Translating Spec# Programs to C# with CodeContracts

Florian Egli

Master's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

FS 2011
April 23, 2012

Supervised by:
Valentin Wüstholz
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Contracts are used to specify the behavior of programs. These contracts can be verified statically, to prove that the program code is correct and the contracts are always satisfied, or they can be asserted during runtime. One programming language that offers support for contracts is Spec#. It implements an ownership model to ensure code correctness. Spec# tool support is limited to sscBoogie for static verification.

CodeContracts is a project that aims to offer contracts for standard .Net, with a different specification language than Spec# and extended tool support.

The goal of this thesis is to design an encoding for Spec# programs in C# and CodeContracts. With this, we are able to translate programs written in Spec# to CodeContracts compatible code, extended with ownership states and assertions that would be checked by sscBoogie, while preserving the programs semantic.

For the translation, we use an extension of CodeContracts, that provides additional specification constructs, called CodeContracts++.

These translated programs can then be analyzed and tested with tools that support CodeContracts, such as Pex for automatic test generation and Clousot for code analysis, that otherwise would not be available for Spec#.

Acknowledgments

I would like to thank Prof. Dr. Peter Müller for the opportunity to do my master's thesis in such an interesting research area. I would also like to thank my supervisor, Valentin Wüstholz, for his great support during my thesis and for the numerous hours of discussion about subtle details in the Boogie encoding and the Spec# compiler architecture.

Contents

Chapter 1

Introduction

This chapter gives an overview of this thesis and the motivation behind it. In the later sections, all technologies involved in this thesis are briefly explained, to ensure a certain background.

1.1 Motivation

The goal of every programmer is to write correct code. Achieving this is not easy. To help the programmer in finding bugs in the code, static verification and code analysis can be used. It is desired that program verification runs automatically, without user interaction. Spec# [29] and CodeContracts (CC) [7], both languages or language extensions provide such automatic verification and analysis tools. But both use different approaches to analyze the code and to find bugs. Spec# uses Boogie [5] with verification conditions to verify the code, whereas CodeContracts uses Clousot [12] which is based on abstract interpretation. Both methods try to ensure that the program code meets the programs specification. Abstract interpretation is more efficient but less precise than verification conditions. Both approaches have their benefits and disadvantages.

A counterexample that violates the code specifications is one of the most understandable ways for tools to point out bugs to a programmer. Pex is a tool, that generates test cases for CodeContracts code and searches such counterexamples. For Spec# there has no such tool been developed so far.

The goal of this thesis is to translate Spec# code (and the necessary assertions for ensuring program correctness) into CodeContracts code, so that Spec# users can benefit from the growing number of tools that exist for CodeContracts.

The translation from Spec# to CodeContracts is not trivial. Spec# implements an ownership model that allows to statically check object invariants and heap modifications. This model must be translated into CodeContracts.

Spec# generates .Net compatible Intermediate Language (IL) code. Runtime checks are only generated for method pre-/postconditions and object invariants. Non-nullable types are checked by the Spec# type system during compilation. All other conditions, such as ownership state, frame conditions etc., are checked by Boogie.

All assertions Boogie creates, will be added to the IL code, using CodeContracts++ (CC++). The benefit of this approach is, that all tools that support IL can process and verify this translated/extended code, and not only tools that support Boogie.

Our translation will use the dynamic frames theory to specify heap modifications. Therefore, the ownership model and heap model handling must be slightly adapted from Boogie.

1.2 Assumptions

In this thesis we focus on some core features of Spec#. It is not the goal of this thesis to provide a full translation of all language features offered by Spec#.

This thesis was done under some assumptions: Visibility and information hiding is not an issue in this thesis. All class members (that are added during the translation) are public. It is assumed that these members are not accessed by the user code, only by the generated validation code. Certain language features, such as generics, delegates and static fields are not covered in this thesis and are left for future work. Checking type safety of the Spec# programs was not taken into account, as type safety is checked by the Spec# type system, and not by Boogie.

1.3 Outline

This thesis aims to translate Spec# programs to CC++ programs. Chapter 2 gives an insight in the Spec# ownership model and defines the translation rules, explained on a concrete example. Chapter 3 explains the architecture of the modified Spec# compiler backend and how the translation rules have been implemented. Chapter 4 evaluates the implemented encoding using Pex and Clousot. Chapter 5 concludes this work.

1.4 Technologies

This section gives a brief overview of the different technologies involved in this thesis and how they are connected. It is not intended to give full background information that is needed to understand the issues discussed in this thesis. For further information, the referenced papers can be consulted.

1.4.1 Spec# and Boogie

Spec# is a research language, developed by Microsoft. It is a superset of C#, .Net 2.0, with additional support for contracts. Some notable language features are: Object invariants, pre-/postconditions, loop invariants, modifies clauses etc.

Spec# implements an ownership model. Every object has a specific state within this model. The ownership states determine if objects are allowed to be modified at a certain point in the program.

ssc is the Spec# compiler. It generates annotated IL code. Contracts are compiled into serialized abstract syntax tree (AST) strings and added as method attributes, or as string parameter of marker methods from the *Microsoft.Contract* namespace. sscBoogie takes this annotated IL code and generates Boogie code (BPL)[1]. This Boogie code can then be statically verified.

Boogie takes Boogie code and checks the consistency between a program and its specification by generating verification conditions, that are checked with the Z3 Satisfiability Modulo Theories (SMT) solver [31].

Mike Barnett and Manuel Fähndrich have plans to modify the current implementation of Spec# and integrate CodeContracts in it [1]. So that Spec# uses the CodeContracts infrastructure instead of the current custom infrastructure. With this, some of our translation steps would become obsolete and we could focus on extending the generated CodeContracts checks with the ownership model. Currently, this is not the case.

1.4.2 CodeContracts

CodeContracts was developed by Microsoft. It is based on the experience gained by the Spec# project. CodeContracts provides support for pre-/postconditions, object invariants, assertions and assumptions, similar to Spec#. It does not contain an ownership model or provide support for non-null types or loop invariants.

The CodeContracts specifications consist of empty method declarations (markers) that are not executable. These specifications are turned into runtime checks by the CodeContracts rewriter (ccrewrite).

[1]sscBoogie can only process code that was compiled with the ssc parameter /debug

Pex Pex is a code analysis tool that automatically finds test cases to achieve high code coverage, using dynamic symbolic execution [30]. It works on the rewritten CodeContracts code.

Clousot Clousot is a static contract checker, using abstract interpretation [12, 11]. It works directly on the CodeContracts declarations.

1.4.3 CodeContracts++

CodeContracts++, or CC++, was developed by Maria Christakis, Peter Müller and Valentin Wüstholz. It is an (independent) extension of CodeContracts and provides additional framing features, such as writes-, reads- and fresh-clauses and ghost variables. CC++'s language extensions are partially based on Dafny.

In our thesis, this library acts as an intermediate language between Spec# and CodeContracts. It is used to model the Spec# ownership methodology using dynamic frames.

1.4.4 Dafny

Dafny provides similar features as Spec#, but uses dynamic frames to specify heap modifications [17, 8]. Code verification is also done with Boogie. Other than Spec#, Dafny does not support inheritance.

To have a certain confidence that our translation rules are correct, a subset of the rules have been prototyped and tested in Dafny. The results from these tests have been integrated in the CC++ translation. In particular, a simplified version of the ownership model has proven to be useful to develop the encoding of our frame condition rules.

Chapter 2

Design

This chapter discusses the design of the translation for the Spec# ownership model and how Spec# programs are translated. Section 2.1 analyzes what kind of runtime checks are generated by Spec# and which runtime checks have to be added during the CC++ translation. Section 2.2 describes the encoding of the ownership model in CC++. Section 2.3 explains the translation rules for the Spec# key language features.

2.1 Spec# Runtime Checks

The advantage of Spec# over C# is its native support for contracts, which allow to specify programs. Spec# allows interoperability with existing .Net code and turns the specified contracts into runtime checks [4]. While Spec# generates runtime checks for pre- and postconditions, frame conditions are not checked at runtime. They are only checked statically, by sscBoogie.

According to Mike Barnett and K. Rustan N. Leino, one reason that frame conditions and ownership states are not checked at runtime by Spec# is, that this could cause programs to fail that otherwise would be correct. Another reason is that these checks can be very expensive and can cause a large runtime overhead, as they compare arbitrarily large portions of the heap. These checks are not in the IL code, but they are added by sscBoogie to the generated Boogie code, to be verified by Boogie. Beneficial would be to have these checks in the IL code, in order to verify them with other tools that work directly on the IL level. Thus, a goal of this thesis is, to convert all assertions that are generated for Boogie into runtime checks.

2.2 Ownership Model and Invariant Semantics

Spec# implements an ownership model, where objects are layered in a forest-shaped hierarchy (no cyclic structures). The invariant semantics of Spec# depends heavily on this ownership model. An object can own other objects and it can be owned by an object. An object that is owned by another object, is called a *representation object* (*rep*). Objects that are owned by the same object are called *peers*. These ownership relations are used to determine the state of objects and which objects are allowed to modify other objects. Depending on the state of an object, its invariants are known to hold or not.

Figure 2.1 shows a possible configuration of object states in the ownership model.

Committed objects are not allowed to be modified and their invariants must hold. Consistent objects can be modified as long as their object invariants are satisfied. Mutable objects are allowed to break their object invariants and can be freely modified. In short, object invariants must hold if the object is in a valid state and is not vulnerable to modifications.

The ownership states determine at what point in the program it is valid to update the fields of a class. Whereas fields can be read independent from the object state [2].

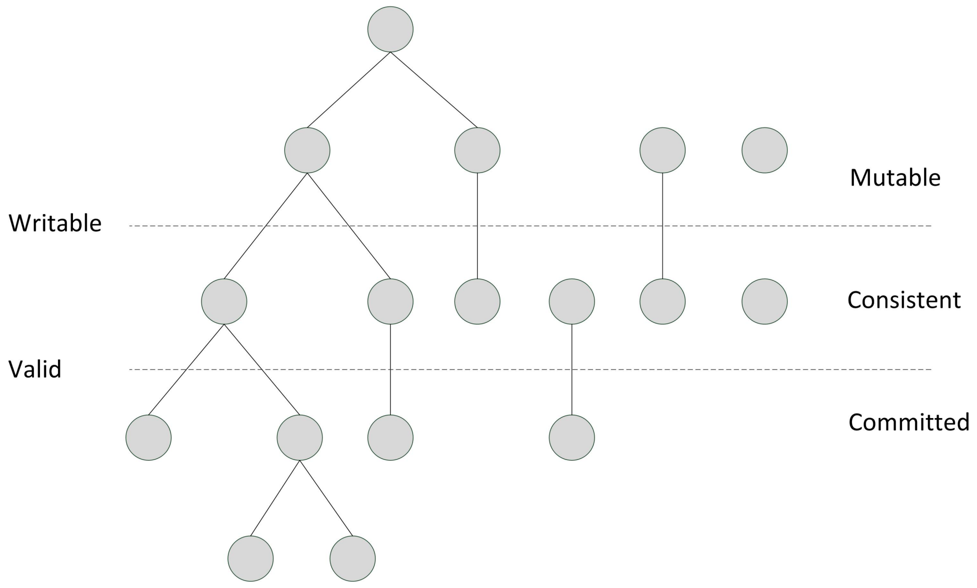

Figure 2.1: Ownership States Example

Field `f` of an object `o` may be modified within a method, if:

- `o.f` is specified (implicitly or explicitly) in the modifies clause of the method.
- object `o` was newly allocated within the method, which implies that `o` is in the fresh set of the method.
- object `o` is committed and (transitively) owned by an object that is allowed to be modified within the method, which implies that `o` is an element of the rep set of an object that is specified in the modifies clause.

The state of an object depends on its own invariant state and the state of its owner. Table 2.1 shows all possible ownership states.

State	**Description**
Committed	The object is not allowed to be modified. The object invariants of all class frames hold.
Consistent	The object is allowed to be modified as long as the object invariants are satisfied. The object is unowned, or the owner is exposed.
Mutable	The exposed class frame is allowed to be freely modified. The object invariants of this class frame can be violated. Directly owned objects can be modified, as they are in the state *Consistent*.
Valid	The object's invariants hold.
Writable	The object is allowed to be modified.

Table 2.1: Ownership States

The states *Valid* and *Writable* overlap with the states *Committed*, *Consistent* and *Mutable*. These two states are used to simplify the ownership rules.

Objects can switch between the states *Consistent* and *Mutable* with *Unpack* and *Pack*. For an object to be in the *Committed* state, it has to have an owner (we say it must be owned) and the owner must be in the state *Consistent* or *Committed*. The states *Consistent*, *Committed* and

Valid apply to an object as a whole. The states *Mutable* and *Writable* apply to a specific class frame. A class frame is a specific type in the inheritance hierarchy of an object.

Figure 2.2 shows the state transitions of an object from *Committed* to *Mutable*. First, to change from `Committed` to `Consistent`, the owner of this object is unpacked, then, to change from `Consistent` to `Mutable`, the object itself is unpacked.

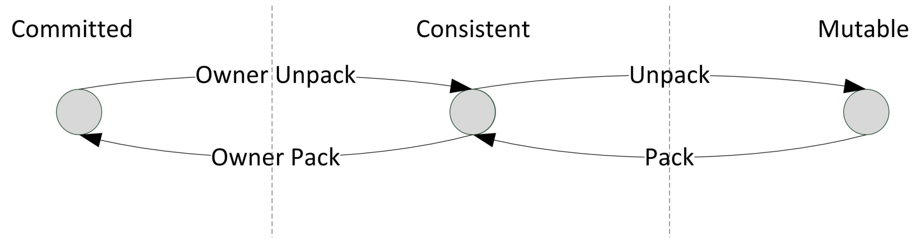

Figure 2.2: Possible State Transitions of an Object

Due to inheritance, an object can consist of multiple types (or class frames) depending on the inheritance structure. Each class frame can have its own object invariants. An ownership relation is always bound to an object reference and a class frame.

Within a method, the method can only modify objects it has a reference to. These are all owned objects, peer objects, objects passed as parameter and object fields (and all transitively reachable fields of these objects). Which of these objects a method is allowed to modify is defined by the frame condition (or modifies clause) of this method. The frame condition defines which objects can change in a method and which objects cannot.

During the execution of a method multiple sets have to be maintained, which store the state of the current execution and which are used for checks; e.g., prestate, poststate, modifiable objects, modifiable fields, fresh.

The described ownership model is called the standard ownership model. sscBoogie supports two additional ownership models, the trivial ownership model and the experimental ownership model. These models are not covered by this thesis.

2.2.1 State Maintenance

To model ownership, the following ghost variable fields are required: `_inv`, `_localinv`, `_owner`, `_ownerframe`, `_rep`. All these fields are defined per object instance. A detailed description of some of these fields can be found here [2, 22].

Ghost variables are variables that are only used for verification and can be removed from the final program without changing the programs behavior. CC++ offers support for ghost variables. These ghost variables are only used by checks generated by the CC++ translation. They are never accessed by the user code. In CC++, the ~ unary operator can be used as shortcut for the read operation on ghost variables; e.g., ~`_inv` returns the value stored in the `_inv` ghost variable. This operator will be used in the following chapters.

Table 2.2 shows how the ghost variable fields are related to the ownership states. Where `<:` denotes the reflexive and transitive subtype operator (e.g., `string <: object`), and `o` denotes an object. `frame` denotes a class frame between the allocated type of object `o` and the root base class. The `Type` property returns the allocated type of an object. The `BaseType` property returns the direct supertype of a type.

Inv

The field `_inv` is of type *Type* and it represents the most derived class frame of an object which is valid; e.g., if `_inv` is set to type `A`, and `A` is a subtype of base type `Object`, then the object invariants of all class frames between `Object` and, including `A` must hold.

