Linking Object-Z with Spec#

Shengchao Qin and Guanhua He
Department of Computer Science, Durham University
{shengchao.qin, guanhua.he}@durham.ac.uk

Abstract

Formal specifications have been a focus of software engineering research for many years and have been applied in a wide variety of settings. Their use in software engineering not only promotes high-level verification via theorem proving or model checking, but also inspires the "correct-by-construction" approach to software development via formal refinement. Although this correct-by-construction method proves to work well for small software systems, it is still a utopia in the development of large and complex software systems. This paper moves one step forward in this direction by designing and implementing a sound linkage between the high level specification language Object-Z and the object-oriented specification language Spec#. Such a linkage would allow system requirements to be specified in a high-level formal language but validated and used in program language level. This linking process can be readily integrated with an automated program refinement procedure to achieve correctness-by-construction. In case no such procedures are applicable, the obtained contract-based specification can guide programmers to manually generate program code, which can then be verified against the obtained specification using any available program verifiers.

Keywords *Formal specification, Object-Z, Spec#, verification, pre/post conditions.*

1 Introduction

Software correctness has become one major concern in software development [22]. Formal methods are expected to play a significant role in ensuring this. On one side, formal methods are adopted to help with existing software products. For instance, formal analysis/verification techniques are used to prove/check that existing programs meet certain desired properties (or to find counterexamples otherwise). On the other hand, formal methods are also expected to fulfill the so-called correct-by-construction approach to software development [29, 1]. This utopia suggests us to make use of a high level formal specification language to specify system requirement and then construct an executable program via a correctness-provable refinement procedure. Although this approach proves to work well for small examples, it seems still a long way to go before we can make use of it to help automate the development of complex software systems.

A realistic solution would be to adopt the correct-by-construction approach in part in the current software development process. If the entire system can be developed solely via the correct-by-construction approach, then it is done. If certain stages (or certain components) can be derived via the correct-by-construction approach (while others cannot), developers only have to deal with the remaining stages (or components). Afterwards we can employ appropriate formal verification tools to ensure the correctness of the remaining part and thus that of the whole system. Conceived by this general idea, we propose in this paper a development framework which supports the automatic translation from high level specifications to program-level specifications. In the subsequent implementation stage, the framework would offer a choice for the developers, which, for example, can be a pure correct-by-construction approach (refinement), or an approach based on manual coding, or a combination of these two. For programmer-generated code, appropriate verification tools are then employed to check that the implementation does fulfill the specification.

To set the scene, let us make the framework more concrete by fixing the high-level and the low-level specification languages. A well-known rigorous high level language is the model-based language Z [34]. To allow object-oriented design to be introduced as early as possible, we would choose Object-Z [33] as our high level specification language. We then choose the Spec# as the low-level specification language. As an extension to the new but popular .Net C# language, Spec# aims at a more cost-effective way to develop and maintain high-quality software [4]. To the best of our knowledge, there is no link between Object-Z and Spec# available yet. Moreover, given the large amount of research effort involved in the Spec# project, we believe soon there will be very good tool support for specifications

12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)
0-7695-2895-3/07 $25.00 © 2007 IEEE

written in Spec#. At the current stage, the Boogie static program verifier [3] can be used to check the consistency between the implemented program and its specification.

The main contribution of this paper is an automated mapping from high-level Object-Z specifications to program-level Spec# specifications which forms the first stage of the above-advocated formal development framework. In the subsequent implementation stage, our framework allows any new or existing formal refinement algorithm to be opted in. In case that such correct-by-construction algorithms are not available and manual implementation has to be adopted, the framework would allow the developers to invoke appropriate verification tools (e.g. Boogie) to formally verify the manually generated code.

The remainder of the paper is organised as follows. Sec 2 briefly introduces both the Object-Z and Spec# specification languages. Sec 3 presents the structural mapping algorithm from Object-Z to Spec#. Sec 4 is devoted to the tool development. Related work is presented in Sec 5. Conclusion and future work follow afterwards.

2 A Brief Overview of Object-Z and Spec#

In this section, we give a preliminary introduction to both the high level specification language Object-Z and the program-level specification language Spec#.

2.1 The Object-Z Language

Object-Z [33] is an object-oriented extension of the model-based specification language Z. An object-oriented specification describes a system as a collection of interacting objects, each of which has a pre-described structure and behaviour [14]. The object-oriented approach to creating a system is currently the popular approach as it is arguable truer to the real world and therefore easier to understand than procedural and functional programming languages. This extension to Z eases the integration of specification with other common software engineering methods such as UML, as exemplified in e.g. [23, 27]. The modularity that Object-Z introduces does improve the clarity of a specification.

The key feature of Object-Z, as always with the object-oriented approach, is its focus on classes. They are represented by use of class schemas, an example of which is shown in Figure 1 (taken from [13]). The example includes many of the basic Object-Z features. The first line of the class schema forms a visibility list: a list of all the operation specifications and variables that are visible to outside of the class. If no visibility list is present then all variables and operations are visible by default. Axiomatic definitions are used when a variable to be known throughout a specification, i.e. a global variable, is to be declared along with some constraints on the variable in the form of predicates, an example of which is the definition of the variable *limit*. A predicate can consist of any first-order logic or standard set theory expression. Therefore, the $=$ symbol indicates equality rather than assignment. A state schema uses its declarations to introduce variables and its predicates to introduce object invariants, as shown in the anonymous schema introducing variable *balance*. If the predicates that make up the initial schema, *INIT*, and all the object invariants hold then the model is said to be in its initial configuration. The remaining schemas within the class schema in the example are all operational schemas. Operational schemas consist of declarations to introduce required variables, e.g. *amount?* in the *withdraw* operation, and predicates defining a method contract, e.g. the two predicates given in *withdraw* operation. If the method contract results in a change in a primary variable's value then that variable must be included in the operational schema's delta list, and the primary variable's name within the predicate must end with the $'$(primed) symbol, e.g. $balance'$. Other naming conventions within operational schemas are that a variable name ending in the ? symbol, e.g. *amount?*, indicates an input variable, while a variable name ending in the ! symbol, e.g. *funds!*, indicates an output variable. The expression used to define the *withdrawConfirm* operation is a composite operation. This is an operation defined in terms of a combination of other operations that can be combined in several different ways. For details of the different combination methods and further details of more advanced features of Object-Z specifications, such as inheritance, object containment and polymorphism, see [33, 13].

