Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 147-166

A TOOL FOR FORMALLY SPECIFYING THE C++ STANDARD TEMPLATE LIBRARY[1]

G. Dévai and N. Pataki

(Budapest, Hungary)

Abstract. In this paper we present ongoing research on formal specification of the *C++ Standard Template Library*. Our goal is to embed the *STL* datatypes and their operations into a system that is used to produce formally verified code.

From the point of view of the *C++* programmers such a tool is a great help when one writes safety critical applications. On the other hand, integration of well-known libraries makes formal methods more useful and more attractive.

We use the *vector* datatype in the examples and present two models to describe the properties of its instructions. The first one is on higher abstraction level and provides a simpler interface, but it is too restrictive. The second one is based on pointer semantics and avoids the limitations of the first model.

1. Introduction

1.1. The *C++ Standard Template Library*

The *C++ Standard Template Library (STL)* [24, 15, 4] is the most popular library based on the generic programming paradigm. *STL* is widely-used, because the library is part of the *C++* Standard. It consists of many useful generic data

[1]This work is supported by "Stiftung Aktion Österreich-Ungarn (OMAA-ÖAU 66öu2)" and "ELTE IKKK (GVOP-3.2.2-2004-07-0005/3.0)".

structures and generic algorithms that work together with containers. *STL* is based on generalization and this causes a simplified interface.

C++ STL consists of three main parts: containers, iterators and algorithms. Containers (e.g. vector, list, map, set, etc.) hold elements. Containers are the generalization of arrays. Iterators guarantee access for the elements in containers and they are nested types of them. Iterators are generalization of pointers, their standard interface originates from pointer arithmetic. Algorithms are fairly irrespective of the used container, because they work with iterators. For instance, we can use the *for_each()* algorithm with all containers. As a result of this layout the complexity of the library is greatly reduced and we can extend the library with new containers and algorithms simultaneously. This is a very important feature because object-oriented libraries do not support this kind of extension. The *C++* standard gives complexity guarantees for each operation.

1.2. Formal specification of *STL*

STL helps programmers to produce safer and more readable code because it provides a more abstract level of programming compared to lower level datatypes and algorithms. However, the library does not exclude all kinds of programming errors. For example it can not prevent indexing out of a vector, it only makes it a little bit safer: when using lower level datatypes like an array, we have the [] operator only, which usually leads to a segmentation fault (or junk data) if we use an out of bound index. In contrast, *STL* has introduced the *at*() function which throws an exception in case of an invalid index. (On the other hand, *STL* also provides [] to have the possibility avoiding the overhead of the runtime check.)

It is also a problem that the library is defined by informal specification. It may lead to misunderstood in some special cases and it does not help when one needs to formally prove the correctness of a safety-critical application.

Our goal is to integrate the *C++ STL* into a formal system to produce *C++* programs that use the *STL* and are proved correct. This way we can avoid the errors that the usage of the library alone cannot prevent. We can also ensure that the produced code conforms to the formal specification given in the beginning of the development process.

Integration of useful libraries such as the *STL* is also quite important from the point of view of formal methods. In industrial software development only those systems and languages are able to gather ground, which support the work of the programmer by a wide scale of useful libraries arriving with the system. By our integration the user of the formal system is not forced any more to work out common datatypes and algorithms from scratch, but has the possibility to use those of the *STL*.

In order to achieve our goals, we had to define new types, functions and

predicates in the selected formal system. These types and functions were then used to state temporal axioms of certain operations of the *STL*, like adding an element to a vector or positioning an iterator. During this work we made use of the higher level of abstraction provided by the *STL*, because the more abstract the entity to specify, the easier its formal specification.

But these formal specifications of the operations are not enough. It often happens that the postcondition of one operation establishes a statement describing a container, but the precondition of the next statement requires it in a different form. To make the life of user easier, we have written tactics that perform these conversions automatically.

1.3. The *LaCert* system

LaCert [8, 1] is a formal system to produce program code that is proved correct. While testing may not find all the errors in a program, formal methods only accept programs that behaves exactly as the (formal) specification prescribes.