State	Mapping
$fullyvalid(o)$	$o.\_inv = o.Type \wedge o.\_localinv = o.Type$
$writable(o)$	$\neg owned(o) \vee mutable(o.\_owner, o.\_ownerframe)$
$consistent(o)$	$fullyvalid(o) \wedge writable(o)$
$committed(o)$	$fullyvalid(o) \wedge owned(o) \wedge fullyvalid(o.\_owner)$
$valid(o)$	$consistent(o) \vee committed(o)$
$mutable(o, frame)$	$localMutable(o, frame) \vee additiveMutable(o, frame)$
$localMutable(o, frame)$	$\neg(obj.\_localinv = frame.BaseType)$
$additiveMutable(o, frame)$	$\neg(o.\_inv <: frame)$
$partiallyConsistent(o, frame)$	$\neg mutable(o, frame)$
$owned(o)$	$\neg(o.\_owner = null)$

Table 2.2: Ownership Mapping

The field `_inv` can only be changed by the commands *additive unpack* and *additive pack*.

Local Inv

The field `_localinv` is of type *Type* and represents that all class frames are valid, except the class frame directly below the class frame at which `_localinv` points at; e.g., in an inheritance structure `SpecialDictionary <: Dictionary <: CCObject`, and `_localinv` points to `CCObject`, then the class frames `SpecialDictionary` and `CCObject` are valid, but `Dictionary` is not.

The field `_localinv` can only be changed by the commands *unpack* and *pack*.

Owner

The field `_owner` is of type object reference and points to the owner object.

The field `_owner` can only be changed once in an object's lifetime, when the object becomes owned by another object. Before the object is owned, the field is null.

Owner Frame

The field `_ownerframe` is of type Type and states the class frame of the owner that owns the object.

The field `_ownerframe` can only be changed once in an object's lifetime, when the object becomes owned by another object. Before the object is owned, a peer group placeholder is assigned to this field.

Rep

The `_rep` field, also known as the rep set/region (*rep*), contains all objects that are transitively owned by an object instance (including the object itself).

The `_rep` set is changed if the current object is set as owner of another object, or at the end of an expose statement when the `_rep` sets of transitively owned objects are updated.

The big difference to the Boogie encoding is, that not only directly owned objects are in the rep set, but all transitively owned objects. Thus, the `_rep` set can be used to check the fresh- and modifies condition.

The following properties hold:

- If an object is valid, then the rep set of every transitively owned object of this object is a subset of this object's rep.
- The object itself is an element of its rep set.

- An object's owner can never be an element of the object's rep set. (This invariant disallows cyclic ownership.)
- If objects in a rep set have the same owner, then the rep sets of these objects are distinct. $(\forall a \in \mathit{this.\_rep},\ b \in \mathit{this.\_rep}\ :\ \mathit{a.\_owner} = \mathit{b.\_owner} \Rightarrow a = b \vee \mathit{a.\_rep} \cap \mathit{b.\_rep} = \emptyset)$
- The owner of every object in a rep set must be an element of rep set itself (except the owner of the object whose rep set it is: $\mathit{this.\_owner} \notin \mathit{this.\_rep}$).

2.2.2 Peer Group Placeholder

Peer group placeholders (PGP) are placeholders for missing owners. If an object is unowned, it has a PGP as owner. Every newly allocated object has its own PGP owner and is the only element in the PGP's rep set. This PGP owner is used, if multiple objects become peers even if they are unowned.

The object that becomes a peer of another object is added to the rep set of the new peers peer group placeholder and changes the owner reference to this peer group placeholder. If the object that becomes peer already has peers, then all peers are updated and the two peer group placeholder rep sets are merged.

Figure 2.3 shows how two rep sets are merged, if an object in one peer group is assigned as peer of an object in another peer group. Object `c` has a peer field `f`. If object `d` is assigned to this field, all objects in the peer group `B` are merged into `A`'s rep set and become part of peer group `A`.

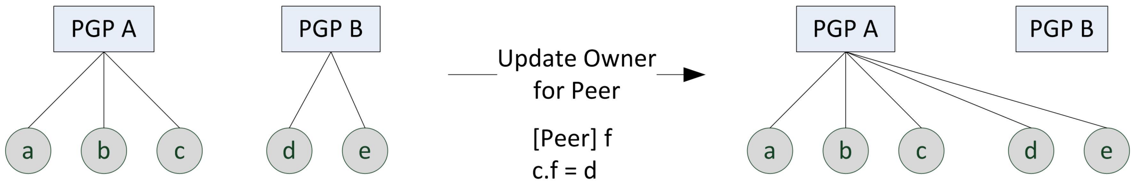

Figure 2.3: Update Owner for Peer

The peer group placeholders are modeled exactly the same as in Boogie.

2.2.3 Ownership Assignment

An object that becomes the new owner of an other object, has to be at least as modifiable as the object itself. Otherwise, this would violate the ownership state rules. Object can only become owned by writable objects.

Table 2.3 shows in what state an object is allowed to become owned and the resulting state; e.g., after the statement `UpdateOwnerForRep(object, owner)`, if the two objects were in the correct state, `object` is owned by `owner` and changed its state.

In the current version of Spec#, the ownership relation can only be assigned once. If an object is owned, it can not be unowned or owned by another object. Therefore, we also prevent this in our model.

2.2.4 Dynamic Frames

The Spec# verifier reasons over the entire heap. This is not possible in CodeContracts, as CodeContracts allows reasoning only over a bounded number of objects. Therefore we must define a method in which we can reason over all objects that can be modified without knowing the states of all objects on the heap. For this we use dynamic frames [16], where all required states and objects are maintained in sets, modeled with ghost variables.

In dynamic frames, every method has a footprint. This footprint states which already allocated objects are allowed to be modified in a method. In addition, we want to reason over newly allocated

		owner		
		mutable	**consistent**	**committed**
object	**mutable**	mutable	Error: RHS and its peers must all be valid	Error: Target object of assignment is not allowed to be committed
	consistent	consistent	committed	Error: Target object of assignment is not allowed to be committed
	committed	Error: RHS may already have a different owner	Error: RHS may already have a different owner	Error: RHS may already have a different owner

Table 2.3: Ownership Mapping

objects, and to determine which objects are fresh at the end of a method. For pure methods, the footprint is defined as the objects that are allowed to be read in the method [15].

We will justify the use of dynamic frames and how dynamic frames can be used to model the Spec# ownership methodology in the following sections.

2.2.5 Method Contracts

Method contracts consist of pre- and postconditions. In Spec#, they are specified with the keywords *requires* and *ensures*. Contracts are required to be side-effect free. sscBoogie translates these Spec# conditions to Boogie conditions.

In Boogie, there is a distinction between *free* and *checked* conditions [18]. Boogie also contains the notion of free invariant and free modifies, but they are not relevant in our context. Checked conditions are assumed by one party (caller or callee) and asserted by the other. Free conditions are assumed by one party, and ignored by the other. The callee assumes all preconditions (free and checked) of a method. The caller of a method is responsible for establishing the checked preconditions and must establish the checked modifies clauses, as can be seen in Table 2.4. At the end of a method, the caller can assume all postconditions (free and checked). The callee has to establish the checked postconditions. Free postconditions have already been established in the method itself and are not checked again on exit by the callee.

	Precondition		**Postcondition**	
	free	**checked**	**free**	**checked**
caller	ignore	assert	assume	assume
callee	assume	assume	ignore	assert

Table 2.4: Difference between Free and Checked Contracts

Boogie creates several default (free and checked) pre- and postconditions per method, additional to the user specified conditions, which establish the ownership model. The preconditions are checked, whereas the postconditions are free. Boogie creates separate checks for consistency and peer consistency, where peer consistency does not imply consistency.

Checked Boogie default preconditions:

- Target object is consistent
- Target object is peer consistent
- Parameters are consistent
- Parameters are peer consistent

Free Boogie default postconditions:

- Newly allocated objects are fully valid
- First consistent owner is unchanged if its `exposeVersion` is
- Frame condition
- Inv/localinv change only in blocks (for all already allocated objects, at the end of the method, they are the same as at the beginning of the method)

The difference between free and checked conditions results from our translation encoding; e.g. fresh and modifies clauses are translated as free conditions, as they are checked directly at the affected program points, and not only at the end a method.

All postconditions are checked explicitly with runtime checks.

2.2.6 Consistency Check

At the beginning and at the end of every method, the object whose method is called must be consistent. Consistent means, that the object is fully valid and its owner is writable. Additionally, the object invariants of all class frames of this object must hold.

The consistency and peer consistency preconditions in CC++ are nearly identical to the preconditions created for Boogie.

Boogie ensures that at the end of a method all newly created objects are fully valid. In our encoding, we ensure this by checking that at the end of the method, all objects that could have been modified (and that could transitively own newly allocated objects) are consistent. This check includes, that if an object is consistent, all transitively owned objects must be committed, and therefore are fully valid. Furthermore, after object construction, the object is consistent (which implies fully valid), and with the constraint that `_inv/_localinv` is only allowed to change in blocks, the object must be fully valid at the end of the method, independent of the methods called on this object.

`_inv` and `_localinv` are only allowed to change in blocks, this means that every unpack must be followed by a pack, within the same method. This is ensured by Spec#'s encoding, as every *expose* keyword is replaced by an unpack and a pack statement.

The Boogie variable `exposeVersion` (also called snapshot) is declared per object and is used to indicate if the object was modified. If the `exposeVersion` at the end of a method is the same as at the beginning of the method, then the object and all its transitively owned objects were not modified. In our encoding, we replace checks that involve `exposeVersion` with checks over the modifies rep set. So, `exposeVersion` is not longer required in our encoding.

2.2.7 Frame Condition

The frame condition depends on the modifies clause of the current method. Boogies frame condition ensures, that at the end of a method only objects that have been specified in the modifies clause have been modified. This is ensured in our encoding by frame condition checks before every field update and method call, as done in sscBoogie.

Well-formedness

sscBoogie does not check the well-formedness of modifies clauses. Variables that are stated in modifies clauses are allowed to be null at evaluation time. To ensure the same behavior in our model, non-null checks are added which avoid dereferencing null values to prevent `NullReferenceExceptions`.

2.2.8 Ghost Variable Encoding

Spec# uses some ghost variables per object for managing ownership states.

The current version of our model requires all objects to inherit from a custom object class. Figure 2.4 shows the properties and methods of this custom object class. The custom object class acts as base class and contains ghost variables for managing the ownership states, as well as methods that must be overridden by all derived types.

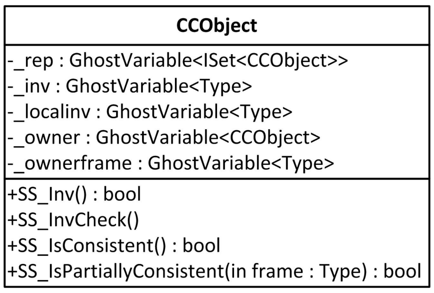
CCObject
-_rep : GhostVariable<ISet<CCObject>>
-_inv : GhostVariable<Type>
-_localinv : GhostVariable<Type>
-_owner : GhostVariable<CCObject>
-_ownerframe : GhostVariable<Type>
+SS_Inv() : bool
+SS_InvCheck()
+SS_IsConsistent() : bool
+SS_IsPartiallyConsistent(in frame : Type) : bool

Figure 2.4: Custom Object Class

The reason for a separate base class is, that the `System.Object` base class can not be extended.

The `CCObject` custom object class is part of the `CodeContractsForSpecSharp` library. The library is explained in section 3.1.3.

Custom Object Class Invariants

The custom object class contains the following object invariants:

- The ghost variable fields (`_inv`, `_localinv`, `_rep`, `_owner`, `_ownerframe`) are never null
- The value of `_owner` is null and the value of `_ownerframe` is a peer group place holder, or the value of `_owner` is not null and `_ownerframe` is not a peer group placeholder.
- For all transitively owned objects (rep set without the current object) it must be, that:
  - The object's owner is not null.
  - The object's ownerframe is not null.
  - The object's owner is in the rep set of the current object.
  - The object's owner is not in the rep set of the object itself.

In Spec#, the `System.Object` frame can not be exposed. In our methodology we prevent the custom object class frame from being exposed as well. A consequence of this is, that the object invariants of the custom object class frame must always hold. We moved these invariants to a separate method `[Pure] public static bool CCSS.Always()`, which is called by the method `SS_Inv` of the custom object class. This method is also called in other places, as we will see in the following sections.

SS_Inv Method

The non-virtual method `[Pure] public bool SS_Inv()` is overridden by each class frame and contains the object invariants of the specific class frame. The method is public because of visibility requirements [22]. The object invariants must be visible to every object that can potentially violate it.

There is an essential difference between Spec# object invariants and CodeContracts object invariants. Spec# object invariants have to hold always when an object is valid. This is always when an object is not exposed. On the other hand, CodeContracts object invariants have to hold only at the end of the object's methods, and they are not checked at all if the fields of this object are modified outside of the object. Therefore additional object invariant checks have to be added during the translation.

SS_InvCheck

The non-virtual method `private void SS_InvCheck()` is overridden by each class frame and is decorated with the `System.Diagnostics.Contracts.ContractInvariantMethodAttribute` attribute. This method is used to tell CC++ what the object invariants are. This method contains a call to the `SS_Inv` method of the current class frame. During rewriting with ccrewrite, object invariant checks are added at the end of every method and constructor. This would not be necessary, the object invariants are already checked in the consistency check. The primary use of this object invariant check method is for other analysis tools that do not depend on rewriting, and that have to know the object invariants.

The ccrewriter converts the methods decorated with `[ContractInvariantMethod]` into a method `private void $InvariantMethod$()`, that is called at the end of every method, property and constructor (except those decorated with `[Pure]`).

SS_IsConsistent Method

The virtual method `[Pure] public bool SS_IsConsistent()` is overridden by each class frame. This method checks the object invariants of the specific class frame, by calling the `SS_Inv` method, and then calls the overridden `SS_IsConsistent` method of the base class. This ensures, that once this method is called, all object invariants of all class frames, from the most lower subclass up to the custom object class, are checked.

The `SS_IsConsistent` method of the custom object class additionally ensures, that the object is fully valid (`_inv` and `_localinv` point to the allocated type of the object), that the object is writable (the object is unowned or its owner is mutable) and all owned objects are committed.

SS_IsPartiallyConsistent

The non-virtual method `[Pure] public bool SS_IsPartiallyConsistent(Type)` is overridden by each class frame. This method checks the object invariants of the specify class frame, by calling the `SS_Inv` method, and then calls the overridden `SS_IsPartiallyConsistent` method of the base class, up to the custom object class. After calling this method, one can be sure that the object invariants of all class frames between the current frame and the root frame hold.

This method takes a type as parameter, which must be a supertype of the allocated object type. The `SS_IsPartiallyConsistent` method of the custom object class ensures that the object is valid up to the specified type and that the object is writable.

2.2.9 Owner Rep

Some fresh and modifies clauses reason over the rep set of the owner of an object. For example, if the reasoning involves peers of an object. Checks for these clauses normally involve consistent or committed objects, for which the rep sets are up to date and include all transitively owned objects, as will be seen in section 2.3.7.

If reasoning over the owner rep set of an object is required, it is most certain that the owner is in the state mutable. Thus, we can not rely on the rep set of the owner to be up to date and to include all transitively owned objects. The owner object can be in a state as shown in Figure 2.7, before the rep update, in which the rep sets of directly owned objects are not subsets of the owner's rep set. But, as the object itself must be at least consistent, all peers of this object are consistent (with up to date rep set) as well.

In the following, when we reason over the rep set of the owner of an object (e.g., `o._owner._rep`), we mean the union of all rep sets of all directly owned objects of this owner. Thus, we can be sure to reason over all objects, even if the owner is in the state mutable. In our encoding, the method `GetAllPeerAndRepObjects`, in the assembly `CCSS`, returns the described set.

Listing 2.1 shows why we can not directly take the owner's rep to reason over fresh objects. Within a method, the current object is consistent and its owner is mutable. After assigning a new object to a `[Rep]` field `f` of a `[Peer]` field `peer` of `this`, the newly allocated object is in the rep set of the peer object, but it is not in the rep set of the common owner of `this` and `peer` as the owner's rep will be updated when it changes its state to consistent (See section 2.3.7). But we will get the new object if we reason over the rep sets of the peers directly.

Listing 2.1: Spec#: Owner Rep Example

```
  this.peer.f = new CCObject();
  this.peer.f.UpdateOwnersForRep(this.peer, typeof(RepExample));
  // this.peer.f is not in the rep set of this.owner
```

The method `UpdateOwnersForRep` is used for `[Rep]` fields and updates the owner and the owner frame of the specified object. In our example, the object assigned to field `f` will be owned by the object assigned to field `peer` at the class frame `RepExample`.