2.2 The Spec# Language

Spec# [4] is an extension of Microsoft's .NET framework programming language C#, and the development environment currently offering support for the Spec# programming system is Microsoft Visual Studio. Spec# is a programming system incorporating specification concepts, rather than a formal specification language. Features the Spec# system has introduced into C# include object invariants [2], non-null types [16], and method contracts. Object invariants are "specifications that constrain the value space of the implementation's data" [4], i.e. conditions that must always hold on all objects of that class, and are declared with the *invariant* keyword. In some cases it is possible that a series of statements in a method will break an invariant. To be able to handle such a situation, Spec# has introduced the block statement identified by the *expose* keyword. Within an expose statement invariants can be broken, as long as all invariants hold again by the end of the statement. The concept of non-null types allows programmers to discriminate between expressions that may evaluate to null and those that

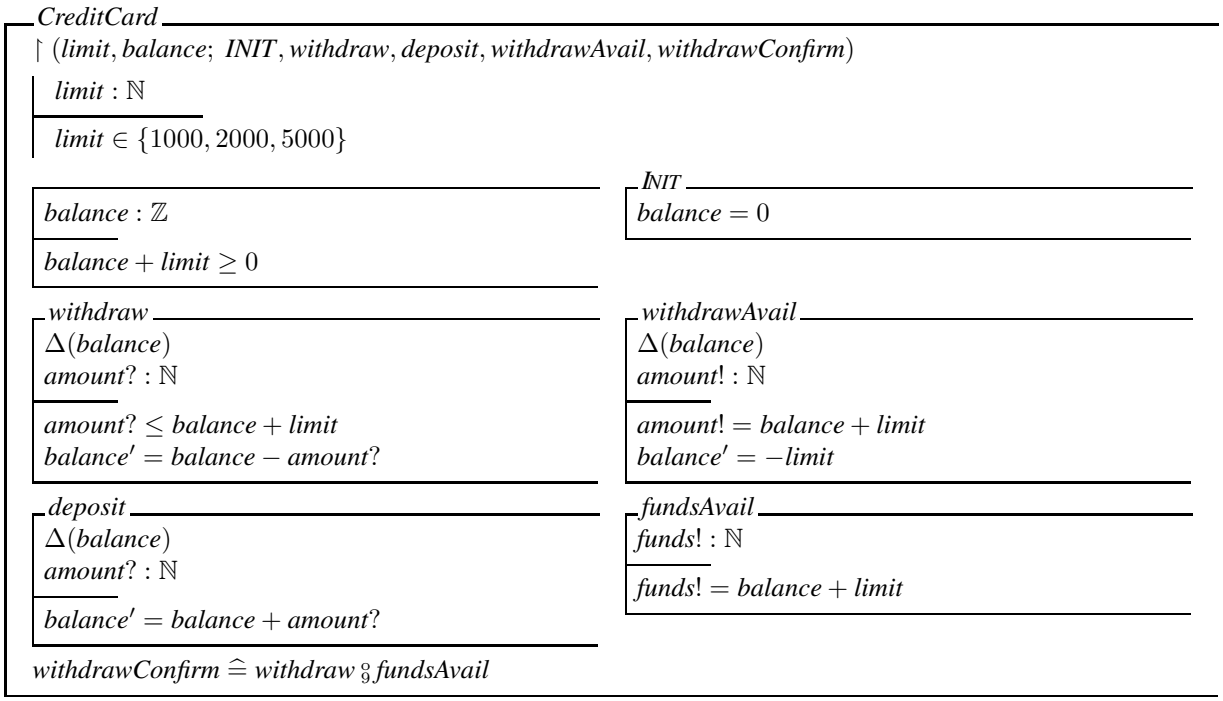


Figure 1. The CreditCard Class in Object-Z

are sure not to [4].

Method contracts consist of a series of conditions known as pre-conditions, post-conditions and frame conditions. Pre-conditions, preceded by the *requires* keyword, define the state in which the system must be to be able to call the method, while post-conditions, preceded by the *ensures* keyword, describe the state in which the method is allowed to return. Frame conditions restrict which pieces of the program state a method implementation is allowed to modify and are preceded by the *modifies* keyword. A full introduction of features of the Spec# programming system and resulting complexities are detailed in [4].

In addition to extending the C# programming language, Spec# also introduces the Boogie verification tool. Boogie operates by verifying the object code rather than the source code, thereby allowing code written in languages other than Spec# to be verified. It can perform both static verification and run-time checks. Static verification is where Boogie uses automated theorem proving to verify a source program. For example, Boogie can verify methods according to specified post-conditions and can ascertain whether the post-conditions are required or not, i.e. if the verification is successful then the run-time checks are excessive since they would never fail. The run-time checks are performed on any pre- or post-conditions, which are turned into inline code that is identifiable as method contract, to ensure that they hold.

3 The Mapping from Object-Z to Spec#

This section is devoted to the mapping process from Object-Z to Spec#. We will present the formal mapping via a few definitions and then state the soundness properties.

To present the design of the mapping, we shall focus on a core subset of Object-Z specification language. The abstract syntax of this core subset is given in Figure 2. The actual concrete syntax that we have implemented subsumes this abstract syntax.

Note that an Object-Z specification is formed by a sequence of class declarations. A class declaration includes a visibility list, an (optional) superclass, some (optional) axiomatic (local) definitions, a state schema, an initial schema, and some state operations. An example of a class declaration is given earlier in Figure 1.