There are lots of formal methods with this same goal. In verification we have tools to prove the soundness of a program when it has been written. The usual setup consists of a semi-automatic theorem prover [6, 12, 7, 19, 25] and a tool to transform the program into its representation in the prover. There are also some projects that try to make the verification fully automatic [27, 3, 5]. However, these approaches have some drawbacks. Most of the time, programs are wrong and the errors are discovered too late in the development process, namely when we find them during the construction of the proof. Another problem is, that programming languages nowadays are quite complex. Supporting all the language elements would make the tools enormously complex, that is why only subsets of the languages are usually supported by these projects.

The other approach (that we claim to have more potential) is refinement [2, 14, 16]. In this case the development starts by writing the formal specification and then refining it until we reach the implementation. This way, design and programming errors are discovered earlier and the resulting implementation is correct by construction.

In some refinement based systems (like in the *B-method* [2], for example) one writes program code (commands, loops, etc.) in the last steps of the refinement. In our system the refinements always result in (more detailed) specifications, and when these are detailed enough, the program code is generated automatically from them. In this way the system never has to deal with unsupported language constructs.

LaCert helps the construction of refinements by metaprogramming techniques. When we reason about programs, there are often used proof fragments, like the proof of a while loop, conditional branch or procedure call, etc. In *LaCert* we

can include these common fragments in *meta-proofs* (called *templates*). These templates can be called with arguments to produce the proof for a concrete program fragment. Metaprogramming facilities of this system also helped us to deal with the specification of *STL* components, as these are based on template metaprogramming capabilities of *C++*.

1.4. Results presented in this paper

In this paper we examine possibilities to specify the instructions of STL containers and their iterators in a formal way. We use the *LaCert* system in our research, but the questions to be solved and our answers given to them are general and can be applied independently of the selected environment.

The main problem that we had to solve is the correct representation of the knowledge that an iterator points to an element in a certain container. In some cases, this knowledge must hold also after the modification of the container (for example if we alter an other element), while other instructions may completely invalidate iterators pointing into that container.

Our solution uses two models: one to describe the contents of a container only and an other one to represent iterators. The 'dangerous' instructions (that may invalidate iterators) should be specified in the first model while other instructions in the second one. We also give the necessary axioms to connect the two models.

By this integration of pieces of *C++ STL* into a formal system we are able to generate *STL*-based code with formally proved properties. We believe that in the long run it will be possible to develop small pieces of large *C++* projects (safety-critical core methods for example) using our method. The results presented in this paper are the first steps towards this goal. At the end of the paper we evaluate our experience with the implementation of the models and point out the next steps to be done.

1.5. Layout of the paper

In Section 2 we present a functional style framework used to describe states of the containers. In Section 3 we present our first approach to specify vector (3.1, 3.2) and iterator (3.3) operations. After pointing out the limitations of this solution, we present an improvement in Section 4. We examine related work in Section 6.

2. Framework

In order to represent the contents of different containers (like vectors, lists, etc.) in the specifications and proofs, we introduce a polymorphic sequence type. We use a functional programming style, as it often happens in the meta- and generative programming paradigms.

We declare the *nil*() function to produce an empty sequence, the .+ and the +. operators to add a new element to the beginning or to the end of the sequence. We show the *LaCert* style declarations of these elements:

```
type( Seq, 1 );
function( ''nil'', Seq(#T) );
function( ''.+'', Seq(#T), #T, Seq(#T) );
function( ''+.'', Seq(#T), Seq(#T), #T );
```

The type declaration introduces the type *Seq* with one type parameter (the type of the elements in the sequence). This type parameter is $\#T$ in the function declarations. In a function declaration, the name of the function is followed by the return type and after that the types of the parameters follow.

The expression representing the sequence of characters a, b and c is then the following:

```
'a' .+ ( 'b' .+ ( 'c' .+ nil() ) )
```

As the operator .+ is defined to be *right associative*, we can omit the parenthesis:

```
'a' .+ 'b' .+ 'c' .+ nil()
```

When we specify operations of *STL* containers, the following predicate will help us a lot:

```
function( ''split'', Boolean, Seq(#T), Seq(#T), Seq(#T) );
```

That is, this predicate takes three sequences as arguments and returns a boolean value. Informally speaking, the expression $split(a, b, c)$ states that a is the concatenation of b and c. We express this property by the following two axioms in *LaCert*.

```
axiom split1( Seq(#T) #seq, Seq(#T) #seqVal )
{
  #seq = #seqVal => split( #seq, nil(), #seqVal );
```

```
}

axiom split2( Seq(#T) #seq, Seq(#T) #front, Seq(#T) #tail )
{
  split( #seq, #front, #elem .+ #tail )
  => split( #seq, #front +. #elem, #tail );
}
```