2.3 Translation Rules

The checks added during the translation are similar, or identical, to the Boogie checks generated by sscBoogie. But instead of reasoning over the entire heap as done in Boogie, our checks reason over bounded sets. For every assert in the generated Boogie code, there is a corresponding check in the CC++ code, adapted to our ownership model encoding.

2.3.1 Dictionary Example

We define the translation rules, using the Dictionary example, taken from the Spec# tutorial [23]. Listing 2.2 shows the Spec# code of a slightly adapted version, which contains all Spec# language features supported by the translation.

Listing 2.2: Spec#: Dictionary Example

```
public class Dictionary {
  [Rep] Node? head = null;
  [Rep] Node? tail = null;
  invariant head == null <==> tail == null;

  public static void InsertTest(Dictionary! d)
    modifies d.*;
  {
    d.Insert("foo", 123);
  }

  public void Insert(string key, int val)
        modifies head, tail;
  {
    expose (this) {
      Node? h = head;
      head = new Node(key, val);
      head.next = h;
      if (tail == null) {
        tail = head;
      }
    }
  }
}

internal class Node {
  public string key;
  public int val;
  [Peer] public Node? next;
  public Node(string key, int val) {
    this.key = key;
    this.val = val;
  }
}
```

Figure 2.5 shows the class diagram of the Dictionary example. The class Dictionary contains two methods, for inserting new values and finding already inserted values. The method `Find` will be skipped in the following, because it contains no concepts that are not already used in the `Insert` method.

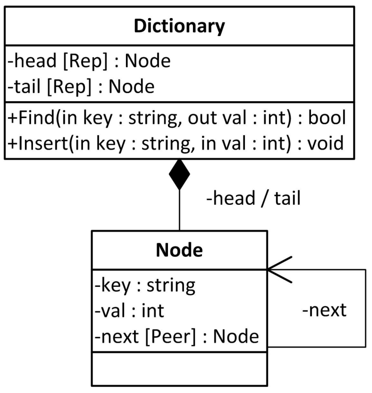

Figure 2.5: Dictionary Example Class Diagram

For every value that is inserted, a new Node is created. Nodes are linked together similar to a linked list. The Dictionary has a reference to the first and the last node in this linked list. These references are declared as `[Rep]`. Nodes are `[Peer]` to each other. Therefore, all Nodes are owned by the Dictionary object.

Dictionary Instance

Figure 2.6 shows the ghost variables of an instance of the `Dictionary` class, after inserting two values and packing the `TestDriver` object. The `Dictionary` object `d` is owned by a `TestDriver` object `t`. The `Dictionary` object and both `Node` objects are committed. The `TestDriver` object has no owner but is fully valid, so it is in the state consistent. As all objects are valid, all rep sets are strict supersets of their owned object's rep sets; e.g., the object `t`'s rep set contains the transitively owned objects `d`, `n1` and `n2`, and `t` itself.

2.3.2 Initialization / Construction

Constructors start as mutable. At the end of every constructor, the constructed object must be partially consistent up to the current class frame. The constructor could be in the middle of an initialization hierarchy, therefore it can not be ensured that the object is fully valid. But it can be ensured that all constructors that are higher in the inheritance hierarchy have already been executed and therefore the object is consistent up to (and including) the current constructor.

At the end of a constructor there is an additive pack [14] to assert partial consistency.

Listing 2.3 shows the code that is generated for the Dictionary constructor. At the end of the constructor, the object is partially consistent on class frame `Dictionary` and the object invariants of class frame `Dictionary` hold. If a caller creates a new instance of type `Dictionary`, after the constructor, it can imply that partial consistency implies consistency, as `Dictionary` is the lowest class frame in the inheritance hierarchy.

Listing 2.3: CC++: Constructor

```
public Dictionary()
{
  Contract.Ensures(this.SS_IsPartiallyConsistent(typeof(Dictionary)));
  // Initialization & base constructor call...
  Contract.Assert(this.SS_Inv());
  SSC.AdditivePack(this, typeof(Dictionary));
}
```

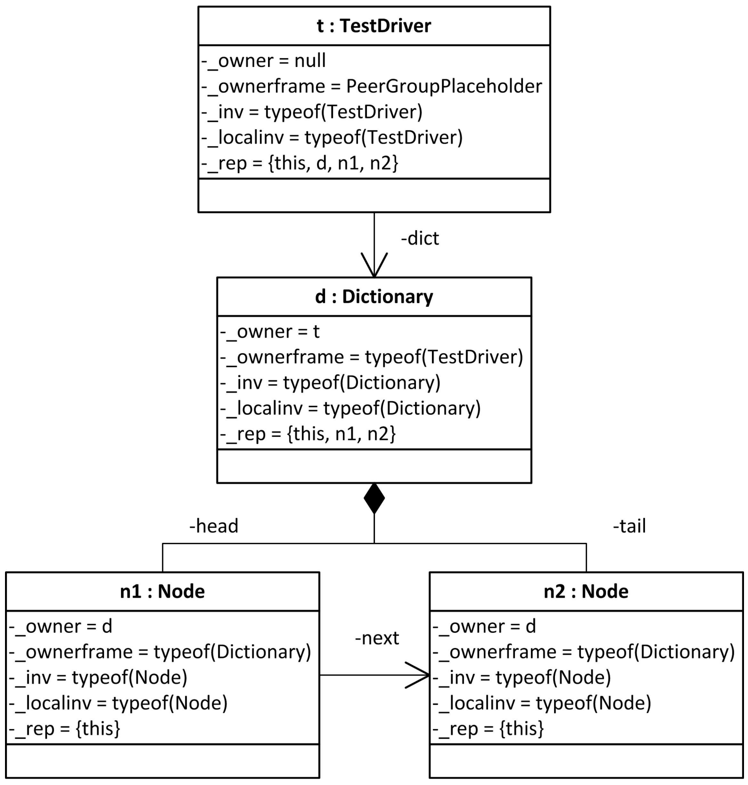
t : TestDriver
-_owner = null
-_ownerframe = PeerGroupPlaceholder
-_inv = typeof(TestDriver)
-_localinv = typeof(TestDriver)
-_rep = {this, d, n1, n2}
-dict
d : Dictionary
-_owner = t
-_ownerframe = typeof(TestDriver)
-_inv = typeof(Dictionary)
-_localinv = typeof(Dictionary)
-_rep = {this, n1, n2}
-head
-tail
n1 : Node
-_owner = d
-_ownerframe = typeof(Dictionary)
-_inv = typeof(Node)
-_localinv = typeof(Node)
-_rep = {this}
-next
n2 : Node
-_owner = d
-_ownerframe = typeof(Dictionary)
-_inv = typeof(Node)
-_localinv = typeof(Node)
-_rep = {this}

Figure 2.6: Dictionary Example Instance Diagram

2.3.3 Object Invariants

Spec# objects can specify object invariants with the keyword *invariant*. Boogie encodes these object invariants with an axiom: The class frame is exposed, or the object invariants of this class frame must hold. CC++ ensures the object invariants with pre- and postconditions and with additional assertions within the methods. Object invariants of valid objects must always hold.

In ownership-based object invariants, only fields of transitively owned objects of an object (including this) are allowed to be used in the object invariants of this object.

Every class frame is extended with a method `[Pure] public new bool SS_Inv()`. This method contains all object invariants defined in the current class frame. In addition to the user defined object invariants, the method `CCSS.Always` is called. The reason for this is explained in section 2.2.8.

Listing 2.4 shows the object invariants of the `Dictionary` class. The field `head` is null if and only if field `tail` is null.

Listing 2.4: CC++: Dictionary Object Invariants

```
[ContractInvariantMethod]
private new void SS_InvCheck()
{
  Contract.Invariant(this.SS_Inv());
}

[Pure]
public new bool SS_Inv()
{
  return (this.Always() && ((this.head == null) <==> (this.tail == null))
    && CCSS.NullOrOwnerIs(this.head, this, typeof(Dictionary))
    && CCSS.NullOrOwnerIs(this.tail, this, typeof(Dictionary)));
}
```

Non-Nullable Fields

In Spec#, non-nullable fields are specified with the suffix `!`, nullable types with the suffix `?`; e.g., `string! nonnullable; string? nullable;`. Non-nullable types are checked by the Spec# type system, which outputs a warning for type violations, but are not checked again by Boogie. If non-nullability would be declared as object invariant (e.g., `invariant nonnullable != null;`), this would be checked by Boogie.

For every non-nullable field of a class frame, an additional non null check is added to the class frames object invariants during translation; e.g., `field != null`. This corresponds to the behavior of a non-null check via object invariants in Spec#, but differs slightly from the behavior of non-nullable types, as non-null fields of exposed objects are only checked before packing the object and not after every assignment.

Rep Fields

For all `[Rep]` fields in a class frame, the object invariants of this class frame are extended with an owner check to assert that the current object is the direct owner if the object that is assigned to this rep field.

Listing 2.4 shows that the object invariants of class `Dictionary` are extended with owner checks for the `[Rep]` fields `head` and `tail`, which assert that `this` on the class frame `Dictionary` is the owner of these two fields.

Peer Fields

For all `[Peer]` fields of a class frame, the object invariants of this class frame are extended with an owner check to assert that the current object has the same owner as the object that is assigned to this peer field.

Listing 2.5 shows that the object invariants of the class `Node` are extended with an owner check for the peer field `next`, which asserts that `this` and `next` are owned by the same object and the same class frame.

Listing 2.5: CC++: Node Object Invariants

```
[ContractInvariantMethod]
private new void SS_InvCheck()
{
  Contract.Invariant(this.SS_Inv());
}

[Pure]
public new bool SS_Inv()
{
  return (this.Always() && CCSS.NullOrOwnerSame(this.next, this));
}
```

2.3.4 Pre- and Postconditions

Spec# methods can specify pre- and postconditions. User defined preconditions are translated into `Contract.Requires` method calls. User defined postconditions are translated into `Contract.Ensures` method calls.

In Spec#, objects mentioned in the precondition must be visible to all callers. Objects, mentioned in the postcondition must be visible to all implementations of this method (also to all methods that override the current implementation in subclasses). In CodeContracts, all objects mentioned in a contract, must be at least as visible as the method in which they appear [6].

Ownership- and Invariant-State Checks

Before an instance method call, the object on which the method is called is required to be consistent and peer consistent. At the end of the method, the object again has to be consistent and peer consistent. The consistency check is explained in section 2.2.8. The peer consistency check ensures that all peer objects are consistent as well.

Listing 2.6 shows the state-check contracts of the instance method `Insert`.

Listing 2.6: CC++: State Check Pre-/Postconditions

```
public void Insert(string key, int val)
{
  ...
  Contract.Requires(this.SS_IsConsistent());
  Contract.Requires(this.IsPeerConsistent());
  Contract.Ensures(this.SS_IsConsistent());
  Contract.Ensures(this.IsPeerConsistent());
  ...
}
```

Static Methods

Static methods have no default state-check pre- and postconditions (as static methods have no *this* instance reference they can refer to). They have no constraints on consistency or peer-consistency. Fresh and modifies clauses are checked for static methods.

Parameters

Per default, parameters of type reference are required to be consistent and peer consistent at the beginning of and at the end of a method, due to the requirement that `_inv`/`_localinv` only changes in blocks. Value type parameters have no special constraints. Depending on whether the parameter is of type nullable or non-nullable, there is a non-null check.

Listing 2.7 shows the different contracts for a nullable parameter `a` and a non-nullable parameter `b`. Parameter `a` is allowed to be null, whereas it is required that parameter `b` is not null.

Listing 2.7: CC++: Parameter State Check Pre-/Postconditions

```
public void ParameterTest(CCObject/*?*/ a, CCObject/*!*/ b)
{
  ...
  Contract.Requires(a == null || a.SS_IsConsistent());
  Contract.Requires(a == null || a.IsPeerConsistent());
  Contract.Requires(b != null && b.SS_IsConsistent());
  Contract.Requires(b != null && b.IsPeerConsistent());
  Contract.Ensures(a == null || a.SS_IsConsistent());
  Contract.Ensures(a == null || a.IsPeerConsistent());
  Contract.Ensures(b != null && b.SS_IsConsistent());
  Contract.Ensures(b != null && b.IsPeerConsistent());
  ...
}
```

The ccrewriter automatically replaces parameter references in `Contract.Ensures` statements with `Contract.OldValue(<reference>)`.

Captured Parameters Captured parameters are allowed to change ownership within a method. They have an additional precondition that requires that the parameter is unowned. (E.g. `Contract.Requires(CCSS.OwnerNone(par));`) They have a weaker postcondition than normal parameters. They can be consistent or committed at the end of a method, because they are allowed to change ownership during the method.

Listing 2.8 shows the contracts for a captured non-nullable parameter `c`. `c` is required to be consistent, peer-consistent and unowned at the beginning of the method. At the end of the method it is ensures that `c` is valid (consistent or committed).

Out Parameters Out parameters have no default state check precondition, as out parameter are assigned within the method. They have default state check postconditions as normal parameters. CodeContracts requires that out parameters in contracts are surrounded with a `ValueAtReturn`

Listing 2.8: CC++: Captured Parameter

```
void CapturedParameterTest(/*[Captured]*/ C c)
{
  ...
  Contract.Requires(c != null && c.SS_IsConsistent());
  Contract.Requires(c != null && c.IsPeerConsistent());
  Contract.Requires(c != null && c.OwnerNone());
  Contract.Ensures(c != null && c.IsValid());
  ...
}
```

statement, because at the beginning of the method, where the contracts access the out variable, the out variable is still unassigned, which is not allowed in C#.

Listing 2.9 shows the contracts for a non-nullable out parameter `a`. The method does not require anything about the parameter, but ensures that the parameter is non-null, consistent and peer consistent at the end of the method.

Listing 2.9: CC++: Out Parameter

```
void OutParameterTest(out A a)
{
  ...
  Contract.Ensures((Contract.ValueAtReturn<A>(out a) != null)
    && Contract.ValueAtReturn<A>(out a).SS_IsConsistent());
  Contract.Ensures((Contract.ValueAtReturn<A>(out a) != null)
    && Contract.ValueAtReturn<A>(out a).IsPeerConsistent());
  ...
}
```

Return Values

If a method has a return value of type reference, the returned object is ensured to be consistent and peer consistent. For non-nullable return types, there is an additional non-null check.

Listing 2.10 shows the contracts for a non-nullable return value of type `A`.

Listing 2.10: CC++: Return Value

```
A/*!*/ ReturnValueTest()
{
  ...
  Contract.Ensures((Contract.Result<A>() != null) && Contract.Result<A>().SS_IsConsistent());
  Contract.Ensures((Contract.Result<A>() != null) && Contract.Result<A>().IsPeerConsistent());
  ...
}
```

Fresh Clause

The fresh clause checks that all objects that are mentioned in the fresh clause have been newly allocated during the method execution. The reasoning region for fresh objects is always a superset of the methods footprint (modifies region). Every method that has at least one modifies clause (default or user-defined) is extended with a CC++ fresh clause.

Listing 2.11 shows how Boogie handles fresh clauses in Dafny: For the expression `ensures fresh(_rep - old(_rep))`, at the end of a method in class `FreshTest`, it is ensured that all objects in the set `_rep` excluding the objects in the set `_rep` at the beginning of the method have not been allocated at the beginning of the method.

Listing 2.11: Boogie: Dafny Fresh Condition

```
  ensures (∀ $o: ref • $o ≠ null ∧ read($Heap, this, FreshTest._rep)[$Box($o)]
    ∧ ¬read(old($Heap), this, FreshTest._rep)[$Box($o)] ⟹ ¬read(old($Heap), $o, alloc));
```

In C# such an "is allocated" information is not available. To overcome this limitation, we store references of newly allocated objects at the point they are created in a fresh set.

In the Spec# methodology, a caller of a method does not care about objects that are allocated within this method and remain unowned. Therefore, we consider an object fresh, if it is allocated and owned within a method.

There are two cases in which an object can be fresh at the end of a method: The object was constructed in the method itself (with the keyword *new*), or the object was constructed in a method that was called by this method and is therefore in the fresh set of the called method.