One significant incompatibility between Object-Z and Spec# is that Object-Z supports multiple inheritance while Spec# does not. To simplify the design, we constrain the subset of Object-Z to adopt only single inheritance. This simplification was also adopted by [18, 28].

The translation from Object-Z to Spec# is conducted in a structural manner, i.e., every class declaration in an Object-Z specification is mapped to a class definition in the corre-

OZSpecification ::= CDecl*
CDecl ::= ↾ VisibList; InheritC; Local*; State; INIT; Op*
VisibList ::= VisibAttr; VisibOp
VisibAttr ::= AttrName*
VisibOp ::= OpName*
InheritC ::= **Inherits** CName
Local ::= VarDecl* [• Predicate]
State ::= VarDecl* [Δ VarDecl*] [• Predicate]
VarDecl ::= v : **T**
Op ::= OpName :: OpExp | v.OpName
| [v.]OpName ⨾ [v.]OpName
| [v.]OpName ∧ [v.]OpName
| [v.]OpName [] [v.]OpName
| [v.]OpName || [v.]OpName
OpExp ::= Δ(AttrName*), VarDecl* • Predicate

Figure 2. The Syntax of Object-Z

sponding Spec# program. We present this structural mapping $\mathcal{L}$ in the following definitions:

Definition 1 (The Structural Mapping $\mathcal{L}$: Spec) *For a given Object-Z specification*

$$ozs \mathrel{\widehat=} OZC_1;\ \ldots;\ OZC_n$$

the mapping algorithm $\mathcal{L}$ generates in Spec# a specification

$$sss = \mathcal{L}(ozs) = SSC_1;\ \ldots;\ SSC_n$$

where each Object-Z class declaration OZC_i is mapped to a Spec# class SSC_i, i.e., $\mathcal{L}(OZC_i) = SSC_i$, for $i = 1, .., n$. (See Definition 2.) □

Definition 2 (The Structural Mapping $\mathcal{L}$: Class) *For a given Object-Z class*

$$OZC \mathrel{\widehat=} ↾ vl;\ inherits\ \widehat{OZC};\ ldef;\ state;\ init;\ op_{j=1}^{m}$$

*where **predicate**$(ldef) = pred_1$, and **predicate**$(state) = pred_2$, **predicate**$(init) = pred_3$, [1] the mapping algorithm $\mathcal{L}$ generates the corresponding Spec# class declaration $SSC = \mathcal{L}(OZC)$ using the following rules:*

1. *SSC has $\mathcal{L}(\widehat{OZC})$ as its immediate superclass.*
2. *All variables declared in ldef and state become instance variables newly declared in SSC. Constants declared in ldef become constants newly declared in SSC.*
3. *SSC has $pred_1 \wedge pred_2$ as its object invariant. That is, the clause*

   invariant $pred_1 \wedge pred_2$;

   appears in SSC (following the instance variables).
4. *A default constructor method (contract) is generated with $pred_3$ as its postcondition.* [2]
5. *For $j = 1, .., m$, the operation op_j is mapped to an instance method $meth_j = \mathcal{L}(OZC.op_j)$ of the class SSC. (See Definition 3.)*
6. *If the visibility list vl is empty, all instance variables and operations are made* public. *Otherwise, only those mentioned in vl are made* public *while others are made* private. □

Definition 3 (The Structural Mapping $\mathcal{L}$: Operations) *Each operation op from an Object-Z class OZC is mapped to a method $meth = \mathcal{L}(OZC.op)$ (with an empty method body) in the corresponding Spec# class $\mathcal{L}(OZC)$ according to the following rules:*

1. *Case (**basic**) $op \mathrel{\widehat=} mn :: \Delta(\underline{r}),\ \underline{u?}{:}\underline{S},\ \underline{v!}{:}\underline{T} \bullet Pred$. Suppose $\underline{sa}$ denotes the set of secondary variables declared in class OZC. The method $\mathcal{L}(OZC.op)$ generated by the mapping algorithm will be*

   void $mn\ (\underline{S}\ \underline{u},\ \underline{out}\ \underline{T}\ \underline{v})$ ***requires*** pre; ***ensures*** $post$; ***modifies*** $\underline{r} \cup \underline{sa}\ \{\}$ [3]

   Note that the conditions pre and post are generated from Pred in a way similar to what Diller used in [11]:

   $$\begin{array}{lcl} pre & \mathrel{\widehat=} & \exists\, \underline{a}', \underline{v!} \cdot Pred[\underline{u}/\underline{u?}][\underline{this.a}/\underline{a}] \\ post & \mathrel{\widehat=} & Pred[\underline{old(this.a)}/\underline{a}][\underline{this.a}/\underline{a}'][\underline{v}/\underline{v!}] \end{array}$$

   Where $\underline{a}$ denote all state variables that may be changed by op, which include the primary variables $\underline{r}$ and all secondary variables declared in the current class. Note also that, due to different notations required in the source and target languages, we rename every a to $old(this.a)$ and then every a' to $this.a$ for the postcondition. [4] *We also eliminate the ? (resp. !) associated with all input (resp. output) variables as Spec# does not use it.*
2. *Case ($\fatsemi$) $op \mathrel{\widehat=} op_1 \fatsemi op_2$. Suppose*

   $$\begin{array}{l} \mathcal{L}(OZC.op_1) \mathrel{\widehat=} \textbf{\textit{void}}\ mn_1(\underline{T_1\ u}, \underline{out\ T_2\ v}) \\ \quad \textbf{\textit{requires}}\ pre_1;\ \textbf{\textit{ensures}}\ post_1;\ \textbf{\textit{modifies}}\ r_1\ \{\} \\ \mathcal{L}(OZC.op_2) \mathrel{\widehat=} \textbf{\textit{void}}\ mn_2(\underline{T_2\ v}, \underline{out\ T_3\ w}) \\ \quad \textbf{\textit{requires}}\ pre_2;\ \textbf{\textit{ensures}}\ post_2;\ \textbf{\textit{modifies}}\ r_2\ \{\} \end{array}$$

[1] The function ***predicate***(...) is to extract the predicate out of a local axiomatic definition, a state schema or an initial schema.