The axiom *split1* has two formal parameters: $\#seq$ and $\#seqVal$, each of type $Seq(\#T)$. We can call these axioms for example to prove the following:

```
s = 'a' .+ 'b' .+ 'c' .+ nil()
=> split( s, nil() +. 'a', 'b' .+ 'c' .+ nil() )
{
  split1( s, 'a' .+ 'b' .+ 'c' .+ nil() );
  split2( s, nil(), 'a' .+ 'b' .+ 'c' .+ nil() );
}
```

In the first two lines of this code you can find the statement to prove, and between the curly braces there is the proof. It is instructive to replace the arguments in the calls of *split1* and *split2* in the definition of the axioms above, and see how does the proof work out.

This kind of reasoning will be quite necessary when we prove properties of programs using *STL* datatypes, so we have created tactics to complete proofs of this kind automatically. That is, we can simply write

```
s = 'a' .+ 'b' .+ 'c' .+ nil()
=> split( s, nil() +. 'a', 'b' .+ 'c' .+ nil() );
```

to make the system produce the proof.

3. Sequence-based model

3.1. Clearing the vector

We use sequences to describe the contents of different containers, like vectors, lists, queues, etc. For this reason we introduce the *values* function, that gives the sequence of the elements of the container. If v is of type $Vector(Character)$, the statement

```
values( v ) = nil()
```

states that v is empty. In *LaCert* we can include explicit information on the current location of the program execution using the predefined variable ip (*instruction pointer*). For example, if we want to state that the program execution is at the label L and the vector v is empty, we can write:

```
ip = L & values( v ) = nil()
```

By connecting two such statements by the $\mathbb{G}$ operator, we gain a *temporal progress property*:

```
ip = K >> ip = L & values( v ) = nil()
```

This progress property states that whenever the program execution reaches label K the program has to reach label L and then the vector must be empty. To state that during this progress an other vector w preserves its only element (the number 2), we can use a *safety property* before the progress property:

```
[ values( w ) = 2 .+ nil() ];
ip = K >> ip = L & values( v ) = nil()
```

If we want to express that this part of the program changes only the variables ip and v, but nothing else, we can state that every formula not containing ip and v is a safety property:

```
independent( $prop, ip ) & independent( $prop, v )
: [ $prop ];
ip = K >> ip = L & values( v ) = nil()
```

When the compiler of *LaCert* needs to decide whether a formula like $values(w) = 2. + nil()$ is a safety property or not, it replaces the *expression variable* $\$prop$ by a the formula and evaluates the condition. As the formula $values(w) = 2. + nil()$ is independent of ip and v, it satisfies the condition.

Now we are about to specify the *clear()* function for vectors. We replace the labels K, L and the vector v by parameters:

```
atom clear( Vector(#T) #vect, Label #before, Label #after )
{
  independent( $prop, ip ) & independent( $prop, #vect )
  : [ $prop ];

  ip = #before >> ip = #after & values( #vect ) = nil();
}
```

Similarly to the *axiom* keyword that we used to write axioms in classical logic, we use the *atom* keyword here to introduce temporal axioms (safety and progress properties) of a target language instruction. If we call this atom with the arguments v, K and L respectively, we get the properties in the example above.

3.2. Adding and removing elements

In the following we show only the progress properties of the atoms to save space. For example, we specify vector's *push_back()* operation. We take advantage of our *split* predicate here to be able to manipulate the end of the contained sequence easier. Let us assume that $\#val$ is the value to add to the end of the vector and $\#vVal$ is the sequence contained by the vector. The progress property of the atom is as follows.

```
ip = #before & split( values(#vect), #vVal, nil() )
>> ip = #after & split( values(#vect), #vVal, #val .+ nil() );
```

If we interchange the pre- and the postcondition, we get the specification of the *pop_back()* operation:

```
ip = #before & split( values(#vect), #vVal, #val .+ nil() )
>> ip = #after & split( values(#vect), #vVal, nil() );
```

Note, how this specification prevents performing *pop_back()* on an empty vector: the precondition states that there is an element ($\#val$) at the end of the vector. If $values(\#vect) = nil()$ holds, the precondition is unprovable.