New objects can be allocated and returned by a method, but they do not have to be tracked as fresh, as long as they are not owned. The fresh checks are independent from the specified modifies clause. Newly allocated objects can become owned by writable objects. Within a method, the only writable objects that are reachable, are the object's peers and the parameter's peers. Therefore, the fresh check includes the rep sets of all peers of this object, as described in section 2.2.9.

For the fresh check, the difference between the new rep set at the end of the method, and the original rep set at the beginning of the method is analyzed. The difference between these two sets must be a subset of all newly allocated objects during the method execution (including all called methods within this method). In CC++ fresh clauses are specified with the statement `ContractPP.Fresh`. Listing 2.12 shows a fresh condition for the modifies clause `modifies this.*` with a captured parameter `capturedParam`. The method `GetAllPeerAndRepObjects` is defined in the assembly `CCSS` and returns the union of all rep sets of all peers of the specified object (including the rep set of the object itself).

Listing 2.12: CC++: Fresh Condition

```
Contract.Ensures(ContractPP.Fresh(this.GetAllPeerAndRepObjects().Except(
  Contract.OldValue<ISet<CCObject>>(this.GetAllPeerAndRepObjects()).Except(capturedParam)
));
```

The fresh condition is a free ensure condition.

Captured Parameters Only captured parameters, besides objects that are newly allocated, can be added to an object's rep set during a method. Therefore, captured parameters have to be excluded from the fresh check.

Runtime Check To check the fresh condition during runtime, multiple local ghost variables are required to store all fresh objects and the rep sets at the beginning and at the end of a method. The fresh set with all newly allocated objects is only accessible within the method. It can not be used in a postcondition. The fresh condition must be checked at the end of the method as an assertion.

Listing 2.13 show a typical fresh check at the end of every method.

Listing 2.13: CC++: Fresh Runtime Check

```
Contract.Assert((~SS$postStateObjects).Except(~SS$preStateObjects).Except(capturedParam).
      IsSubsetOf(~SS$fresh));
```

Pre-State Objects The local ghost variable `GhostVariable<HashSet<CCObject>> SS$preStateObjects` stores all objects of all rep sets that can be extended with fresh objects during the method. This variable is assigned at the beginning of the method.

Post-State Objects The local ghost variable `GhostVariable<HashSet<CCObject>> SS$postStateObjects` is filled with the same rep sets as the pre-state variable, but this variable is set at the end of the method. So, it can contain more objects than the pre-state variable. These additional objects must be objects that have been allocated and owned during the method execution.

Fresh Objects The local ghost variable `GhostVariable<HashSet<CCObject>> SS$fresh` contains all objects that are newly allocated during the method execution. After every constructor call and method call, the fresh objects are added to this set.

2.3.5 Modifies Clause

The modifies clause specifies which objects/fields a method is allowed to modify. The objects specified in the modifies clause and the fresh objects must always be a superset of the modifies clauses of all methods called by this method.

Modifies clauses for fields have a slightly different semantics in Spec# than in Dafny. The Spec# modifies clause `modifies o.f` is equivalent with Dafny's modifies clause `modifies o.f, o'f`. The Dafny modifies clause `o.f` allows to modify the object that is assigned to field `f` in object `o`, whereas the modifies clause `o'f` allows to assign a new object to the field `f` in object `o`. CC++'s modifies clauses are similar to the Dafny modifies clauses. Therefore, in our encoding, two modifies sets are maintained per method: One set containing modifiable objects and one set containing modifiable fields.

Listing 2.14 shows how Boogie handles Dafny modifies clauses: For the modifies clause `modifies a` in a method of class `ModifiesTest`, it is (free) ensured that all fields of all objects, that are not equals the object referenced by field `a`, have not been modified at the end of the method. For this, Boogie reasons over all objects in the heap.

Listing 2.14: Boogie: Dafny Modifies Condition

```
free ensures (∀<alpha> $o: ref, $f: Field alpha • { read($Heap, $o, $f) } $o ≠ null ∧ read(
    old($Heap), $o, alloc) ⟹ read($Heap, $o, $f) = read(old($Heap), $o, $f) ∨ $o = read(
    old($Heap), this, ModifiesTest.a) ∨ $o = b#0);
```

In our encoding, we use the rep sets of the specified objects to reason over the objects that are allowed to be modified within a method. This is valid, as the rep sets contain all objects that are potentially allowed to be modified by an object.

Table 2.5 shows which regions are affected by the specified modifies clause. If an object is allowed to be modified, also all transitively owned objects of this object are allowed to be modified and must be in the rep set. A detailed proof for this policy is explained in [2].

Modifies Clause	Modifiable Objects/Fields Set
o.0	{}
o.*	o.rep
o.**	o.owner.rep
o.f	o.rep - {o} + field f

Table 2.5: Modifiable Region per Modifies Clause

The modifies clause `o.0` prevents the method from modifying the object `o` or any transitively owned objects. `o.*` allows the method to modify the object `o` and all its transitively owned objects. `o.**` allows the method to modify the object `o`, all peers of `o` and all transitively owned objects of these objects. `o.f` allows the method to set the field `f` of object `o`, and to modify all objects transitively owned by `o`, but not to modify `o` itself (except setting field `f`).

The modifies clauses are cumulative. If multiple modifies clauses are specified per method, the union of all modifies clauses is built. For example, the resulting superset of `modifies this.0, this.f` is `this.f`.

A method can only modify its owned objects, its peers and its method parameters. So, in the frame condition we only have to reason over the rep sets of these objects and not over the entire heap. This is a valid restriction, because an object can not modify any field outside this scope.

Modifies clauses in CC++ are specified with the statement `ContractPP.Writes`. Listing 2.15 shows the modifies clause for method `InsertTest`. The Spec# modifies clause for this method is specified as `modifies dic.*`. This means, all objects in the `dic` object's rep set are allowed to be modified within this method.

Listing 2.15: CC++: Modifies Clause for Method InsertTest

```
ContractPP.Writes(dic.GetRep());
```

In comparison to the modifies clauses for method `InsertTest`, Listing 2.16 shows the modifies clause for method `Insert`. The Spec# modifies clause is specified as `modifies this.head, this.tail`. This means, only the fields `head` and `tail` of the current object are allowed to be modified and all transitively owned objects from the current object's rep set, excluding the current object itself.

Listing 2.16: CC++: Modifies Clause for Method Insert

```
ContractPP.Writes(this.GetOwnedObjects());
IField<CCObject>[] locations = new IField<CCObject>[] {
        Field<CCObject>.Create(this, "Dictionary.head"),
        Field<CCObject>.Create(this, "Dictionary.tail")
};
ContractPP.Writes<CCObject>(locations, ContractPP.VerificationResults.VerifiedUnder());
```

Modifies clauses are checked within the method for every assignment and every method call. It is not necessary to check these conditions again on method exit. Therefore modifies conditions are *free*, and not *checked*.

Runtime Check To check the modifies condition during runtime, multiple local ghost variables are required to store the rep sets at the beginning of a method and fields mentioned in the modifies clauses.

Modifiable Objects The local ghost variable `GhostVariable<HashSet<CCObject>> SS$modifiableObjects` stores all objects that are allowed to be modified by the current method.

Modifiable Fields The local ghost variable `GhostVariable<HashSet<Field<CCObject>>> SS$modifiableFields` stores all fields that are allowed to be modified by the current method.

The exact use of these ghost variables is explained in the following sections.

2.3.6 Method Calls

Method calls are surrounded by several checks. Before a method call, the caller must check multiple conditions to ensure that calling this method is valid in the current context. After a method call, the caller has to maintain its sets for keep track of newly allocated objects during the method execution.

Frame Condition Check

Before a method call, the caller must check if the modifies clause of the current method satisfies the modifies clause of the method that is to be called. All objects that can be modified by the called method must also be allowed to be modified in the caller method (as they are in the caller's modifies set or in the caller's fresh set).

The frame condition is satisfied, if:

- For every object in the callee's modifiable objects set, it is:
  - The object is in the caller's modifiable objects set
  - Or the object is in the caller's fresh objects set

- For every field in the callee's modifiable fields set, it is;
  - The field is in the caller's modifiable fields set
  - Or the fields object is in the caller's modifiable objects set
  - Or the fields object is in the caller's fresh objects set

Listing 2.17 shows the frame condition check in the method `InsertTest`, before the method call `dic.Insert`. The modifies clause of the `Insert` method is `modifies this.head, this.tail`. These two fields are filled into the modifiable fields set of the callee. The caller method specifies the modifies clause `modifies dic.*`, which includes all owned objects. Therefore, the object `dic` is in the modifiable objects set of the caller, and the frame condition is satisfied.

Listing 2.17: CC++: Method Call Frame Condition Check

```
GhostVariable<ISet<CCObject>> SS$methodModifiableObjects_Insert =
  new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>());
GhostVariable<ISet<IField<CCObject>>> SS$methodModifiableFields_Insert =
  new GhostVariable<ISet<IField<CCObject>>>(new HashSet<IField<CCObject>>());
if (dic != null)
{
  (~SS$methodModifiableObjects_Insert).AddRange(dic.GetOwnedObjects());
  (~SS$methodModifiableFields_Insert).Add(Field<CCObject>.Create(this, "Dictionary.head"));
  (~SS$methodModifiableFields_Insert).Add(Field<CCObject>.Create(this, "Dictionary.tail"));
}
Contract.Assert(CCSS.IsModifiesClauseSatisfied(~SS$methodModifiableFields_Insert,
  ~SS$methodModifiableObjects_Insert, ~SS$modifiableFields, ~SS$modifiableObjects,
  ~SS$fresh));
dic.Insert("foo", 0);
```

Fresh Set Update

At the end of a method, new objects could have been allocated and owned during the method execution. To keep the fresh set of the caller method up-to-date, the fresh set of the callee method must be added to the caller method's fresh set.

Runtime Check Before and after every method call, the objects of all rep sets that can change during the method execution are saved and compared. The difference of the post-state set, without the pre-state and without the captured parameters, are newly allocated objects. These objects are added to the local fresh set. The rep sets are determined similar to the pre-/post-state sets for the fresh check, explained in section 2.3.4.

Method Pre-State Objects The local ghost variable `GhostVariable<HashSet<CCObject>> SS$methodPreStateObjects` stores all objects of all rep sets that can be extended with fresh objects during the callee method. This variable is assigned before the method is called.

Method Post-State Objects The local ghost variable `GhostVariable<HashSet<CCObject>> SS$methodPostStateObjects` is filled with the same rep sets as the method pre-state variable, but this variable is set after the method call. So, it can contain more objects than the method pre-state variable.

Listing 2.18 shows the fresh set update, after the method call `dic.Insert`. During the method execution, new objects could have been added to the object's rep set. After the method call, the difference between the pre- and the poststate sets are added to the local fresh set.

It is possible that references to objects can change within the method call (especially for cascaded objects). Therefore a reference to the original object must be saved to compare the rep sets of the same objects after the method call. The temp variable `SS$dicTemp` would not be necessary in this case, but it is used if sets of method parameters or fields are checked, whose references can change within a method. For instance, if instead of `dic`, the rep set of `dic.head` would be compared, we have to ensure that after the `Insert` method call we keep a reference to the old `dic.head` object.

Listing 2.18: CC++: Method Call Fresh Set Update

```
GhostVariable<ISet<CCObject>> SS$methodPreStateObjects_Insert =
  new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>());
GhostVariable<ISet<CCObject>> SS$methodPostStateObjects_Insert =
  new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>());
if (dic != null)
{
  ~SS$methodPreStateObjects_Insert.AddRange(dic.GetAllPeerAndRepObjects());
  SS$dicTemp = dic;
}
dic.Insert("foo", 0);
if (SS$dicTemp != null)
{
  ~SS$methodPostStateObjects_Insert.AddRange(SS$dicTemp.GetAllPeerAndRepObjects());
}
(~SS$fresh).AddRange(~SS$methodPostStateObjects_Insert).Except(
  ~SS$methodPreStateObjects_Insert));
```

The fresh clause of a method consists of the statement `ContractPP.Fresh(<owner's rep set> - Contract.OldValue(<owner's rep set>))`, as described in section 2.3.4. To get the same fresh set outside the method, all sets specified in the `old` statement have to be saved in ghost variables by the caller before the method call, and subtracted from the same sets after the method call. Thus, the caller gets the set of all objects that have been allocated and owned within the called method.

2.3.7 Expose

The expose statement is used to change the state of an object. An expose consists of the statements unpack and pack. Unpack and pack can only occur in pairs within a method. It is not allowed/possible to unpack an object without packing it again in the same method.

Exposing a delayed object is not valid. An object can not be exposed within its delayed constructor or its delayed method, and so can not the object of a delayed parameter.

Before an unpack, there is a frame condition check to assert that the object is allowed to be unpacked.

There are two different versions of expose: A local expose and an additive expose.

Additive Expose

The additive expose statement allows to un-/pack multiple class frame on the same object. The unpack order must be strictly from the lowest subclass frame up to the base class frame.

Additive expose uses the statements `public static void AdditiveUnpack(this CCObject obj, Type frame)` and `public static void AdditivePack(this CCObject obj, Type frame)`.

Additive Unpack The additive unpack statement sets the `_inv` ghost variable of the specified object to the base type of the specified frame. This means that the object invariants of the specified frame are allowed to be violated, as the specified class frame becomes mutable.

Additive unpack requires the following:

- The specified object must be writable
- The specified object's `_inv` type is equal to the specified frame type
- The specified object's `_localinv` is equal to the allocated type

Additive unpack ensures the following:

- The specified object is mutable on the specified frame type

Additive Pack The additive pack statement sets the `_inv` ghost variable of the specified object to the specified frame type. This means that the object invariants of the specified frame must hold.

Additive pack requires the following:

- All directly owned objects of the specified object are fully valid
- The specified object's `_inv` type is equal to the base type of the specified frame
- The specified object's `_localinv` is equal to the allocated type

Additive unpack ensures the following:

- The specified object is partially consistent up to the specified frame type

Local Expose

With a local expose, at most one class frame of an object can be unpacked. Before another class frame is exposed, all class frames that were exposed before must be packed again.

Local expose uses the statements `public static void Unpack(this CCObject obj, Type frame)` and `public static void Pack(this CCObject obj, Type frame)`.

Local Unpack The local unpack statement sets the `_localinv` ghost variable of the specified object to the base type of the specified frame. This means that the object invariants of the specified frame are allowed to be violated.

Local unpack requires the following:

- The specified object must be writable
- The specified object's `_inv` type must be a subtype of the specified frame type
- The specified object's `_localinv` is equal to the allocated type

Local unpack ensures the following:

- The specified object is mutable on the specified frame type

Local Pack The local pack statement sets the `_localinv` ghost variable of the specified object to the specified frame type. This means that the object invariants of the specified frame must hold.

Local pack requires the following:

- All directly owned objects of the specified object are fully valid
- The specified object's `_inv` type must be a subtype of the specified frame type
- The specified object's `_localinv` is equal to the base type of the specified frame type

Local unpack ensures the following:

- The specified object is fully consistent

Listing 2.19 shows the expose statements of the method `Insert`. Before every unpack, there is a check that ensures that the object that is unpacked is allowed to be modified. Before every pack (local or additive), there is call to the object's `SS_Inv` method, to ensure that the object invariants are satisfied before the object is packed.

Listing 2.19: CC++: Local Expose

```
Contract.Assert(CCSS.IsObjectModifiable(this, ~SS$modifiableFields,
  ~SS$modifiableObjects, ~SS$fresh));
CCSS.Unpack(this, typeof(Dictionary));
...
Contract.Assert(this.SS_Inv());
CCSS.Pack(this, typeof(Dictionary));
```

The reason why the frame condition check before `Unpack` and the object invariants check before `Pack` are separate checks and are not part of these method's precondition is, that these checks correspond with the sscBoogie encoding, which has also separate checks. Other reasons are of technical nature. The frame condition check before `Unpack` is separate, to simplify the `Unpack` signature. The invariant check before `Pack` is separate, because the method that is called to check the object invariant depends on exposed class frame. The correct method call must be added statically during compilation, as it is not possible in C# to dynamically bind a method call to a dynamic type. (In order to check the object invariant as precondition of the method `Pack`, the statement `((type)obj).SS_Inv()` must be allowed, where `type` is a variable of type `Type`. This is not possible in C#, without using reflection.)