[2] Note that object invariant is not added into the postcondition explicitly but is enforced implicitly in Spec#.

[3] This provides one possible solution. An alternative solution we used in our prototype implementation is to map one of the output variables, say the first one (if there are more than one), to the result which will be returned by the method.

[4] Here we attach 'this' to every instance variable to simplify the presentation in a later case. This is optional in our implementation.

The method generated for op is as follows:

$$\mathcal{L}(OZC.op) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn(\underline{T_1\ u}, \underline{\textit{out}\ T_3\ w})$$
$$\textbf{\textit{requires}}\ pre;\ \textbf{\textit{ensures}}\ post;\ \textbf{\textit{modifies}}\ r$$
$$\{\underline{T_2\ v};\ mn_1(\underline{u}, \underline{\textit{out}\ v});\ mn_2(\underline{v}, \underline{\textit{out}\ w});\ \}$$

where $pre \mathrel{\widehat{=}} pre_1 \wedge \exists \underline{v} \cdot (post_1 \Rightarrow pre_2)$, *and* $post \mathrel{\widehat{=}} \exists \underline{v} \cdot ((post_1 \wedge no\chi(r_2 - r_1)) \circ (post_2 \wedge no\chi(r_1 - r_2)))$, *and* $r \mathrel{\widehat{=}} r_1 \cup r_2$. *The predicate* $no\chi(V)$ *indicates that all variables in V remain unchanged. It is defined as follows:*

$$\begin{array}{lcl} no\chi(\{\}) & \mathrel{\widehat{=}} & true \\ no\chi(\{v\} \cup V) & \mathrel{\widehat{=}} & (v = old(v)) \wedge no\chi(V) \end{array}$$

This auxiliary predicate is vital here for the generation of the correct postcondition for the composite operation, as the modifies frame for the composite operation can be larger than that of its constituent operations. Given two predicates $P_1(\underline{old(v)}, \underline{v})$ *and* $P_2(\underline{old(v)}, \underline{v})$, *the sequential composition of them is defined as follows (as in [21]):*

$$P_1 \circ P_2 \mathrel{\widehat{=}} \exists \underline{v_0} \cdot P_1(\underline{old(v)}, \underline{v_0}) \wedge P_2(\underline{v_0}, \underline{v})$$

3. *Case* ($\wedge$) $op \mathrel{\widehat{=}} op_1 \wedge op_2$. *Suppose*

$$\mathcal{L}(OZC.op_1) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn_1(\underline{T_1^i\ u_1}, \underline{\textit{out}\ T_1^o\ v_1})$$
$$\textbf{\textit{requires}}\ pre_1;\ \textbf{\textit{ensures}}\ post_1;\ \textbf{\textit{modifies}}\ r_1\ \{\}$$
$$\mathcal{L}(OZC.op_2) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn_2(\underline{T_2^i\ u_2}, \underline{\textit{out}\ T_2^o\ v_2})$$
$$\textbf{\textit{requires}}\ pre_2;\ \textbf{\textit{ensures}}\ post_2;\ \textbf{\textit{modifies}}\ r_2\ \{\}$$

For simplicity, let us assume the parameter lists are disjoint, i.e., $\{\underline{u_1}, \underline{v_1}\} \cap \{\underline{u_2}, \underline{v_2}\} = \varnothing$. *Let* $\{\underline{T^i\ u}\}$ *denote* $\{\underline{T_1^i\ u_1}\} \cup \{\underline{T_2^i\ u_2}\}$, *and* $\{\underline{T^o\ v}\}$ *denote* $\{\underline{T_1^o\ v_1}\} \cup \{\underline{T_2^o\ v_2}\}$. *The method generated for op is as follows:*

$$\mathcal{L}(OZC.op) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn(\underline{T^i\ u}, \underline{\textit{out}\ T^o\ v})$$
$$\textbf{\textit{requires}}\ pre;\ \textbf{\textit{ensures}}\ post;\ \textbf{\textit{modifies}}\ r\ \{\}$$

where $r \mathrel{\widehat{=}} r_1 \cup r_2$, $pre \mathrel{\widehat{=}} pre_1 \wedge pre_2$, *and* $post \mathrel{\widehat{=}} post_1 \wedge post_2$.

4. *Case* ($[]$) $op \mathrel{\widehat{=}} op_1 [] op_2$. *Suppose*

$$\mathcal{L}(OZC.op_1) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn_1(\underline{T^i\ u}, \underline{\textit{out}\ T^o\ v})$$
$$\textbf{\textit{requires}}\ pre_1;\ \textbf{\textit{ensures}}\ post_1;\ \textbf{\textit{modifies}}\ r_1\ \{\}$$
$$\mathcal{L}(OZC.op_2) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn_2(\underline{T^i\ u}, \underline{\textit{out}\ T^o\ v})$$
$$\textbf{\textit{requires}}\ pre_2;\ \textbf{\textit{ensures}}\ post_2;\ \textbf{\textit{modifies}}\ r_2\ \{\}$$

Note that it is required that both constituent methods have the same set of parameters. The method generated for op is as follows:

$$\mathcal{L}(OZC.op) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn(\underline{T^i\ u}, \underline{\textit{out}\ T^o\ v})$$
$$\textbf{\textit{requires}}\ pre;\ \textbf{\textit{ensures}}\ post;\ \textbf{\textit{modifies}}\ r\ \{\}$$

where $r \mathrel{\widehat{=}} r_1 \cup r_2$, $pre \mathrel{\widehat{=}} pre_1 \vee pre_2$, *and* $post \mathrel{\widehat{=}} old(pre_1) \wedge post_1 \vee old(pre_2) \wedge post_2$. *The postcondition states that if a particular branch is chosen for actual execution, then the corresponding postcondition should be guaranteed in the post-state.*