Let us suppose that we want to refine the following specification:

```
ip = L & values( v ) = 'a' .+ 'b' .+ nil()
>> ip = M & values( v ) = 'a' .+ 'b' .+ 'c' .+ nil()
```

It is clear, that the *push_back*() instruction can solve this task, but calling its specification is not enough. We also have to prove the appropriate *split* formula in the precondition of the operation, and analogously, from the *split* formula in the postcondition of the atom we have to prove the postcondition of the progress property that we want to refine. As we have already mentioned, we have implemented tactics that discharge such proof obligations automatically.

3.3. Iterators in the sequence-based model

Iterators are abstractions of indexes and pointers. We use them to access elements of different containers uniformly. To specify pointer operations, we need

predicates that tell which element in which container does the iterator point to.

The idea is to extend *split* by a new parameter: the iterator. We also change the first parameter a little bit: instead of the values of the vector, we write there the vector itself. This way, we get a predicate like

$$split(vect,\ iter,\ front,\ tail),$$

which states, that

- if we concatenate $front$ and $tail$, we get the values of the vector $vect$,
- and if $tail$ is not empty, the iterator $iter$ points to its first element,
- and if $tail$ is $nil()$, then $iter$ equals to the $vect.end()$ value.

To make this clearer, let us observe the progress property of the instruction that increments the iterator:

```
ip = #before
& split( #vect, #iter, #front, #elem .+ #tail )
>> ip = #after
& split( #vect, #iter, #front +. #elem, #tail );
```

In the postcondition, $\#elem$ is concatenated at the end of $\#front$, so, according to the meaning of this extended split, $\#iter$ now points to the first element of $\#tail$, which is the one following $\#elem$. The progress property for decrementing an iterator is similarly straightforward.

This specification differs from the one in [21], because here we make it explicit in which container the iterator points to. It is also different from the lower level pointer arithmetic specification of [18].

Now we change one element of the vector via an iterator. If the parameter representing the value to write is $\#val$, the progress property of this operation is the following.

```
ip = #before & split( #vect, #iter, #front, #elem .+ #tail )
>> ip = #after
& split( #vect, #iter, #front, #val .+ #tail );
```

Note, how the precondition prevents the programmer indexing out of the vector: if the precondition holds, then we can be sure that $\#iter$ points to the $\#elem$ element of the vector.

3.4. Limitation of this model

As we will demonstrate in this section, the presented model is too restrictive: if there are two iterators pointing to the same container and we use one to change an element, we loose all information about the other iterator.

Let us observe the safety properties of the operation that changes an element. As this instruction modifies the variables *ip* and *#vect*, its safety property must be this one:

```
independent( $prop, ip ) & independent( $prop, #vect )
: [ $prop ];
```

Now let us assume that before the execution of this instruction we have the two iterators *iter*1 and *iter*2, and we know the following about them:

```
split( v, iter1, nil(), 'a' .+ 'b' .+ nil() )
split( v, iter2, nil() +. 'a', 'b' .+ nil() )
```

That is, *iter*1 points to the first element (a), while *iter*2 points to the second one (b). If we write the character c in place of a, from the temporal axiom of this operation we get

```
split( v, iter1, nil(), 'c' .+ 'b' .+ nil() ).
```

But what happens to our previous knowledge about *iter*2? The compiler of *LaCert* uses the safety property of the instruction to decide whether that formula is still valid after the execution of the operation or not. The $\$prop$ expression variable is replaced by the formula $split(v, iter2, nil() + .'a', 'b'. + nil())$ in the condition we gave above, and that is evaluated:

```
independent( split( v, iter2, nil() +. 'a', 'b' .+ nil() ), ip )
& independent( split( v, iter2, nil() +. 'a', 'b' .+ nil() ), v )
```

The variable v occurs in the formula, so the second part of the condition fails. This means that the formula is not a safety property of this instruction and we lose our previous knowledge about the iterator *iter*2.

This is all right, because v is really changed, and instead of

```
split( v, iter2, nil() +. 'a', 'b' .+ nil() )
```

we would like to have

```
split( v, iter2, nil() +. 'c', 'b' .+ nil() )
```

after the instruction. But, unfortunately, the current specification of this instruction does not provide this new knowledge.

What can we use this model for, considering the limitation pointed out above?

- If the elements of the container are not modified, multiple iterators may be used to read the contents.
- If one single iterator is used, it may both read and write the elements of the container.