Rep Update

A difference of our encoding to the Boogie encoding is the handling of owned objects. Boogie only considers directly owned objects, whereas our encoding takes all transitively owned object into account. Thus, we have to do additional work in maintaining the rep sets. The reason for the different rep set handling is the handling of the modifies condition and frame condition.

If the owner relationship between objects changes, the rep set of the owner object is updated and the rep set of the newly owned object is added to the owner's rep set. The rep sets of the transitive owners (above the direct owner) remain unchanged. The rep set can only change for writable objects, therefore their owners must be exposed. At the end of every expose (local or additive), the rep set of the exposed object is updated. During the update, the rep set is defined as the union of the rep sets of all directly owned objects, which are consistent, and the current object. With this, all valid objects have an up-to-date rep set.

This method is correct, because only one object in the ownership cone can be in the state consistent. All objects lower in the hierarchy are committed, and have an up-to-date rep set. And all objects higher in the hierarchy are exposed and do not require an up to date rep set.

Figure 2.7 shows how the rep set of an object is updated when the object is packed. After the pack, all directly owned consistent objects become committed, and the rep set of the packed object is a superset of all transitively owned rep sets.

2.3.8 Constructor Calls

A constructor's fresh clause is a special case of a normal method call's fresh clause. At creation time of a new object, the object does not own any object, so its rep set's only element is the object itself. At the end of the constructor, the entire rep of the new object must be fresh (except captured parameters). An object in construction can not become owned within its constructor, as it is not in a valid state. Therefore, no rep set of a potential owner can be modified within a constructor and it is sufficient to add the entire newly created object's rep set to the fresh set of the caller method. In a normal method call, the difference of the owner rep set before and after the method call would be added to the fresh set. For constructors, as optimization, the entire rep set of the constructed object can be considered fresh, as after the constructor, the owner of the constructed object is a peer group placeholder and its only rep elements are the object itself and objects that have been newly allocated and owned within the (non-delayed) constructor. No other peer elements can be in the peer group placeholder. Therefore, the owner rep set is equivalent to the object's rep set, and no pre- and poststate comparison must be made.

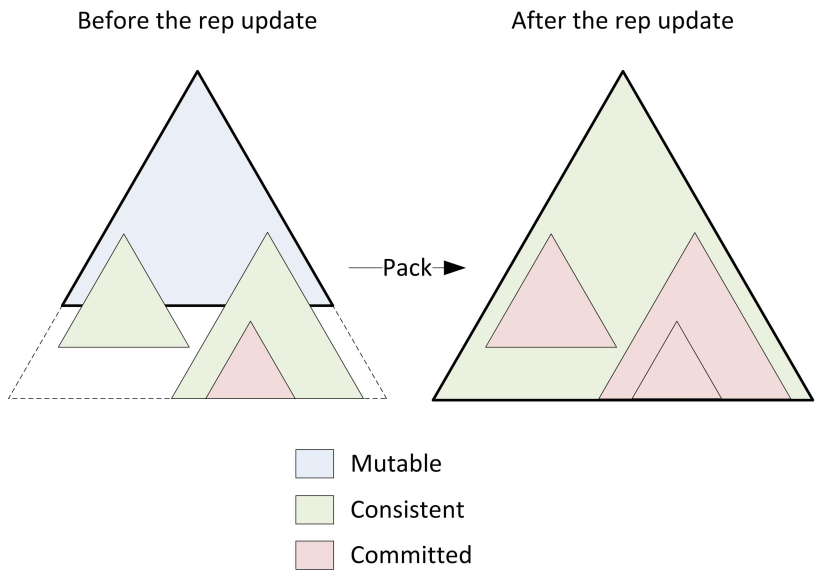

Figure 2.7: Rep Update

Listing 2.20 shows the creation of a new `Node` object within the method `Insert`. After construction, the entire rep set of the `Node` is added to the fresh set. The method `GetRep` returns the rep set of an object.

Listing 2.20: CC++: Constructor Call

```
Node SS$nodeTemp = new Node(key, val);
(~SS$fresh).AddRange(SS$nodeTemp.GetRep());
```

Parameters

If the constructor has normal parameters, then for these parameters fresh checks with pre-/poststate comparison are required, as newly allocated objects can be owned by the passed objects. Parameters are handled the same as in normal method calls, as described in section 2.3.6.

Captured Parameters

The handling for captured parameters in constructor calls is similar to the handling of captured parameters for normal method calls, as described in section 2.3.4. Captured parameters must be excluded from the method's fresh set, as the have already been allocated outside the method.

2.3.9 Local Variable Assignment

No special handling is required for local variable assignments within a method. Even if the local variable is non-null, there is no explicit non-nullness check. The non-null test is done by the type system of the Spec# compiler, which outputs a warning for possible non-null conversions. Non-nullity for local variables is inferred by a data flow analysis of the Spec# compiler and is not checked again in Boogie.

2.3.10 Field Assignment

Field assignments are handled differently to normal local variable assignments. Depending on the type of a field (`[Rep]`, `[Peer]`), the owner relationship of the assigned object can change and different frame conditions must be asserted.

Frame Condition Checks

Fields can be specified in modifies clauses and object invariants. Before a field assignment, there is a frame condition check which asserts that the field is modifiable in the current context.

The frame condition is satisfied, if:

- The object which declares the field is not null (and also all objects in the transitive field hierarchy are not null; e.g., `a != null` ∧ `a.b != null` ∧ `a.b.c != null`)
- The object which declares the field is writable
- The field is in the method's modifiable fields set
  or the fields declaring object is in the method modifiable objects set
  or the fields declaring object is in the method's fresh set

Listing 2.21 shows the field assignment to `this.head` in the method `Insert`. This frame condition check succeeds, because the modifies clause of this method is `modifies this.head, this.tail`. The `head` field is element of the modifiable fields set of this method.

Listing 2.21: CC++: Field Assignment Frame Condition Check

```
Contract.Assert(this != null && this.IsWritable());
GhostVariable<Field<CCObject>> SS$assignedField = new GhostVariable<Field<CCObject>>(
  new Field<CCObject>(this, "Dictionary.head"));
Contract.Assert(CCSS.IsFieldModifiable(~SS$assignedField, ~SS$modifiableFields,
  ~SS$modifiableObjects, ~SS$fresh));
this.head = SS$nodeTemp;
```

Object Invariants Checks

Field updates must be restricted as fields can be used in object invariants. After a field assignment, the object invariants must be checked if they are still satisfied, or if not, then the state of the object that defines the field must be mutable.

In our encoding, after a field update, we check all object invariants of the involved class frame. It would be sufficient to only check the object invariants that depend on the updated field, as done in Boogie. But for this, the object invariants would have to be separated with an own method per object invariant. Checking all object invariants in this case is correct, since, if the object was valid, before the field update the object's invariants were true, and after the field update only the object invariants that depend on the updated field are possibly violated.

Listing 2.22 shows the object invariant check after the field assignment to `this.head` in the method `Insert`. This assertion succeeds, because the `Dictionary` class frame of `this` is exposed at this point in the program.

Listing 2.22: CC++: Rep Field Assignment Object Invariant Check

```
this.head = SS$nodeTemp;
this.head.UpdateOwnersForRep(this, typeof(Dictionary));
Contract.Assert(this.IsMutable(typeof(Dictionary)) || this.SS_Inv());
```

Additive Fields After an assignment to an additive field (fields that are decorated with the attribute `[Additive]`), it must only be asserted that the object, that declares the field, is additively exposed. An object invariant check here would be invalid, because additive fields can be used in object invariants of subclasses, and the assignment is allowed to break these invariants until the corresponding class frame is additively packed.

Instead of directly after the assignment, the object invariants are checked before the pack. This behavior is valid, as assignments to an additive field require the object to be additively exposed.

Listing 2.23 shows the frame condition check, if the `head` field would be additive. Then, it is asserted the object is additively exposed and the object invariant check is delayed till the additive pack.

Listing 2.23: CC++: Additive Rep Field Assignment Object Invariant Check

```
this.head = SS$nodeTemp;
this.head.UpdateOwnersForRep(this, typeof(Dictionary));
Contract.Assert(this.IsAdditiveMutable(typeof(Dictionary)));
```

Rep Fields

Objects that are assigned to fields, that are decorated with the attribute `[Rep]`, are automatically owned by the object and the class frame that declares the field. Automatic rep owner assignments use the statement `public static void UpdateOwnersForRep(this CCObject subject, CCObject owner, Type ownerFrame)`.

Listing 2.22 shows the `Node` that is assigned to the rep field `this.head` becomes owned by `this`.

Rep owner update requires the following:

- The owner is writable
- The object is unowned or has already the object as owner that declares the rep field
- The owner is exposed on the owner frame that the field object is assigned to
  or the object is peer consistent

Rep owner update ensures the following:

- The object is owned by the object that declares the rep field
- The owner rep set is a superset of the assigned object's rep set

Peer Fields

Generally, in Spec# it is easier to deal with fields that are declared as `[Rep]`. But if aggregated objects have the same owner and these objects have references to each other, they are declared as `[Peer]`. Objects that are assigned to fields that are decorated with the attribute `[Peer]`, implicitly become owned by the same object and class frame as the object that declares the field. Automatic peer owner assignments use the statement `public static void UpdateOwnersForPeer(this CCObject subject, CCObject peer)`.

Listing 2.24: CC++: Peer Field Assignment Object Invariant Check

```
this.head.next = h;
this.head.next.UpdateOwnersForPeer(this.head);
Contract.Assert(this.head.IsMutable(typeof(Node)) || this.head.SS_Inv());
```

Listing 2.24 shows the `Node` assignment to the peer field `this.head.next` in the method `Insert`. The owner of `this.head.next` is set to the same owner and the same owner frame as `this.head`.

Peer owner update requires the following:

- The peer is writable (unowned or the owner is mutable on the owner frame)
- The object is unowned or has already the same owner as the peer

Peer owner update ensures the following:

- The object is owned by the same owner as the peer
- The owner's rep set is a superset of the assigned object's rep set

The preconditions of every method check for peer consistency. If an object is exposed, no method on its peers can be called.

2.3.11 Array Element Assignment

Assignments to an array element are preceded by a covariance check and a boundary check. The covariance check asserts that the element that is assigned, is a subtype of the arrays element type. The boundary check asserts that the index is within the boundaries of the array.

Listing 2.25 shows an assignment to an array `arr` at index `i`.

Listing 2.25: CC++: Array Element Assignment Boundary and Covariance Check

```
  Contract.Assert((0 <= i) && (i < this.arr.Length));
  Contract.Assert((SS$elementTemp == null) || CCSS.IsSubtype(SS$elementTemp.GetType(), this.arr
      .GetType().GetElementType()));
  this.arr[i] = SS$elementTemp;
```

These checks are not necessary during runtime, as both cases would cause a runtime error during execution. The assertions are mainly for the verifier.

2.3.12 Contract Inheritance and Virtual Methods

Virtual methods inherit the preconditions, postconditions and modifies clauses from the overridden method of its base class [10]. An overridden method in Spec# can have stronger postconditions, but it is not allowed to specify any additional precondition.

The Spec# compiler extends overridden methods with the correct contracts. And ccrewrite automatically inherits pre- and postconditions from virtual base methods. This causes duplicate contracts that are rejected by ccrewrite. To prevent this situation, the CC++ code generator ensures that contracts are specified only once per inheritance structure.

2.3.13 Pure Methods

Pure methods are methods without side-effects. Besides in the user code, they are allowed to be used in object invariants and method contracts.

They have different preconditions than normal methods [9]. They require the declaring object to be peer valid, but do not require anything about its owner.

Pure methods are not allowed to modify any fields of old objects (they must be side-effect free) and they are not allowed to have any modifies clause stated. Therefore, they have an empty modifies set.

Allocating new objects within a pure method and returning them is valid. But a newly allocated object can not become owned within the method, therefore pure methods do not require a fresh check at the end of the method. Methods that call pure methods do not have to update their fresh set after a pure method call.

Pure methods are allowed to have user defined pre- and postconditions.

Listing 2.26 shows the specification for a pure method.

Listing 2.26: CC++: Pure Method

```
[Pure]
internal void PureMethod()
{
  Contract.Requires(this.IsPeerValid());
  Contract.Ensures(this.IsPeerValid());
}
```

Pure methods are decorated with the attribute `[Pure]`.

2.3.14 Loop Invariants

Loop invariants as separate language feature are not supported by CC++. Loop invariants can be derived by Clousot. To support Clousot and to ensure to have the same loop invariants as in Spec#, inline assertion statements with the loop invariants are added to the loop.

Loop invariants must hold on loop entry and must remain true in every loop iteration and on loop exit.

There are four types of loops in Spec#: *ForEach*, *For*, *While*, *DoWhile*. The Spec# Compiler (ssc) converts all loops, except *DoWhile*, into *While(true)* loops, represented in the IL as unconditional branch. The jump out of the loop can be conditional (only for *DoWhile*) or unconditional.

Loop invariants can only contain variables that are also available outside of the loop scope. For instance, elements that are returned by the enumerator in *ForEach* loops can not be used in loop invariants, because they are only accessible inside the loop.

User defined loop invariants are available as serialized IL string parameter of the method `AssertionHelpers.LoopInvariant(string, string)`.

The loop invariants are translated into assertions. These assertions are checked at the beginning of each loop iteration before the condition check. Given the structure of the loops after compilation these checks are sufficient to prove the invariants.

2.3.15 Old

In Spec# postconditions, the `old` keyword can be used to refer to the method's pre-state. `old()` statements are replaced with the CodeContracts statement `Contract.OldValue<T>(T value)`, where T is the declared type of the passed value.

2.3.16 Result

Spec# `result` statements can be used in postconditions and refer to the method's return value. They are replaced with the statement `Contract.Result<T>()`, where `T` is the return type of the method the postcondition is declared in.

2.3.17 Inline Assertions

Inline assertions are used to specify that the specified condition must hold at a certain program point. These assertions of the form `assert <condition>` are translated into `Contract.Assert(<condition>)`.

The difficulty here is, the condition in the AST is only available as string. The condition is recovered from this string using the *ContractDeserializerContainer* of the Spec# compiler framework.

2.3.18 Inline Assumptions

Inline assumptions are used to give the program verifier additional conditions it can rely on. These assumptions of the form `assume <condition>` are translated into `Contract.Assume(<condition>)`

Similar to inline assertions, the condition has to be recovered from a serialized string.

2.3.19 Quantifiers and Reductions

Quantifiers and reductions can be used in pre- and postconditions, invariants, loop invariants and normal code. They work on integer ranges (e.g., `forall{int i in (0: arr.Length); arr[i] > 0}`) and on collections (e.g., `forall{int i in arr; i > 0}`).

Table 2.6 shows all quantifiers and reductions supported by Spec# and CC++.

These quantifiers and reductions are redirected to static methods in the `CCSS` class. For all quantifiers and reductions, there are two possible signatures. E.g., `bool ForAll(int fromInclusive, int toExclusive, Predicate<int> predicate)` and `bool ForAll<T>(IEnumerable<T> collection, Predicate<T> predicate)`.

Unlike the quantifier methods in the class `System.Diagnostics.Contracts.Contract`, the `CCSS` quantifier and reduction methods return `true` for empty sets. (The `Contract` class throws an exception if `fromInclusive > toExclusive`. This behavior is different from the one in Spec#)

Quantifiers and reductions are translated into delegates.

Quantifier and Reductions	Description
ForAll	The expression must be true for all elements
Exists	The expression must be true for at least one element
Count	Counts the number of element for which the expression is true
Sum	Adds up all expressions
Product	Gets the product of all expressions
Min	Gets the expression that evaluates to the lowest value
Max	Gets the expression that evaluates to the highest value
ExistsUnique	The expression must be true for exactly one element

Table 2.6: Quantifiers and Reductions

2.3.20 Owner Class

The class `Microsoft.Contract.Owner` contains pure and non-pure methods that can be used in contracts, invariants and within the code. They are used to change or evaluate an object's owner relationship. The `Owner` method calls are translated to `CCSS` method calls with the same signature.