5. *Case* ($\|$) $op \mathrel{\widehat{=}} op_1 \parallel op_2$. *Suppose*

$$\mathcal{L}(OZC.op_1) \mathrel{\widehat{=}}$$
$$\textbf{\textit{void}}\ mn_1(\underline{T_1\ u_1}, \underline{S_1\ w_1}, \underline{\textit{out}\ S_2\ w_2}, \underline{\textit{out}\ T_3\ v_1})$$
$$\textbf{\textit{requires}}\ pre_1;\ \textbf{\textit{ensures}}\ post_1;\ \textbf{\textit{modifies}}\ r_1\ \{\}$$
$$\mathcal{L}(OZC.op_2) \mathrel{\widehat{=}}$$
$$\textbf{\textit{void}}\ mn_2(\underline{T_2\ u_2}, \underline{S_2\ w_2}, \underline{\textit{out}\ S_1\ w_1}, \underline{\textit{out}\ T_4\ v_2})$$
$$\textbf{\textit{requires}}\ pre_2;\ \textbf{\textit{ensures}}\ post_2;\ \textbf{\textit{modifies}}\ r_2\ \{\}$$

According to the semantic definition of parallel composition in Object-Z, the parallel composition will identify some input variables from either operation to be equated with some output variables (with same basenames) in another operation. In the above, the input variables $\underline{w_1}$ *out of* $\mathcal{L}(op_1)$ *will be equated with the output variables* $\underline{w_1}$ *in* $\mathcal{L}(op_2)$ *and be hidden from the final result. Similar for the variables* $\underline{w_2}$. *Thus, the method generated for op is as follows:*

$$\mathcal{L}(OZC.op) \mathrel{\widehat{=}}$$
$$\textbf{\textit{void}}\ mn(\underline{T_1\ u_1}, \underline{T_2\ u_2}, \underline{\textit{out}\ T_3\ v_1}, \underline{\textit{out}\ T_4\ v_2})$$
$$\textbf{\textit{requires}}\ pre;\ \textbf{\textit{ensures}}\ post;\ \textbf{\textit{modifies}}\ r\ \{\}$$

where $pre \mathrel{\widehat{=}} \exists \underline{w_1}, \underline{w_2} \cdot pre_1 \wedge pre_2$, $post \mathrel{\widehat{=}} \exists \underline{w_1}, \underline{w_2} \cdot post_1 \wedge post_2$, *and* $r \mathrel{\widehat{=}} r_1 \cup r_2$.

6. *Case (Aggregation)* $op \mathrel{\widehat{=}} x.op_1$. *Suppose* $x :: \downarrow X$, *and*

$$\mathcal{L}(X.op_1) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn(\underline{T_1\ u}, \underline{\textit{out}\ T_2\ v})$$
$$\textbf{\textit{requires}}\ pre_1;\ \textbf{\textit{ensures}}\ post_1;\ \textbf{\textit{modifies}}\ r_1\ \{\}$$

The method generated for op is as follows:

$$\mathcal{L}(OZC.op) \mathrel{\widehat{=}} \textbf{\textit{void}}\ mn(\underline{T_1\ u}, \underline{\textit{out}\ T_2\ v})$$
$$\textbf{\textit{requires}}\ pre;\ \textbf{\textit{ensures}}\ post;\ \textbf{\textit{modifies}}\ r$$
$$\{v.op_1(\underline{u},\ \underline{\textit{out}\ v});\ \}$$

where $pre \mathrel{\widehat{=}} pre_1[x/this]$, *and* $post \mathrel{\widehat{=}} post_1[x/this]$. *If* $this \in r_1$ *then* $r = \{x\}$ *else* $r = \varnothing$. [5]

Remark: Object-Z does not impose behavioural subtyping in inheritance hierarchies; instead, it allows even arbitrary redefinition of operations in subclasses. The above mapping rule for aggregation works under the assumption that behavioural subtyping has been

[5] The generation of frame conditions can be very tricky. In Spec#, an *ownership* based method is proposed to deal with frame conditions when aggregate objects are involved [2] . For simplicity, here we do not take into account the permission to modify fields of aggregate objects.

imposed in the inheritance hierarchies in the Object-Z specifications.[6] *This is a proof obligation imposed on the specifier. For those specifications where behavioural subtyping does not hold, the above rule will not work. However, it is possible to develop a more general way to handle this issue (thanks to Graeme Smith) but we will not elaborate it in this paper.*

7. *Case (Composition with Aggregation) In case that a composition is made of aggregate operations, i.e., $op \mathrel{\widehat{=}} x.op_1 \oplus y.op_2$, where x and y are state variables of OZC, and $\oplus$ is from $\{\fatsemi, [], \wedge, \|\}$. This can be handled in a way similar to case 2-5 above. The pre/post conditions for the resulting method can be generated in the same way as in case 2-5, except that we need to rename '*this*' in pre/post of op_1 to 'x', and '*this*' in pre/post of op_2 to 'y'. We generate the modifies clause in a way similar to case 6.* □

Note that the mapping algorithm has not filled in the method bodies in this stage. They are left to the next development stage either by formal refinement or by manual coding. It is worth mentioning that in the case ($\fatsemi$) and the aggregation case, the code for the composite method can be generated automatically by invoking the constituent methods, though in the aggregation case, the actual method to be called will be determined dynamically due to polymorphism. In other cases, we have to generate the code for the composite method directly.

We now state the soundness properties of our mapping process $\mathcal{L}$. We will make use of a refinement relation analogous to that defined in [19].

Informally speaking, the Spec# specification generated by the mapping process is better than (or at least as good as) its source Object-Z specification. To formally state this, we will need to go to the meta level (semantic level). However, in order to focus better on the soundness and keep this part short, we will not present the formal semantics for the source/target languages here. Instead, we assume they are already available for use. Intuitively, the semantics of a specification is the set of programs (i.e. implementations) that satisfy the specification.