These cases cover a wide range of algorithms, but not all: in *bubble sort*, for example, we use two iterators and also alter the container interchanging its elements.

4. Pointer-based model

The limitation that we pointed out at the end of the previous section is a special case of the problems arising when we try to reason about programs operating with pointers and dynamic memory management. This is no surprise, since the implementation of the *STL* containers and iterators use pointers and heap memory, and on the other hand it is possible to simulate the heap operations by containers like a vector or a list.

Local reasoning by *separation logic* [22, 20] is an elegant solution to prove properties of programs using heap memory. In a previous work [9], we integrated separation logic into *LaCert*. To manage to do that, we had to transform the special logical connectives of separation logic back into classical logic. This transformation was based on the semantics of separation logic [28] and resulted in a model where each variable allocated on the heap memory had a label (as an abstraction of its address), and in the proofs we kept count of which pointer points to which label.

A similar solution is applicable now. We introduce a new type called *Cell*, and use the $\sim$ operator to join a label and a value to form a *Cell*. Similarly to the *values* function that we used before, we can declare the *cells* function that gives the sequence of cells in a container. For example we can write

```
cells( v ) = (X ~ 'a') .+ (Y ~ 'b') .+ ...
```

to express that the first two elements of container v are a and b with addresses X and Y, respectively. In case of vectors, instead of labels we could use indexes, but this does not make sense for lists for example, where it is easy to insert elements

in the middle of the container which would modify all indexes after the insertion point.

If the iterator *iter*1 points for instance to the first element (a) in the example above, we can think of it as an object containing a pointer which equals to X (the 'address' of the element). So we declare the function *label* to get that label from the iterator:

```
label( iter1 ) = X
label( iter2 ) = Y
```

These formulas together with the above description of the cells of the container v state that the iterators *iter*1 and *iter*2 point to the first and second elements of v respectively.

Let us examine whether this solution solves the limitation of the sequence-based model pointed out in Section 3.4. If we overwrite the value a by c using the iterator *iter*1, we get

```
cells(v) = (X ~ 'c') .+ (Y ~ 'b') .+ ...
```

This operation modifies the instruction pointer (ip) and the vector (v) only, so the formulas describing the iterators remain valid:

```
label( iter1 ) = X
label( iter2 ) = Y
```

That is, we can prove that after the instruction *iter*1 points to c and *iter*2 points to b.

4.1. Iterator operations and the *vect.end*() value

To specify the iterator operations correctly, we have to think about the *vect.end*() iterator value of a vector *vect*. If we increment an iterator pointing to the last element of *vect*, then it becomes equal to *vect.end*().

As we have seen in Section 3.3, the sequence-based model handles this special value quite naturally: the assertion

```
split( values(vect), it, seq, nil() )
```

means that the values stored in *vect* are in the sequence *seq* and the iterator *it* equals to *vect.end*().

In the pointer-based model we have assertions of the form $label(it) = addr$, where *addr* is the address of the element the iterator is pointing to. That is,

we need a 'virtual' element at the end of the vector, and an iterator equal to *vect.end*() should hold the address of that element. For example, to state that *vect* contains three characters, *a*, *b* and *c* and $it = vect.end()$ we can write:

```
cells( vect )
  = (L1~'a') .+ (L2~'b') .+ (L3~'c') .+ (End~'?') .+ nil()
& label( it ) = End
```

The value of the 'virtual' element is completely indifferent as the specifications of operations will avoid accessing it.

Using this solution, the specifications of *incrementing* and *decrementing* an iterator are the following:

```
ip = #before & label( #iter ) = #Cur
& split( cells(#vect), #front,
         (#Cur~#cVal) .+ (#Next~#nVal) .+ #tail )
>> ip = #after & label( #iter ) = #Next;

ip = #before & label( #iter ) = #Cur
& split( cells(#vect), #front,
         (#Prev~#pVal) .+ (#Cur~#cVal) .+ #tail )
>> ip = #after & label( #iter ) = #Prev;
```

The precondition of *writing an element* via an iterator has to ensure that it is not the 'virtual' element we want to modify. That is why $!(\#tail = nil())$ is present in the precondition of the following specification.

```
ip = #before & label( #iter ) = #Addr & !(#tail = nil())
& split( cells(#vect), #front, (#Addr~#val) .+ #tail )
>> ip = #after
& split( cells(#vect), #front, (#Addr~#newVal) .+ #tail );
```

4.2. Vector operations and iterator invalidation

Operations that change the size of a vector may result in *relocation* of the container. For example, if we add a new element at the end of the vector using the *push_back* instruction and the *capacity* (the size of memory reserved for the vector) is exceeded, all the elements of the vector are copied to a new location in the memory. This means, that the iterators pointing to this container are not valid any more, they are *invalidated*.

