.sri.com/.

[3] Pattern forms, http://c2.com/cgi/wiki?PatternForms.

[4] Patterns home page. http://hillside.net/patterns/patterns.html.

[5] J. C. Bicarregui and J. S. Fitzjerald. *Proof in VDM: A Practioner's Guide*. Springer-Verlag, 1994.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. WILEY, 1996.

[7] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada. A maude tutorial, March 2000.

[8] Amnon H. Eden. Giving the quality a name. *Journal of Object Oriented Programming*, June 1998.

[9] Amnon H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, Department of Computer Science, Tel Aviv University, 2000.

[10] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. Springer-Verlag, 1985.

[11] J. L. Fiadiero and T. Maibaum. Sometimes "tomorrow" is "sometime". In H. Ohlbach D. Gabbay, editor, *Temporal Logic*, volume 827 of *Lecture Notes in Atrificial Intelligence*, pages 48–66. Springer-Verlag, 1996.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[13] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[14] A. Gustavsson and A. Ersson. Formalizing the intent of design patterns. `http://www.csd.uu.se/~eden/patterns_and_frameworks/papers/`, July 1999.

[15] K. Lano, S. Goldsack, and J. Bicarregui. Formalizing design patterns. *In 1st BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computer Science*, 1996.

[16] J. Loecks, H-D. Erich, and M. Wolf. *Specification of Abstract Data Types.* WILEY, 1996.

[17] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, , and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 1, pages 189–412. Oxford University Press, 1993.

[18] Tommi Mikkonen. Formalizing design patterns. *Preceedings of the 20th International Conference on Software Engineering*, 1998.

[19] Coplien Schmidt, editor. *Pattern Languages of Program Design.* WILEY, 1996.