We will make use of the following definitions in the soundness theorem.

Definition 4 *We say a Spec# method meth (with contract $(pre, post)$) refines (or implements) an Object-Z operation op, denoted as $op \sqsubseteq_{com} meth$, if every implementation of the method contract is also an implementation of the operation. That is,*

$$\forall\, com \;\cdot\; \vdash \{pre\}\; com\; \{post\} \;\Rightarrow\; \rho\, com \in [\![op]\!]$$

[6] Suppose class Y inherits class X, and operation op is defined in X but redefined in Y. Behavioural subtyping requires the following: $pre(X.op) \Rightarrow pre(Y.op)$, and $post(Y.op) \Rightarrow post(X.op)$.

Note that for simplicity, we assume the (in and out) parameters in meth and the variables declared in op have the same set of base names. The mapping ρ is to map every input variable u to u?, and every output variable v to v!. □

Definition 5 *We say a Spec# class SSC refines (or implements) an Object-Z class OZC, denoted as $OZC \sqsubseteq_{class} SSC$, if (1) the superclass of SSC refines the superclass of OZC; (2) the instance variables defined in SSC are identical to those variables defined in OZC; (3) for every operation op in OZC, there is a method meth in SSC such that $op \sqsubseteq_{com} meth$.* □

Definition 6 *Given an Object-Z specification $ozs \mathrel{\widehat{=}} OZC_1;\ \ldots;\ OZC_n$, and a Spec# specification $sss = \mathcal{L}(ozs) = SSC_1;\ \ldots;\ SSC_n$, we say sss refines (or implements) ozs, denoted as $ozs \sqsubseteq sss$, if $OZC_i \sqsubseteq_{class} SSC_i$, for $i = 1, .., n$.* □

The following theorem states the soundness of the mapping algorithm.

Theorem 1 (Soundness) *The mapping algorithm $\mathcal{L}$ (defined in Definition 1,2,3) is sound. That is, for a given Object-Z specification ozs and its corresponding Spec# specification $sss = \mathcal{L}(ozs)$, we have $ozs \sqsubseteq sss$.* □

4 The Tool Development

In this section we discuss the development of the prototype tool.

4.1 Overview of the Implementation

The graphical representation for Object-Z is good for visualisation but does not seem to be a favorable input format for the tool to implement. So we decide to go for a plain-text based representation. Instead of using XML-based representation (e.g. as did in [18]), we take the LaTeX representation for Object-Z as the input syntax for our system. Thanks to the LaTeX package for Object-Z [24], we can easily obtain the graphical representation from the LaTeX format.

We have chosen the functional language Haskell [31] to be our implementation language as Haskell is a very good language for fast prototyping and it has a parser combinator library, called Parsec [25] which can be used to build the parser very quickly. The implemented tool is composed of three components, namely, a parser, a mapper, and a printer, which are built as three Haskell functions, *parsing*, *mapping*, and *printing*, respectively. The function *parsing* parses the input Object-Z LaTeX file to an Object-Z syntax tree, the function *mapping* transforms the Object-Z syntax tree into a Spec# syntax tree, while the function *printing* is a pretty printer which converts the Spec# syntax tree to a

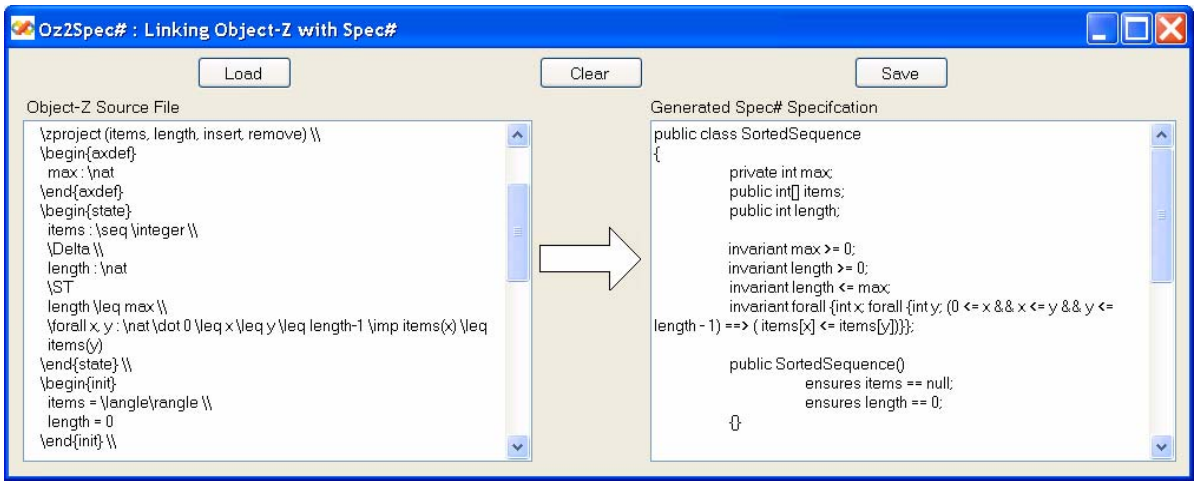


Figure 3. Oz2Spec#: Linking Object-Z with Spec#

Spec# program. In a nutshell, the system can be viewed as a composition of the above three functions:

$$OZ2SSp = printing \circ mapping \circ parsing$$

We have also built a graphical user interface (GUI) in Microsoft .Net Framework. A snapshot of the GUI is shown in Figure 3.

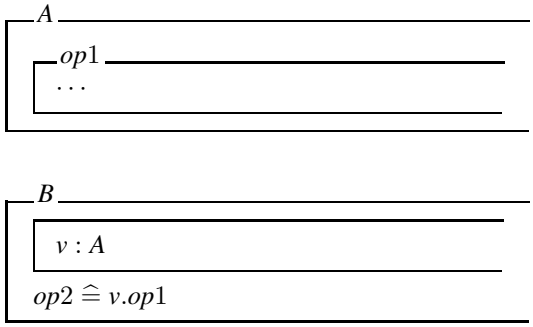
In our system, the parser component parses an Object-Z source file into a list of Object-Z class declaration trees, which are then converted to a list of Spec# class declaration trees by the mapper component. Finally, they are pretty-printed into one source file which is the Spec# program. We will elaborate a bit more on the mapping function.