After such an operation we do not know anything about the addresses of the elements, we know their values only. For this reason we do not specify these instructions in the pointer-based model, but leave the specifications presented in Sections 3.1 and 3.2.

4.3. Connecting the two models

The previous section states that we have to use the specifications of the sequence-based model to add elements to a vector. Let us assume that we use the pointer-based model and know that

```
cells(v) = (X~'a') .+ (Y~'b') .+ (End~'?') .+ nil()
```

holds for the vector v. If we want to add new elements to the vector now, we need information on $values(v)$ instead of $cells(v)$.

To connect the two models, we declare the function *vals*, which extracts the sequence of values from a sequence of cells, forgets the addresses as well as the last, 'virtual' cell. This is given by the following axioms:

```
vals( #end .+ nil() ) = nil();

vals( (#X~#v) .+ #elem .+ #tail ) = #v .+ vals( #elem .+ #tail );
```

So we can prove the following:

```
vals( (X~'a') .+ (Y~'b') .+ (End~'?') .+ nil() )
= 'a' .+ 'b' .+ nil()
```

If we add the axiom

```
vals( cells(#v) ) = values(#v);
```

where $\#v$ is an arbitrary vector, we can prove:

```
cells(v) = (X~'a') .+ (Y~'b') .+ (End~'?') .+ nil()
=> values(v) = 'a' .+ 'b' .+ nil()
```

From now on it is possible to use the specification of *push_back* described in Section 3.2.

To switch between the two models in the other way around, we can use the following axioms, where $\#cs$ is a sequence of cells[2]:

```
vals( #cs ) = nil()
=> exists( Label @End, exists( Character @c,
     #cs = (@End~@c) .+ nil()
   ) );
```

[2]Existential quantification $\exists x.P$ in *LaCert* has the following syntax:
`exists(T @x, P)`, where T is the type of the existentially quantified variable $@x$.

```
vals( #cs ) = #val .+ #tail
=> exists( Label @Addr, exists( Seq(Cell) @cs,
     #cs = (@Addr~#val) .+ @cs & vals(@cs) = #tail
   ) );
```

By recursive applications of these rules we can prove the following:

```
values(v) = 'a' .+ 'b' .+ nil()
=> exists( Label @X, exists( Label @Y, exists( Label @End,
   exists( Character @c,
     cells(v) = (@X~'a') .+ (@Y~'b') .+ (@End~@c) .+ nil()
   ) ) ) )
```

LaCert supports the introduction of parameters for existentially quantified variables, which is a well known proof technique in first order logic. By introducing *parX*, *parY*, *parEnd* and *parC* parameters for @*X*, @*Y*, @*End* and @*c*, we get:

```
cells(v) = (parX~'a') .+ (parY~'b') .+ (parEnd~parC) .+ nil()
```

From this point we can use the specifications of the pointer-based model.

5. Evaluation of the implementation

Up to now we have implemented the specifications of the mentioned operations for the *vector* container type. Even when writing quite simple programs in the system it turned out that establishing the precondition of an operation from the postcondition of the previous one often requires several logical steps. Consider the example at the end of Section 3.2.

We have also observed that, most of the time, these formula-transformations can be automated. Using the template facilities of *LaCert* we started to create tactics to ease the construction of proofs. At the end of Section 2 you can find an example. We experienced this as a great help and we plan to implement more tactics.

One can see that switching between the two models is tedious if done by hand, but can be done automatically. That is, we will be able to define templates that generates the necessary proof fragments.

Currently, in our system we can write specifications and proofs of small and simple programs that use *vector* and its *iterators*. Our system outputs correct *C++* programs using the mentioned features of the *STL*.

6. Related work

Various efforts have been made towards a safer usage of *STL*. In the following we give a short overview of these projects and point out the differences of our work.

David Musser and *Changqing Wang* designed a *Hoare logic* style calculus [18] for specific *C++* and *STL* language elements. Because they were interested mainly in the verification of *STL implementations*, their machinery for iterators is similar to pointer arithmetic and it makes proofs rather complicated. In contrast, we are interested in producing verified *programs that use STL*, therefore our specifications can take advantage of the higher level of abstraction introduced by iterators.