Table 2.7 shows how the `Owner` method calls are mapped to `CCSS` methods.

Spec# Owner Method	CCSS Method
Assign	OwnerAssign
AssignSame	OwnerAssignSame
Different	OwnerDifferent
Is	OwnerIs
None	OwnerNone
Same	OwnerSame
ElementProxy	*not implemented*
New	*not implemented*

Table 2.7: Owner Call Mapping

The methods `ElementProxy` and `New` are not supported. `ElementProxy(arr)` returns an object that is a peer to all elements in an array `arr`. `New` is similar to `None` but slightly stronger, as it requires that the owner of the specified object is a peer group placeholder and the owner frame points to the object itself.

2.3.21 Result Not Newly Allocated Attribute

The method attribute `[ResultNotNewlyAllocated]` can be used to specify that the object that is returned by a method was not newly allocated within the method. This attribute is only valid for pure methods. The effect of this attribute is a new not-fresh check at the end of the method. For callers of this method, no additional handling is required, because if the result is not a new object, it is not in the fresh set of the method.

Listing 2.27 shows how an object `o` is checked for freshness before it is returned from a method that is decorated with the `[ResultNotNewlyAllocated]` attribute. The object was not allocated within the method or any method called by this method if it is not in the method's fresh set.

Listing 2.27: CC++: Result Not Newly Allocated

```
Contract.Assert(!(~$fresh).Contains(o));
return o;
```

2.3.22 Inside Methods

Methods that are decorated with the attribute `[Inside]` can be called if the object is exposed on the class frame that declares the method. The attribute `[Inside]` removes the default pre- and postconditions. Instead it is required and ensured that the object is exposed on this class frame. Such methods can not assume that the object invariants hold at the beginning of the method, and the caller of this method can not assume the object invariants to hold at the end of the method call.

Listing 2.28 shows how the method contracts are adapted if the method `InsideMethod` is decorated with the attribute `[Inside]`.

Listing 2.28: CC++: Inside Method

```
class InsideTest : CCObject
  /*[Inside]*/
  void InsideMethod()
  {
    Contract.Requires(this.IsMutable(typeof(InsideTest)));
    Contract.Ensures(this.IsMutable(typeof(InsideTest)));
  }
}
```

2.3.23 Additive Methods

Methods that are decorated with the attribute `[Additive]` have special default contracts which allow the object to be additively exposed within the method. It is required that `_inv` of the current object points to the the class frame in which the method is defined, so an additive expose on the current class frame is valid. The object itself must be writable.

Listing 2.29 shows how the method contracts are adapted if the method `AdditiveMethod` is decorated with the attribute `[Additive]`.

Listing 2.29: CC++: Additive Method

```
class AdditiveTest : CCObject
  /*[Additive]*/
  void AdditiveMethod()
  {
    Contract.Requires(this.IsAdditiveExposable(typeof(AdditiveTest)));
    Contract.Ensures(this.IsAdditiveExposable(typeof(AdditiveTest)));
  }
}
```

Chapter 3

Implementation

This chapter explains the implementation of the translation rules. Section 3.1 describes the architecture of the CC++ code generator as part of the sscBoogie compiler backend and where the translation takes part in the tool chain. Section 3.2 explains how additional Spec# language features, that have not been covered by the previous chapter, are encoded and translated. In section 3.3 we justify important design decisions. The last section 3.4 lists the options available to configure the translation process.

3.1 Architecture

This section explains the architecture of the CC++ code generator. We focus on the sscBoogie compiler backend and omit the overall structure of sscBoogie.

3.1.1 Toolchain

Figure 3.1 shows the idealized tool chain of the possible translation paths. Besides the existing Boogie code generation path, sscBoogie is extended to generate Pex and Clousot compatible code.

The Spec# compiler (ssc.exe) generates intermediate language (IL) code, with the contracts encoded as class-/method-attributes, Spec# specific runtime checks and marker method calls.

sscBoogie removes the Spec# runtime checks and meta data and replaces them with CC++ equivalents.

The CC++ rewriter (ccpprewrite) (which is currently included in sscBoogie, but will be excluded in a separate executable, so it can be used as independent tool) replaces the CC++ declarations, which act as markers and have no intrinsic effects, with runtime checks and CodeContracts method calls.

The CodeContracts rewriter (ccrewrite) replaces the CodeContracts method calls with runtime checks and rewrites the code so that it can be executed.

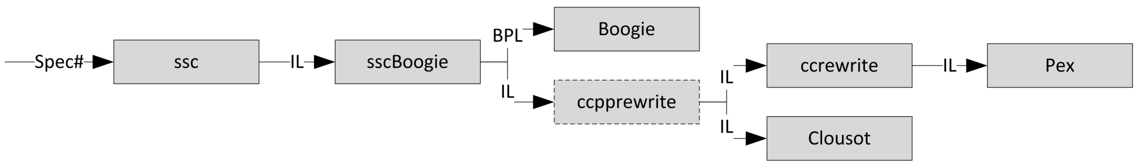

Figure 3.1: Idealized Toolchain (ccpprewrite is currently part of sscBoogie)

Pex requires the code to be rewritten, before it can generate tests. Clousot works on the un-rewritten code.

3.1.2 Overview

The CC++ code generator is part of the *BytecodeTranslation* project, within the sscBoogie solution.

Figure 3.2 shows the class diagram of the sscBoogie compiler backend, especially of the code generation part. The shaded classes are the newly added classes for the CC++ code generation. The white classes are the corresponding Boogie code generation classes. All classes that handle CC++ code are marked with the prefix `CC` in their name.

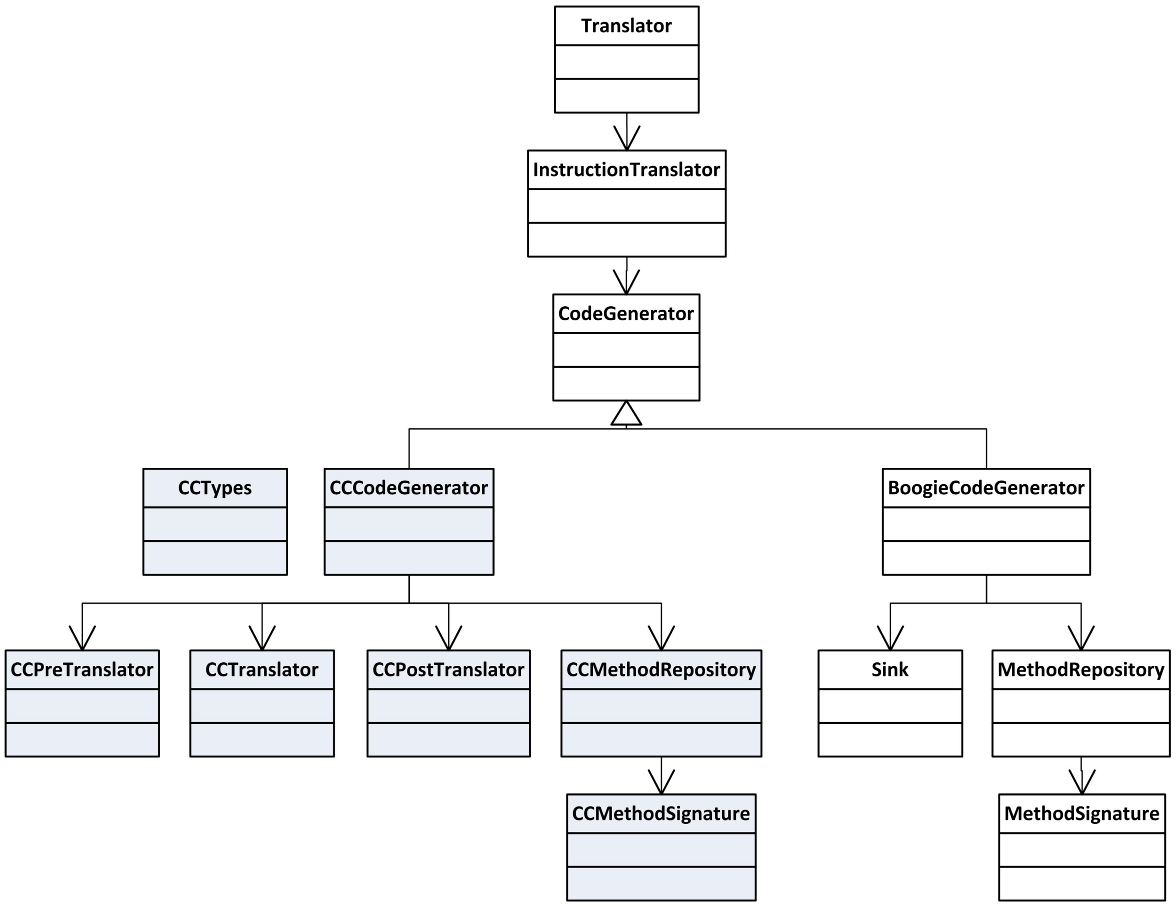

Figure 3.2: Class Diagram

For the encoding, the compiler backend of sscBoogie was extended. The code generation part of sscBoogie was extracted and abstracted by a common abstract `CodeGenerator` base class. Both, the Boogie code generator and the newly created CC++ code generator inherit from this `CodeGenerator` class, which contains shared functionality.

The structure of the CC++ code generation classes is similar to the Boogie code generator classes. For nearly every Boogie specific class, there is an equivalent class for CC++. The code generation independent classes are shared.

The CC++ code generator works on the normalized abstract syntax tree (AST). The normalized AST contains only basic language features, that directly correspond to IL statements. High level language constructs, such as logical operators have been replaced during compilation, using short-circuiting. Also, if and while are replaced with conditional and unconditional branches. This causes some additional work in recovering higher level language features to determine the required translation steps.

The classes `CCPreTranslator`, `CCTranslator` and `CCPostTranslator` inherit from the `StandardVisitor` class, which operates on the AST and visits all nodes in it. These three visitor classes are called on the AST in the order in which they are listed. The class `CCPreTranslator` resets properties on specific nodes and removes unused statements. The class `CCTranslator` does the real translation from Spec# to CC++. At the end of the translation process, class `CCPost-Translator` removes Spec# specific methods and attributes that are not required anymore from the translated classes. The class `CCTypes` is a helper class which contains a reference to all method nodes that are added to the AST during the translation, and is the pendant to the `Types` class in the compiler framework.

3.1.3 Ownership Methodology Model

The Spec# ownership methodology is modeled in a separate project, called *CodeContractsForSpecSharp*. This is a separate assembly, containing the `CCObject` custom object class, and static methods for manipulating the ownership state ghost variables. All methods that manipulate the `CCObject` fields are part of the static `CCSS` class. This class contains the ownership model logic as well as helper methods.

The main reason why the ownership model is in a separate assembly and is not directly compiled into the translated code is, it is much easier to change methods in a separate C# project than in sscBoogie.

Figure 3.3 shows the dependencies between the assemblies, before and after the translation. The *CodeContractsForSpecSharp* assembly is referenced, together with the *CCPlusPlus* assembly, in every assembly that is translated.

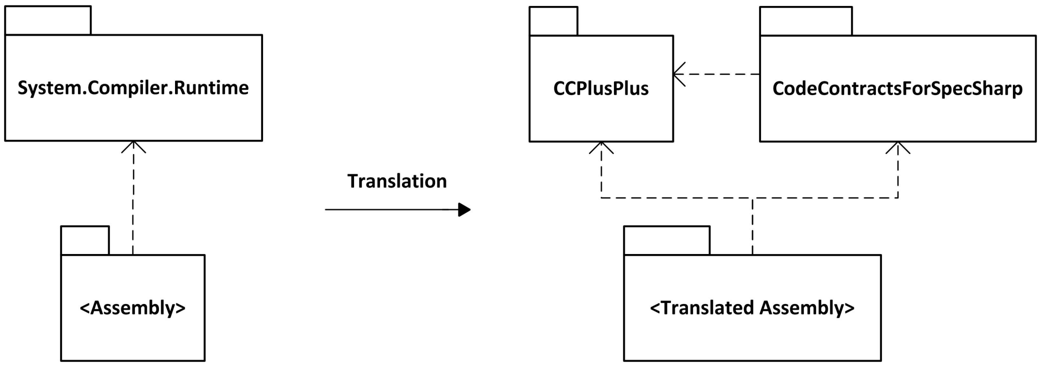

Figure 3.3: Dependencies

Spec# assemblies depend on the assembly *System.Compiler.Runtime*, which contains the *Microsoft.Contracts* namespace. During the translation, all dependencies to this assembly are removed, so it is not longer required after the translation.

3.1.4 Garbage Collection

Ghost variables store references to objects. These referenced objects can not be collected by the garbage collector (GC), even if the objects are not used anymore and not referenced anywhere else. To allow the garbage collector to collect objects whose only references exist in ghost variables, weak-references can be used. The `_owner` reference is not allowed to be weak, because the owner state of an object is required to determine the state of the object itself. Therefore, the owner must always be available, even if it is not referenced anywhere else (and could potentially be collected by the GC).

Weak references are not used in our current implementation, since it is not clear how provers can handle them. Additional work must be done in analyzing provers behavior with weak references.

In our case garbage collection is not a big issue, because the translated Spec# code is only used for static analysis, and is not executed, or only parts of the code are executed in unit-tests. Therefore, GC was not considered an issue in this thesis.

3.1.5 Nested Calls

Spec# statements can consist of nested calls, whose method output is the direct input of another method. E.g., `string.IsNullOrEmpty(o.ToString())`. Because every method call requires additional checks, cascaded calls are entangled, aligned according to their execution order and the results of the method calls are stored in temporary variables, which are then passed to the next method call. This partitioning allows to add checks between the method calls and to include the method results in the checks.

3.1.6 Translation Output

sscBoogie takes an assembly that was compiled with the Spec# compiler and outputs the translated assembly. The translated assembly is saved with the suffix *_ccpp* to the same directory as the input assembly.

The module name of the resulting assembly is also extended with the suffix *_ccpp*. This is required for Pex. Pex can only work with assemblies whose module name is equal to the assembly name.

The resulting assembly is targeted for the .Net runtime 4.0, even if Spec# assemblies are compiled for .Net 2.0, as the translated assembly uses contract classes from the .Net 4.0 standard libraries.

3.1.7 Custom Standard Visitor vs. Generic Visitors and Code Generators

The Boogie code generator/translator works on the control flow graph (CFG) or *flattened* AST [3]. This graph can not be translated back to (useful) IL code. It is for code analysis only. Therefore a separate standard visitor must be used that works on the normal intermediate representation (IR) AST. The consequence and disadvantage of this approach is, the code generation part can not be fully abstracted away from the visitor part. Boogie and CC++ must have their own visitors.

3.2 Language Features

This section explains the translation of special language features that have not been handled in the previous chapter. The focus of this section is mainly on the implementation of technical details and less on the ownership model itself, as was the focus of the previous chapter.

3.2.1 Modopt

The Spec# compiler adds `modopt(NonNullType)` statements to the IL code to distinguish between nullable and non-nullable types. These `modopt` statements change the signature of methods [26]. They are replaced by non-null checks and are removed during the translation, as they are not used anymore.

3.2.2 Runtime Contracts Attribute

The ccrewrite tool uses a `System.Diagnostics.Contracts.RuntimeContractsAttribute` module attribute to determine if the module was already rewritten. Spec# uses the same attribute to annotate its modules. In order to process the translated module with ccrewrite, these attributes are removed during the translation process.

3.2.3 Compiler Generated Names

The CC++ code generator generates names for created methods and variables that start with the prefix `SS_` and `SS$`. These prefixes should not be used in user code.

3.2.4 Delegates

For anonymous delegates, sealed closure classes are generated, which contain a function for each delegate. Because these delegates are used in pre- and postconditions, the closure classes and its methods visibility is changed to public, so they are accessible outside the method.

Listings 3.1 and 3.2 show how an `exists` quantifier is translated into a delegate. A new class `closure` is generated, which contains a method `Function` that contains the delegate body.