4.2 Mapping

The mapping function from Object-Z to Spec# forms the core part of our implementation. It takes as a list of Object-Z syntax trees and generates as output a list of Spec# syntax trees. As we described in section 3, the mapping function is conducted in a structural manner, i.e., it maps an Object-Z class declaration to a corresponding Spec# class declaration where all the constituents from the source and the target have the similar correspondence.
The mapper component mainly consists of the following functions:

- A function *predicate* is used to extract predicates out of *local definitions*, the *state schema*, or the *INIT* schema. While a function *variable* is to extract variables out of *local definitions* and the *state schema*. The visibility property (*public* or *private*) for instance variables in Spec# classes is decided in accordance with the *visibility list* given in the source Object-Z specification.
- A function *mapConstructor* maps the *INIT* schema to a *Constructor* method. It also transfers all the predicates in *INIT* to the post-condition for the constructor method.
- A function *mapMethods* maps Object-Z *operations* to Spec# *methods* using the algorithm given in section 3.

Our system maintains the information about class dependencies (mainly for inheritance and aggregation) as such information is required during the mapping. For instance, if a class B has an instance variable v of type A, where A is another class declared in the specification, then class B depends on class A. In such a scenario an operation of class B may be defined in terms of an operation of class A via the variable v:



The class dependency information is stored in a class table in our implementation. During mapping, when an operation refers to another operation in another class, we can search through the class table to find that class from which we can obtain the required method contracts in order to generate the contract for the method in the depending class. As mentioned earlier, one way to support polymorphism (e.g.

in the above class B, v might be defined as $v : \downarrow A$) is to impose a proof obligation on the specifier such that behavioural subtyping holds in inheritance hierarchies.

Part of the Spec# specification generated from the CreditCard example in Figure 1 are shown in Figure 4. Another example is given in the appendix.

```
  public class CreditCard
{
    public int limit;
    public int balance;

    invariant ...;
    invariant balance + limit >= 0;

    public CreditCard()
      ensures balance == 0;
    {}

    public void withdraw(int amount)
      requires amount >= 0;
      requires amount <= balance + limit;
      ensures balance == old(balance) - amount;
      modifies balance;
    {}

    private int fundsAvail()
        ensures result == balance + limit;
    {}

    public int withdrawConfirm(int amount)
      requires amount >= 0;
      requires amount <= balance + limit;
      ensures balance == old(balance)-amount;
      ensures result == old(balance)-amount+limit;
      modifies balance;
    {}
    ...
}
```

Figure 4. Part of the Spec# Code generated for the CreditCard Example

5 Related Work

As an object-oriented extension to the state-based specification language Z [34], Object-Z [14, 33, 13] has been a well-studied high level formal specification language. It has also been used in many applications, e.g., the modelling of Java concurrency [15], agent modelling [20]. There are also limited tool support available for Object-Z, e.g. [35]. Object-Z has also been extended to support modelling of different complex systems, e.g., integrating with CSP [32, 17], relating with Pi-calculus [36], blending with Timed CSP [26], coupling with Timed Automata [12].

Compared to Object-Z, Spec# [4] is a rather new specification language. It extends also new but already very popular .Net programming language C# with various specification mechanisms, including method contracts, object invariants [2], assertions, non-null types [16] etc. The Boogie tool [3] is proposed to statically verify specifications written in Spec#. Apart from static verification, Spec# also has good support for runtime assertion checking. JML [6] is a similar program specification language which is based on Java.

A closely related work is the Object-Z to JML parser by Giles [18], where an implementation for a mapping from Object-Z to JML is reported. Similar to ours, their mapping also ignores multiple inheritance in Object-Z. While they require Object-Z input files to be given in XML format and generate JML programs, we take as input the latex format of Object-Z specifications which can be easily compiled to graphical representation. In [18], the design of the mapping is presented informally, while in this paper, we give a formal definition of the mapping algorithm, correctness of which is provable. One feature of Object-Z is that it allows multiple output variables in its operations. This becomes a problem in the translation from Object-Z to JML [18], in which they have to use wrapper classes or translate the multiple output variables into a single vector output. However, it is not a problem in our translation from Object-Z to Spec#, thanks to the feature that C# (hence Spec#) supports *output* parameters which can be used to represent multiple outputs.

The work in [28] investigated the linking between CSP-OZ and UML/Java. Their aim is to generate part of the CSP-OZ specifications from an initially developed UML model, and then transform the CSP-OZ specification to Java, with CSP-OZ playing an intermediate role and serving to verifying correctness of the UML model. Although they have presented their approach via a large case study, no general transformation algorithm is reported. Cavalcanti and Sampaio [7] reported another approach to combining CSP-OZ with Java, where CSP-OZ specifications are translated into CTJ, an extension of Java with CSP-like processes and channels, using a number of refinement rules.

6 Conclusion and Future Work

To support the correct-by-construction approach in complex software development, we advocate the use of a formal framework in which a high-level formal language (e.g. Object-Z) is used for requirement specifications, and a lower level language (e.g. Spec#) is employed for program specifications; in the implementation stage formal refinement procedures can be integrated to transform the obtained (more concrete) program specifications to actual program code. If such provably-correct refinement procedure is not applicable or can only be applied in part, the manually coded part can then be verified by an integrated prover. We have designed in this paper a formal mapping from the high level language Object-Z to the program-level language Spec#, and built a prototype tool to implement the mapping.