Musser also worked out *algebraic specifications* [17] which are more difficult to use for verification. Both the Hoare style and the algebraic specifications were integrated into the *Tecton Proof System* [13]. The *MELAS* [26] system was also developed to support the verification process. It works by symbolic execution of the code and it is integrated into the *GDB* debugger. In order to use that system one writes the program, complies it and tries to verify certain properties by *MELAS*. In our case this process is reversed: one writes the specification first, then refines an implementation from it and the code (which is correct by construction) is generated automatically.

Matthew H. Austern uses *pre- and postconditions* to specify the *STL* data-types and operations in his book [4]. However his work is not a formal description, therefore (just as the STL standard) it is not suitable for formal verification directly. On the other hand it is applicable for educational purposes and can be used as a basis for formalization efforts.

When one deals with metaprogramming, it is often difficult to decipher error messages originating from erroneous template instantiations. The *Boost Concept Check Library (BCCL)* [23] checks whether template parameters conform to certain restrictions (so called *concepts*) that describe the intended use of the template in question. This tool is not for verification of the resulting program, but it makes the usage of *STL* easier and safer.

The *STLlint* [11] project aims finding common programmer errors in *C++* code using *STL*. Such as similar tools (like *ESC/Java2* [10]) it does not *verify* the code, because it can not find all kinds of errors. In our system it is a main requirement that every program generated by the system must conform to the specification.

The antecedent of our current work is a *Hoare style* specification described in [21]. Now we integrate similar specifications into *LaCert* to obtain computer assistance for the proofs and get running *C++* programs as a result.

7. Summary

7.1. Applicability

Now we give scenarios on the applicability of our results. We think that they can already be used by researchers dealing with similar problems and in the education of formal methods or of the *C++ STL*. We expect that in the future it will be applicable also by C++ programmers.

Even though our research is in progress, for *researchers of formal methods* our results are already useful. We have presented relevant questions that are to be solved in order to formally specify containers and iterators. We have also given possible solutions to these questions. The main ideas of these solutions are independent of the formal system we use, and — to a certain extent — independent of C++, too. This means that our ideas are applicable using other formal systems and maybe using other languages with standard containers and iterators.

In *education* one can use our research as a real-life example when teaching formal methods. Our work demonstrates that even specification of basic instructions is not a simple task and gives ideas how to solve the difficulties. When teaching *C++ STL*, presentation of the instructions' formal specifications can lead to deeper understanding of the library. While natural-language descriptions may lead to misunderstoods in special cases, the formal ones are unambiguous. If we can cover a larger part of the *STL*, our work will be useful as a reference, too.

In the long run our system will be usable for *C++ programmers*. In large software projects there are often safety critical code fragments. If the overhead of a formal method is not too high, it is worth to produce these code fragments with proved properties. The main problem is that there are frequent, not-too-complicated proof fragments that computers fail to produce fully automatically, and that are tedious to produce by hand. Our approach is to generalize these fragments and encapsulate them into templates. As we have pointed out in Section 5, these templates and tactics help a lot to reduce the overhead of formal program development. We expect that by implementing more of them (for example to help switching between the two models) the effort needed to use our system in practice will be acceptable.

7.2. Future plans

The next container we plan to formally specify is *list*. The sequence-based specifications will be quite similar to those of *vector*, but the pointer-based model will differ. This is due to the different iterator invalidation rules for lists. For example, inserting elements into a list does not invalidate iterators. This means, that it will be possible to give a specification for insertion in the pointer-based model, too.

The containers we have mentioned so far are *sequences*. In the *STL* there are also *associative containers* like *set* and *map*. We will examine the possibilities to specify these containers, too.

Besides containers, the *STL* contains a wide range of algorithms, like counting or searching elements, sorting a container, etc. It will be a challenge to correctly specify these instructions, especially, because they can get other (usually user-defined) functions as parameters. Consider sorting, where the user can define an own comparison operation and sort the container according to that. This leads to the topic of *composing specifications*.

References

[1] **Home of LaCert:** `http://deva.web.elte.hu/LaCert`

[2] **Abrial J.-R.,** *The B-book: assigning programs to meanings*, Cambridge University Press, New York, NY, USA, 1996.