Listing 3.1: Spec#: Exists Quantifier

```
class Foo{
  public void QuantifierTest(int[]! arr)
    requires exists{int i in (0: arr.Length); arr[i] > 0};
  {}
}
```

Listing 3.2: CC++: Exists Quantifier

```
class Foo{
  public class closure
  {
    int[] arr;
    Foo thisvalue;

    public closure(Foo thisvalue, int[] arr)
    {
        this.thisvalue = thisvalue;
        this.arr = arr;
        base.ctor();
    }

    public bool Function(int i)
    {
        return this.arr[i] > 0;
    }
  }

  void QuantifierTest(int[] arr)
  {
    closure: SS$Closure = new closure(this, arr);
    Contract.Requires(CCSS.Exists(0, arr.Length, new Predicate<int>(SS$Closure.Function)));
  }
}
```

Generating delegates in sscBoogie involves all steps of the compilation process, from the Scoper to the Optimizer.

3.2.5 Singe Point of Return

Every method in the normalizer AST contains only a single point of return, which simplifies the task of adding fresh checks and additional assertions that must be checked before the method returns.

3.2.6 ssc Generated Runtime Checks

The Spec# compiler (ssc) automatically generates runtime checks for user defined pre- and postconditions that throw `RequiresExceptions` or `EnsuresExceptions`. These checks are not required after the translation to CC++, as these conditions are moved into `Contract.Requires` and `Contract.Ensures` statements during translation. The compiler generated checks (that are the first and the last statement in the method body) are removed by the pre-translator. These checks are surrounded by a try-catch block which handles the `ContractMarkerException` and rethrow the exception.

3.2.7 Attributes

All Spec# attributes are removed from the translated code (e.g., `EnsuresAttribute`, `RequiresAttribute`, `RepAttribute`, `PeerAttribute`, `WitnessAttribute`), or replaced with CodeContracts attributes (e.g., `PureAttribute`), as they are not used anymore.

Pure

The attribute `Microsoft.Contract.PureAttribute` is replaced with the attribute `System.Diagnostics.PureAttribute`.

3.3 Design Decisions

This section justifies some of our most important design decisions.

3.3.1 Ownership Model

To model the Spec# ownership methodology in CC++, some ghost variables are required for bookkeeping (e.g., `_owner`, `_ownerframe`, `_inv`, `_localinv`, `_rep`), as explained in the previous chapters.

Where can these ghost variables be stored?

- Global static dictionary. Advantage: simple implementation. Disadvantage: Pex and Clousot have issues with static references to complex data structures.
- Require all objects to implement a specific interface that provides properties for accessing the ghost variable fields. Disadvantage: Static type checking gets lost, because objects are never passed between methods of this interface type. Before every check, a cast to the interface would be required.
- Add the ghost variable fields directly to the default `System.Object` class. Disadvantage: This is not possible without recompiling the .Net standard libraries.
- Inserting a new custom object class between the default `System.Object` base class and the classes that derive from `System.Object`. Disadvantage: With this approach, all .Net standard libraries have to be recompiled, so that all standard classes derive from the custom object class.
- Inserting fields to every object dynamically during runtime with the `Reflection.Emit` library. This is not possible, as `Reflection.Emit` just returns a new type with the added fields, but does not modify the original object.
- Compiler inserts fields to every class statically. This has the same disadvantages as the custom object class approach. Additionally, no helper methods can be used, as all checks have to be statically bound to the object types.
- Replace objects with proxy objects, as done in Aspect Oriented Programming.
- Redirect calls at runtime to custom objects, as done in Moles [27].

Generally, it is much more complicated for a program verifier to reason over dynamically modified code, if not even impossible. Extending classes statically at compile time is always the better approach for software verification.

In our implementation we use a custom object class which contains the required ghost variables. As a future extension, a hybrid solution with a global static dictionary can be implemented to support already compiled classes.

3.3.2 Frame Guard

Spec# adds runtime checks for pre-/postconditions, object invariants and expose statements to the IL code. The checks are handled in the class `Guard`, which is instantiated for each object.

The `Guard` class does not consider class frames, invariant levels or frame conditions. Some `Guard` methods that are added by the Spec# compiler are empty method declarations, which are used as marker for sscBoogie.

The `Guard` class can not be extended or adapted for our needs. Therefore, we remove all references to this `Guard` class from the code instead of reusing it. All functionality provided by the `Guard` class is also supported by our `CCSS` class.

3.3.3 Frame Condition Evaluation

All `ContractPP.Modifies` conditions are replaced by frame condition checks before every field assignment or method call. Even though, these frame condition checks can occur somewhere in the code, the sets with modifiable objects and fields are determined at the beginning of the method. It would most probably be sound to determine the sets as they are checked in the frame conditions (and possibly contain more objects; e.g., captured parameters). But this would not represent the behavior of the Boogie encoding, as the frame condition in Boogie uses the state at the beginning of the method (with `old($Heap)`) to reason over modifiable objects.

3.3.4 Inv / Local Inv Modeling

The `_inv` and `_localinv` ghost variables could have been added to every class frame as `boolean`, instead of two `Type` fields. The idea behind this is, that it is easier for the verifier to reason over boolean fields instead of type fields. This would require that all checks have to be added statically to the code during compilation. No helper methods could be used, as is used in the current implementation with the `CCSS` library.

Although, this could be a possible solution for the problem of Pex with creating instances of type `Type`, as described in section 4.3.

Even if the `boolean` fields solution has some benefits, we decided to stick to the same approach as is used in Boogie and to use `Type` fields, to avoid diverging from the original model.

3.4 sscBoogie Options

This section describes all options that are supported by the the CC++ code generator. Not all available options are supported. Some Spec# options are not relevant for the CC++ code generation. Others have not been implemented due to time restriction.

3.4.1 Spec#

This section contains all Spec# options that are supported by the CC++ translation process.

modifiesDefault

The *ModifiesDefault* option specifies the modifies clauses that are added to each method by default, in addition to the user specified modifies clauses. (See Table 3.1)

Value	**Default CC++ Encoding**
0-4	no default modifies clause
5 (default)	(1) plus modifies this.* (only for mutable classes)
6	(5) plus modifies <obj>.* for all declared objects

Table 3.1: modifiesDefault Option

ModifiesDefault values 1 - 4 would allow to modify the variable `exposeVersion`, which is used in the Boogie encoding to determine if an object is exposable. In our encoding, the separation of `exposeVersion` is not possible anymore. This is a small limitation of our encoding in terms of granularity of modifies conditions compared to Boogie. In our encoding, specifying a field or an object as modifiable always implies the object to be exposable. Values 1-4 generate the same modifies clauses as value 0, which means "just what is declared".

3.4.2 CC++ Translation

The CC++ translation has some additional options. The options are parsed in the class `CCCommandLineOptions`.

Table 3.2 shows the supported options for the CC++ translation.

Spec# Parameter	**Description**
-ccprint	Translates the specified assemblies to CC++
-ccrewrite	Rewrites the translated assemblies with ccrewrite.exe. Includes -ccprint.
-skipCcpp	CC++ statements that are not guaranteed to be well-formed, are removed from the code. This parameter is important if the code is going to be executed. Otherwise, null-reference exceptions are possible to occur.
-pex	Runs Pex after translation. Includes -ccprint, -ccrewrite and -skipCcpp. The Pex wizard creates a test project and then starts Pex on it
-disableAssertUI	Disables the assert user interface. No assertion exception dialog is shown if an assertion fails. This flag is used for the test suite
-throwException	Throws an exception if an error occurs during translation. Otherwise, only outputs an error message

Table 3.2: CC++ Translation Options

The parameter *-ccprint* is used, if the translated code will be analyzed with Clousot. After translation, all CC++ and CodeContracts markers remain in the code. The code is not executable, as this is prevented by CodeContracts. Because the ccpprewriter is currently part of sscBoogie, CC++ runtime checks are already in the code. They are guaranteed to be well-formed.

The parameter *-ccrewrite* translates the assembly to executable code. The code contains CC++ and rewritten CodeContracts declarations. Because parameters passed to CC++ markers are not guaranteed to be well-formed, runtime-exceptions are possible (e.g., `ContractPP.Writes(a.b.c.d)` where one of these fields is null). To remove the CC++ markers and only leave CC++ runtime checks in the code, which contain well-formedness checks, the parameter *-skipCcpp* can be used.

Pex can be started directly from sscBoogie, with the parameter *-pex*. This parameter causes sscBoogie, after translation, to call the Pex wizard to create a test project including test methods and to call Pex to analyze these generated test methods.

Normally, if an assertion fails during the translation process in sscBoogie, an error message is shown to the user. This behavior is not desired if the translation is started as part of an automated test suite, for which it is known that certain tests fail, because these specific language features are not supported yet. To deactivate the error message GUI and to allow sscBoogie to continue after a failed assertion, the parameter *-disableAssertUI* can be used.

Exceptions (not to be confused with assertion failures) that happen during the translation are catched and instead, an error message is traced. Sometimes, especially during debugging, it is useful to propagate runtime exceptions up to the user. This can be done with the parameter *-throwException*.

Chapter 4

Evaluation

To evaluate the validity of our chosen translation encoding, we wrote a test suite that contains examples for all implemented language features that can be translated. The tests have been tested with Pex and analyzed with Clousot.

In addition to this new test suite, all test drivers for the old existing sscBoogie tests have been extended, so they translate the Spec# test programs into CC++ code. The IL code of these translated programs is then compared with the IL code of the last known good version of the test and the difference is returned. This allows regression testing of the translation process.

4.1 Test Suite

The test suite consist of Spec# classes. Every class tests a specific Spec# language feature. All tests can be translated into CC++ code and can be successfully executed. The translated code behaves the same as in Spec#.

4.2 Runtime Tests

Some test suites have been extended with a test driver that compiles the Spec# test code, translates it to CC++ code and then executes the translated code. The most of the Spec# tutorial examples and all of our newly written tests can be executed and checked at runtime with these test drivers.[1]

4.3 Pex

Pex first needs to generate parametrized test stubs from the translated code, using *Pexwizard.exe*. The generated stubs can be automatically tested with *Pex*. The assembly under test must be rewritten by the ccrewriter before it can be tested.

Pex uses a constraint solver to determine new test inputs for parameters to achieve high code coverage. The test inputs are automatically generated, using the object invariants of the created object, to determine the validity of the generated input.

For most objects, it is necessary to provide factory methods that return instances of more complex classes, for which Pex can not guess how to create new instances. For example, Pex can not create instances of `GhostVariable` objects. To ensure that the factory methods are correct, object factory tests can be used [28].

For most of the tests, Pex outputs a *path bound exceeded* warning. In principal, this issue can be solved by set higher boundaries (i.e., `MaxConditions`, `MaxBranches`). To high boundaries, result in long execution time or stack-overflow exceptions.

[1]The test suites *SscBoogie\Test\CodeContracts* and *SscBoogie\Test\tutorial* have been extended with test drivers that allows the translated programs to be executed.

We experienced that Pex can not generate tests for non-trivial cases. Pex crashes because of to high boundaries, which cause a stack overflow exception. A possible reason for this could be that the ownership rules are defined recursively and Pex tries to achieve complete code coverage. Further analysis has to be done in this area.

The `CCSS` library uses static delegates with static fields, which cause warnings by Pex. Pex considers them not to be deterministic. Many `CCSS` methods, which handle the ownership model, contain `Type` parameters. A known limitation of Pex is, it can not create instances of type `Type`.

4.4 Clousot

Clousot is integrated in the CodeContracts *cccheck* tool. We attempted to verify our test suite with this tool — and failed.

Listing 4.1 shows the arguments we used for testing. These arguments are identical to the arguments the CodeContracts Visual Studio plugin uses to verify code.

Listing 4.1: Clousot Arguments used for Testing

```
cccheck.exe -nobox -nologo -nopex -remote -warninglevel full  -maxwarnings 50
  -nonnull -arrays -wp=true -bounds:type=subpolyhedra, reduction=simplex,
  diseq=false -adaptive -arithmetic -enum -check assumptions -suggest requires
  -repro "-resolvedPaths:<Paths to .Net 4.0> -libPaths:<Paths to .Net 4.0>;
  <Path to CC++ library> <assembly>
```

The results were not as good as expected. Clousot can not prove the tests without warnings. One reason is, that our encoding heavily depends on collections from the .Net standard library, which store the sets used for nearly all checks. The specifications provided by Microsoft for these collections are not strong enough to prove the required properties, as can be seen in Listing 4.2.

Listing 4.2: C#: Clousot Evaluation

```
HashSet<int> set = new HashSet<int>();
set.Add(1);
Contract.Assert(set.Contains(1)); // Warning: assertion unproven
```

How the results, achieved with Clousot, can be improved, is an open point for future work.

Chapter 5

Conclusion

This chapter concludes this thesis. Section 5.1 gives an overview of the current state of the implementation, which features are not supported yet and possible future extensions. Section 5.2 summarizes the achievements.

5.1 State of the Implementation

This section shows the current state of the implementation of the CC++ translation. Due to time constraints, we focused on implementing the core features of Spec#, leaving additional features as future extensions.

5.1.1 Not Implemented Features

The following features are not or only partially implemented in the current version, due to lack of time.

Delayed Constructors/Methods

An object is *delayed* until it is fully initialized, and all non-nullable fields have been assigned. The `[Delayed]` attribute can be used to call methods on such delayed objects. The `[Delayed]` attribute is valid for methods, constructors, properties and parameters. The `[NonDelayed]` attribute is only valid for constructors. Constructors are delayed by default. Non-delayed constructors require that all non-nullable fields of the current class frame are initialized before the base constructor is called [13]. As all non-null checks, this is checked by the sscBoogie type system, and not by Boogie. To encode this behavior in our model, a non-null check for all non-nullable field of the current class frame can be added before the base constructor call.

The `[Delayed]`/`[NonDelayed]` attributes have no influence on the constructors pre- or postconditions. At the end of a constructor, the constructed object has to be consistent and all its invariants have to hold.

A delayed method does not require the method to be consistent or peer-consistent and it is also not ensured. The method has no default pre- and postconditions. Delayed objects can not rely on their non-nullable field to be non null, so at the beginning of a delayed method it must be assumed that non-nullable field can be null.

The ccrewriter does not support contracts that stand in a constructor before the constructor base call. They are simply ignored by the rewriter. In Spec# delayed constructor base calls are supported. In order to implement delayed constructor base calls, there must be a way to rewrite such (in normal C# code invalid) contracts.

An other issue with delayed constructor base calls is, that ghost variables are used before they are initialized in the custom object class constructor.

Visibility-based Object Invariants

Besides ownership-based object invariants, there is another type of object invariants: visibility-based object invariants. Visibility-based object invariants allow to use fields in object invariants of objects that are not owned (that are not decorated with [Rep]) [22]. For instance, they can be used to state properties of peer objects.

Fields that are decorated with `[Dependent(Type)]` can be used in visibility-based object invariants, whereas `Type` is the type of the class where the invariant is defined. The `[Dependent(Type)]` attribute is required to extend the frame condition for field updates with additional checks.

Visibility-based object invariants require to keep track on all objects that use `[Dependent(Type)]` fields in their object invariants. Every time such a field changes, all object that have a reference to the object whose field changed, have to be checked if their object invariants still hold. This is currently not possible in our implementation, but would be a future extension.

Arrays

In handling arrays, several open points exist in the current implementation.

Ownership of Arrays and Array Elements Array is a special type and can not inherit from the custom object class. In the current implementation it is not possible to set the owner of arrays, but it is possible to set the owner of array elements, using the method `Owner.Assign` or `Owner.AssignSame`. The attributes `[ElementsRep]`, `[ElementsPeer]` can not be used to specify array element ownership. Also, the method `Owner.ElementProxy` is not supported.

Arrays in Modifies Clauses The array indexer arr[*] can not be used in modifies clauses. This modifier could be used to specify that all elements of an array are allowed to be modified by a method.

Arrays of Non-Null Elements The marker method `NonNullTypes.AssertInitialized`, indicating that all non-nullable array elements have been initialized, can not be used.

NoDefaultContract

Methods that are decorated with `[NoDefaultContract]` have no default pre- and postconditions. This attribute is currently ignored.

Read Clauses

Pure methods can be decorated with `[Reads(ReadsAttribute.Reads)]` to specify what fields a method is allowed to read. Possible read values are *Owned*, *Nothing* and *Everything*. The implementation of this language feature is similar to the implementation of modifies clauses and can use the CC++ feature `ContractPP.Reads`. To check read clauses, additional checks before every field access are required.