Our future work is to complete the above-mentioned development framework. On one hand, we shall explore formal refinement calculi that can be used for the refinement from the concrete program specifications (where only

method contracts are available) to the actual implementations (where method bodies are filled up). We should try to make use of existing refinement tools if possible, or build our own one. On the other hand, we shall look into appropriate program verifiers which can be used to verify the final program code especially when the method bodies are manually programmed. A good candidate is the Boogie static verifier [3] by Microsoft research team. Boogie generates from a Spec# specification verification conditions (VCs) which are discharged by an underlying theorem prover. At the moment, Boogie calls the Simplify theorem prover [10] but hopefully they will integrate Boogie with their own prover in the near future. Depending on the properties that may be involved in the system to develop, more advanced provers may be integrated into our framework, e.g. the termination prover [5, 9], the shape and size verifiers [8, 30], etc. On the other hand, it would be very interesting to investigate how to extend our mapping algorithm so that it works for a more powerful specification language like TCOZ [26], which can be a challenging future work as well.

Acknowledgement

The work is supported in part by the EPSRC funded project EP/E021948/1. We would like to thank the anonymous reviewers for their valuable comments. We thank Graham Odds for a Java-based prototype implementation in his final year project which raised many interesting issues and motivated us to investigate them more deeply. We are also grateful to Graeme Smith and Jun Sun for helpful discussion on Object-Z.

References

[1] R. J. Back, A. Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[2] M. Barnet, R. DeLine, M. Fahndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs*, 3(6), June 2004.

[3] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *International Symposium on Formal Methods for Components and Objects (FMCO)*, 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[5] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *ACM Symposium on Principles of Programming Languages (POPL)*, Nice, France, January 2007.

[6] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[7] A. Cavalcanti and A. Sampaio. From CSP-OZ to Java with Processes. In *Workshop on Formal Methods for Parallel Programming, held in conjunction with International Parallel and Distributed Processing Symposium*. IEEE CS Press, 2002.

[8] W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. St. Louis, Missouri, May 2005.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, June 2007.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[11] A. Diller. Z and Hoare Logics. In *Z User Workshop*, 1991.

[12] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and W. Yi. Timed Patterns: TCOZ to Timed Automata. In *Formal Methods and Software Engineering (ICFEM04)*, Seattle, WA, USA.

[13] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.

[14] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[15] R. Duke, L. Wildman, and B. Long. Modelling Java Concurrency with Object-Z. In *International Conference on Software Engineering and Formal Methods (SEFM 2003)*. IEEE Computer Society Press, 2003.

[16] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003*, pages 302–312, Anaheim, CA, USA, 2003.

[17] C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In *Formal Methods for Open Object-Based Distributed Systems*. Chapman & Hall, 1997.

[18] N. Giles. An Object-Z to JML Parser, 2002. MSc thesis, Imperial College.

[19] J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Second Asian Symposium on Programming Languages and Systems (APLAS'04)*, pages 415–436, 2004.

[20] V. Hilaire, O. Simonin, A. Koukam, and J. Ferber. A Formal Approach to Design and Reuse Agent and Multiagent Models. In *Agent Oriented Software Engineering (AOSE 04)*, Lecture Notes in Computer Science, 2004.

[21] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[22] C.A.R. Hoare and J. Misra. Verified software: Theories, tools, experiments: Vision of a grand challenge project. In *The VSTTE conference – Verified Software: Theories, Tools, Experiments*, ETH Zurich, October 2005.

[23] S.-K. Kim and D. Carrington. Formalizing the uml class diagram using object-z. In *Second International Conference on The Unified Modeling Language: Beyond the Standard (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98, 1999.

[24] P. King. Printing Z and Object-Z LATEX documents. May 1990.

[25] D. Leijen. Parsec, a fast combinator parser. November 2001.

[26] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.

[27] H. Miao, L. Liu, and L. Li. Formalizing UML models with Object-Z. In *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 523–534, 2002.

[28] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *4th International Conference on Integrated Formal Methods, (IFM 2004), Canterbury, UK*, pages 267–286, April 2004.

[29] C.C. Morgan. *Programming from Specifications*. Prentice Hall International, second edition, 1994.

[30] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. Nice, France, January 2007.

[31] S. Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[32] G. Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[33] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[34] M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, second edition, 1992.

[35] J. Sun, J.S. Dong, J. Liu, and H. Wang. Object-Z Web Environment and Projections to UML. In *International World Wide Web Conference (WWW-10)*. ACM Press, 2001.

[36] K. Taguchi, J.S. Dong, and G. Ciobanu. Relating Pi-calculus to Object-Z. In *IEEE International Conference on Engineering Complex Computer Systems (ICECCS'04)*. IEEE Press, 2004.

SortedSequence

$\upharpoonright (items, length, insert, remove)$

$max : \mathbb{N}$

$items : \operatorname{seq} \mathbb{Z}$
Δ
$length : \mathbb{N}$

$length \leq max$
$\forall x, y : \mathbb{N} \bullet 0 \leq x \leq y \leq length - 1$
$\quad \Rightarrow items(x) \leq items(y)$

INIT

$items = \langle\rangle$
$length = 0$

insert

$\Delta(items)$
$n? : \mathbb{Z}$

$length < max$
$n?$ **in** $items'$
$\forall x, y : \mathbb{N} \bullet 0 \leq x \leq y \leq length' - 1$
$\quad \Rightarrow items'(x) \leq items'(y)$

...

```
public class SortedSequence {
    private int max;
    public int[] items;
    public int length;

    invariant max > 0;
    invariant length >=0 && length <= max;
    invariant forall {int x; forall {int y;
              (0<=x && x<=y && y<=length-1)
              ==> (items[x]<=items[y])}};

    public SortedSequence()
        ensures items == null;
        ensures length == 0;
    {}

    public void insert(int n)
        requires length < max;
        ensures n in items;
        ensures forall {int x; forall {int y;
                (0<=x && x<=y && y<=length-1)
                ==> (items[x]<=items[y])}};
        modifies items, length;
    {}
    ...
}
```