[3] **Antoy S. and Hamlet D.,** Automatically checking an implementation against its formal specification, *IEEE Trans. on Software Engineering,* **26** (1) (2000), 55-69.

[4] **Austern M.H.,** *Generic programming and the STL*, Addison-Wesley, 1999.

[5] **Barnes J.,** *High integrity software: The SPARK approach to safety and security*, Addison-Wesley, 2003.

[6] **Bertot Y. and Castéran P.,** *Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions*, Texts in Theoretical Computer Science, Springer Verlag, 2004.

[7] **de Mol M., van Eekelen M. and Plasmeijer R.,** Theorem proving for functional programmers, *Sparkle: A functional theorem prover,* LNCS **2312**, 2001, 55-71.

[8] **Dévai G.,** Programming language elements for correctness proofs, *Acta Cybernetica* (to appear)

[9] **Dévai G. and Csörnyei Z.,** Separation logic style reasoning in a refinement based language, *Proc. 7th Int. Conf. on Applied Informatics* (to appear)

[10] **Flanagan C., Leino K.R.M., Lillibridge M., Nelson G., Saxe J.B. and Stata R.,** Extended static checking for Java, *Proc. of the ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation (PLDI'2002)*, **37** (2002), 234-245.

[11] **Gregor D. and Schupp S.,** Stllint: lifting static checking from languages to libraries, *Software – Practice & Experience*, **36** (3) (2006), 225-254.

[12] **Horváth Z., Kozsik T. and Tejfel M.,** Extending the Sparkle core language with object abstraction, *Acta Cybernetica*, **17** (2005), 419-445.

[13] **Kapur D., Musser D.R. and Nie X.,** An overview of the Tecton proof system, *Theoretical Computer Science*, **133** (1994), 307-339.

[14] **McDonald J. and Anton J.,** *Specware - producing software correct by construction,* Technical report KES.U.01.3, Kestrel Institute, CA, 2001.

[15] **Meyers S.,** *Effective STL*, Addison-Wesley, 2001.

[16] **Morris J.M.,** A theoretical basis for stepwise refinement and the programming calculus, *Sci. Comput. Program.*, **9** (3) (1987), 287-306.

[17] **Musser D.R.,** *Tecton description of STL container and iterator concepts,* RPI Computer Science Department, Troy, NY, 1998.

[18] **Musser D.R. and Wang C.,** *A basis for formal specification and verification of generic algorithms in the C++ standard template library,* Technical report 95-1, RPI Computer Science Department, Troy, NY, 1995.

[19] **Nipkow T., Paulson L.C. and Wenzel M.,** *Isabelle/HOL – A proof assistant for higher-order logic*, LNCS **2283**, Springer-Verlag, 2002.

[20] **O'Hearn P., Reynolds J. and Yang H.,** Local reasoning about programs that alter data structures, LNCS **2142**, 2001, 1-19.

[21] **Pataki N., Porkoláb Z. and Istenes Z.,** Towards soundness examination of the C++ standard template library, *Electronic Computers and Informatics*, 2006, 186-191.

[22] **Reynolds J.,** Separation logic: a logic for shared mutable data structures, *Logic in Computer Science. Proc. 17th Annual IEEE Symp. on Logic in Computer Science, 2002,* 55-74.

[23] **Siek J. and Lumsdaine A.,** Concept checking: Binding parametric polymorphism in C++, *Proc. of First Workshop on C++ Template Programming, 2000,* 113-125.

[24] **Stroustrup B.,** *The C++ programming language*, special edition, Addison-Wesley, 2000.

[25] **Pásztor Varga K. and Várterész M.,** Usability of some theorem proving systems, *PU.M.A.*, **15** (2-3) (2004), 273-284.

[26] **Wang Ch. and Musser D.R.,** Dynamic verification of C++ generic algorithms, *Software Engineering*, **23** (1997), 314-323.

[27] **Winkler J.,** The frege program prover FPP, *Internationales Wissenschaftliches Kolloquium*, **42** (1997), 116-121.

[28] **Yang H. and O'Hearn P.,** A semantic basis for local reasoning, *Foundations of software science and computation structure*, 2002, 402-416.

G. Dévai and N. Pataki
Department of Programming Languages and Compilers
Eötvös Loránd University
Pázmány Péter s. 1/C
H-1117 Budapest, Hungary
{deva,patakino}@elte.hu