The current implementation of Spec# does not enforce the reads clause within the method body.

Immutable Types

Immutable types are obsolete in Spec#. A more powerful concept of frozen objects [24] for Spec# is currently in development.

Invariants for static fields

According to Leino and Müller [21, 20], static fields can theoretically be used in invariants. This feature is not fully implemented yet in Spec#. Therefore, static fields and static invariants are not considered in the translation. They could be implemented, using an additional static ghost variable in the custom object class, that keeps track of all instances of certain classes that define such static fields.

Recursion Termination Level

In order to prove that pure methods terminate, a recursion termination level is used by Spec#. Recursion termination levels are automatically inferred by Spec# or can be specified using the attribute `[RecursionTermination(r)]`. Recursion termination is directly checked by the Spec# compiler, but the `[RecursionTermination]` attribute is still available in the IL code, even it is ignored by sscBoogie. This attribute is ignored by the current implementation and removed from the translated code.

Model Fields

Model fields can be used in contracts [19]. They can be read independent from the object's state and are updated at the end of an expose. Model field satisfy conditions are part of the object invariants.

Out-of-Band Contracts

Out-of-Band contracts allow to specify contracts for already compiled assemblies or Spec# external code. The contracts are stored in a separate repository.

The pendant of Spec# out-of-band contracts are CodeContracts contract reference assemblies [6]. They contain the public visible interfaces of an assembly along with its contracts.

Checked Exceptions and Exceptional Postconditions

Checked exceptions and exceptional postconditions [25] are not considered in our current implementation, as they are not fully supported by the current implementation of Spec#.

Structs

Structs are not supported, as Structs can not be handled by our current implementation. Structs can not inherit the required ghost variables from the custom object class.

Verify

The `[Verify]` attribute can be used to specify method or classes that do not have to be verified. Classes that are decorated with the `[Verify(false)]` attribute are currently not translated at all. This is due to the sscBoogie design, where the `[Verify]` attribute is checked, before the code generator visitors are called. This check must be moved within the code generator, as the CC++ code generator has a different behavior than the Boogie code generator.

5.1.2 Future Work

This section describes some possible extensions to the current implementation. These extensions are not mandatory in order to use our system, but they improve the usability and the functionality of the current implementation.

Support for Standard Libraries

As mentioned in section 3.3.1, the current implementation of our ownership model depends on a custom object class, which all objects must derive from. This restriction is not useful if classes from standard libraries are used (which are already compiled). A future extension will be to overcome this limitation and provide ownership support for all classes, independent from their inheritance structure.

One solution to this restriction could be, to use a static dictionary which stores the ghost variables for all instances of library classes.

CC++ Rewriter

Currently, the CC++ rewriter is included in the CC++ code generation part of sscBoogie and all runtime checks are written besides their `ContractPP` markers in the code. Normally, sscBoogie should only output code with CC++ markers. At a later stage, the CC++ rewriter should replace them with runtime checks.

In the current implementation, both, the `ContractPP` markers and the resulting runtime-checks exist in the code. Because some CC++ markers throw exceptions if non-wellformed parameters are passed, they can be skipped via the command line parameter *-skipCcPp*.

As a future extension, the runtime check generation out of CC++ markers can be separated from the CC++ code generator and moved to a separate executable.

CC++ markers whose runtime-check generation should be separated: `ContractPP.Writes`, `ContractPP.Reads`, `ContractPP.Fresh`, `GhostVariable`.

Optimization for Pex

Additional test cases are tested with Pex. Interesting examples with good test results are evaluated. The generated CC++ code is optimized for Pex, to achieve better results and to find better test cases. To achieve this, a solution for suitable object creation via factory methods must be found. To overcome the limitation with Pex boundaries (see section 4.3), the translated ownership model must possibly be simplified (without recursive definitions).

Optimization for Clousot

Test cases are analyzed with Clousot. The generated CC++ code is optimized for Clousot, to achieve better results and more meaningful output messages. The issue is, many of the classes in the .Net standard library do not contain specifications, but they are used for the encoded ownership model. Clousot can not verify these parts of the code. Missing specifications must be added to overcome this limitation.

Error Handler

The CC++ translator can save errors that occurred during the translation process with the `System.Compiler.ErrorHandler` component. These errors can later be used to display a message in the commandbox, or show squiggles in the Visual Studio, if the CC++ translation is run as plug-in.

Verifier Feedback Integration

Verification results can be reintegrated into the CC++ code, so that other tools, such as Pex, can use this additional information; e.g., Methods that have been proven incorrect by sscBoogie can be extended with additional assert statements that help finding suitable test cases to produce an error for the statement whose verification failed.

Visual Studio Integration

The CC++ code generation could be started directly from Visual Studio, to provide a better user experience than the command prompt. When building a Spec# project within Visual Studio, the CC++ code generator is run on the resulting assembly. All available options could be configured via a property panel.

5.2 Conclusion

In this paper we presented a way to translate Spec# programs to CC++ programs and implemented the translation rules in the sscBoogie verifier.

The first task was to define the translation rules and to model the Spec# methodology in CC++. The rules are similar to the Boogie encoding and use similar ghost state annotations. The ghost states specification has been adapted from Boogie. For every Boogie assertion, there is one in our encoding. CC++ uses a different heap model than Spec#. Spec# and Boogie reason over the entire heap, whereas CC++ uses a dynamic frames approach and stores owned objects in rep sets. We translated the Spec# ownership model to dynamic frames.

A subset of the translation rules have been encoded in Dafny, as CC++ contains similar language features as Dafny. The prototype has shown that the chosen ownership model encoding with dynamic frames is suitable for verification of programs.

The sscBoogie backend was extended with an additional code generator, which implements the defined translation rules and generates CC++ code. Because, extending sscBoogie was harder than expected and Spec# has numerous features, a subset of all Spec# language features was implemented.

The resulting code for some translated examples was evaluated with Pex and Clousot. The evaluation showed that the current translation rules are too complex to be handled by Pex and Clousot. Further work has to be done to simplify the encoding.

Summing up the project, with the current implementation it is possible to translate simple examples, such as the tutorial examples, from Spec# to CC++ and to execute the translated programs. In order to translate real code, as sscBoogie itself, further work must be done, and more language features have to be supported.

Appendix A

Encountered Difficulties

This chapter shows some of the difficulties encountered during this thesis. The difficulties are mainly of technical nature, caused by the limitations of the technologies used.

A.1 sscBoogie

This section contains difficulties encountered while adding the CC++ code generator to the sscBoogie verifier framework.

A.1.1 Normalizer

The Normalizer can only run on expressions that have not already been normalized. An assembly that is read from the filesystem is normalized. Newly created expressions that are added during the translation process are not normalized. These newly created expressions must be normalized before they can be added to the assembly. The normalizer replaces complex language features with simple language features that can be represented by IL statements.

Some expressions that are not normalized can not be written by the CCI Writer (e.g., *TypeOf*, *LogicalAnd*, *LogicalNot*, *LogicalNot*).

Calling the Normalizer is not idempotent. If the Normalizer visits an already normalized method, in certain cases, it will change the expression to invalid IL code.

A.1.2 Recover Serialized Contracts

Serialized contracts, as used in *assert* and *loop invariants*, are recovered with the `ContractDeserializer.ParseContract` method. The issue is, this function interprets variables in the contract as class fields and ignores local variables. After recovering the contract the variables in the contract must be checked and, if a corresponding local variable with the same name exists, replaced.

A.1.3 AST Node Equality

In the Spec# compiler, the AST nodes do not contain an `Equals` method. Thus, in sscBoogie, objects are considered different even if they refer to the same node. This causes some issues during translation and requires some additional checks that otherwise could be skipped.

A.1.4 sscBoogie and Boogie

Currently, the Boogie and sscBoogie solutions are not compatible. The command line arguments/options used by sscBoogie are part of Boogie, so they can currently not be extended with options used by the CC++ code generator. As long as these options can not be added, ssc can not run sscBoogie as plugin and pass the required arguments to sscBoogie to run the CC++ code generator.

When the option handling of sscBoogie switches to the options class in the sscBoogie solution, then sscBoogie can be run as a plugin of ssc.

A.2 CodeContracts

This section contains difficulties encountered with the current release of CodeContracts. The issues occurred in the ccrewriter.

A.2.1 Higher Level Language Features in Contract.OldValue()

CodeContracts has issues with binary operators, Lambda expressions and Delegates in `Contract.OldValue()` statements. If these expressions are used in `OldValue()` statements. CodeContracts rewriting fails, and during runtime the exception "Common Language Runtime detected an invalid program" is shown.

Listing A.1: C#: Higher Level Language Features in Contract.OldValue() Statements

```
// Crashes
Contract.Ensures(Contract.OldValue(a || b));

// Works
Contract.Ensures(Contract.OldValue(a) || Contract.OldValue(b));
```

Ternary expressions (e.g., `bool ? val1 : val2;`) are not allowed either.

This bug has already been reported, but it has not been fixed yet: http://social.msdn.microsoft.com/Forums/eu/codecontracts/thread/11fc463f-9dd2-4037-b2d8-5c1eceed4513

In this thesis we found a workaround for this issue by wrapping the failing statements inside a method.

A.2.2 Contract.OldValue() Deep Copy

The statement `Contract.OldValue()` creates a copy of the specified object reference and does not do a deep copy. This means if an original set has to be kept for later use in the postcondition, it has to be copied into a new set that can not be modified outside the `OldValue()` statement.

Listing A.2: C#: Keep Original Set Example

```
Contract.OldValue(new HashSet<CCObject>(~o._rep));
```

A.2.3 Contracts before Constructor Base Call

Issues arise if contracts occur within a constructor before the base constructor call.

Before Rewriting

The ccrewrite tool ignores contracts that occur before the base constructor call within a constructor. But, such contracts are required by CC++ to handle delayed base constructor calls.

After Rewriting

CodeContracts uses a pragmatic rewriting approach. During rewriting, the references to old values are assigned as first operations in a method. This also happens in constructors where no variables have been allocated yet. So it must be considered that the old values are null, even if they should not be.

A.2.4 Overridden Methods with TypeOf Operator in Contracts

The Spec# tutorial example *Sec2.2-Additive* [23] contains an overridden method `AddMoney`. This method is decorated with an `[Additive]` attribute, which requires the method to be additively exposable. This attribute causes the method contracts to contain a `typeof` operator. Because the method is overridden, the contracts require different `typeof` calls for each class frame. In the example, there is a `typeof(Purse)` statement in the base class and `typeof(SwissPurse)` statement in the derived class.

The `[Additive]` attribute requires different preconditions for each class frame of overridden methods. But overridden methods are not allowed to contains additional preconditions. They are rejected by the ccrewriter. The Spec# methodology is sound, but in the current version of CodeContracts this can not be implemented.

It is still an open issue how to add additional preconditions to overridden methods. Overridden methods with `[Additive]` attributes can not be handled by the current implementation.

A.3 Pex

Pex has issues handling certain main methods. The main method arguments are not allowed to be named ”args”, because this name is used by Pex in the automatically generated test stubs.

Error message: *error CS0136: A local variable named 'args' can not be declared in this scope because it would give a different meaning to 'args', which is already used in a 'parent or current' scope to denote something else*

As a workaround, the main method argument that is generated by default by the Visual Studio template must be renamed.

Appendix B

Uncovered Bugs

This chapter contains the bugs we uncovered in sscBoogie and Boogie during this thesis.

B.1 sscBoogie

B.1.1 Quantifiers

sscBoogie contained a bug which created wrong Boogie code for quantifiers over collections. Boogie was not able to prove the generated code. This bug has been fixed by Valentin Wüstholz in change set 5bcd877b913d.

B.2 Boogie

B.2.1 Lambda Expressions

Boogie crashed for lambda expressions which contained `old` statements. The `old` statements were not replaced after lambda expansion. This bug has been fixed by Valentin Wüstholz in change set c4cc0e2cf4ae.

Appendix C

Translation Rules Prototyping in Dafny

Dafny was used to prototype a subset of the CC++ translation rules, as Dafny supports similar features as CC++. The prototypes raised our confidence in the translation rules.

In our translation encoding, we depend on a custom object class, that contains all ghost variables and from which all classes must derive from.

Dafny does not support inheritance by itself, except that all classes are implicitly direct subclasses of the base class `object`, which does not contain any members. In order to translate the Spec# ownership model to Dafny, the Dafny compiler had to be modified to allow custom object base classes.

With the modified version of Dafny, it is possible to declare a custom object class in the code and extend it with fields and methods. These members can then be used by all other classes (that inherit these members from the object class). The name of the class that should be used as object class can be specified via the Dafny command line parameter *-customobjectclass:<class_name>*. Note that `object` can not be used as class name, because it is a keyword in Dafny. It is recommended to use `Object` as class name instead. If no custom object class is defined in the code, the built in object class is used by the compiler.

Listing C.1 shows a custom object class `Object`. Class `A` inherits from this custom object and has access to the field `_foo` of the custom object class. This was not possible in the original Dafny implementation.

Listing C.1: Dafny: Custom Object Class

```
class Object{
  var _foo : int;
}

class A{
  method Bar(){
    this._foo := 1;
  }
}
```

Appendix D

Dictionary Example

Listing D.1 shows the complete CC++ code (before rewriting) of the Dictionary example used in section 2.2.

Listing D.1: CC++: Dictionary Example

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Contracts;
using CodeContractsForSpecSharp;
using CCPlusPlus;

namespace CodeContractsForSpecSharp.Examples
{

  /// <summary>
  /// CC++ encoding of the Fig11-Dictionary example in the Spec# tutorial.
  /// </summary>
  public class Dictionary : CCObject
  {
    /*[Rep]*/
    private Node head;
    /*[Rep]*/
    private Node tail;

    public Dictionary()
    {
      // Constructor contracts
      Contract.Ensures(this.SS_IsPartiallyConsistent(typeof(Dictionary)),
        "This must be partially consistent");
      Contract.Ensures(ContractPP.Fresh(this.GetAllPeerAndRepObjects().Except(
        Contract.OldValue<ISet<CCObject>>(this.GetAllPeerAndRepObjects()))),
        "Fresh clause violated");
      ContractPP.Writes(this.GetRep());

      // GhostVariable initialization
      GhostVariable<ISet<CCObject>> SS_modifiableObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Modifiable_Objects");
      GhostVariable<ISet<IField<CCObject>>> SS_modifiableFields =
        new GhostVariable<ISet<IField<CCObject>>>(new HashSet<IField<CCObject>>(),
          "Modifiable_Fields");
      GhostVariable<ISet<CCObject>> SS_fresh =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Fresh_Objects");
      GhostVariable<ISet<CCObject>> SS_preStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PreState_Objects");
      (~SS_modifiableObjects).AddRange(this.GetRep());
      (~SS_preStateObjects).AddRange(this.GetAllPeerAndRepObjects());

      // this.head update - frame condition
      Contract.Assert(this != null, "Fieldowner of field 'head' is null.");
      Contract.Assert(this.IsWritable(), "Fieldowner of field 'head' is not writable");
      GhostVariable<IField<CCObject>> SS_gf0 =
        new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this, "Dictionary.head"),
          "Assigned_Field");
      Contract.Assert(CCSS.IsFieldModifiable((IField<CCObject>)~SS_gf0, ~SS_modifiableFields,
        ~SS_modifiableObjects, ~SS_fresh), "Field 'head' is not modifiable.");
      this.head = null;
      this.head.UpdateOwnersForRep(this, typeof(Dictionary));
      Contract.Assert(this.IsMutable(typeof(Dictionary)) || this.SS_Inv(),
        "Fieldowner of field 'head' is not mutable or invariant does not hold.");

      // this.tail update - frame condition
      Contract.Assert(this != null, "Fieldowner of field 'tail' is null.");
      Contract.Assert(this.IsWritable(), "Fieldowner of field 'tail' is not writable");
      GhostVariable<IField<CCObject>> SS_gf1 =
        new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this, "Dictionary.tail"),
          "Assigned_Field");
      Contract.Assert(CCSS.IsFieldModifiable((IField<CCObject>)~SS_gf1, ~SS_modifiableFields,
        ~SS_modifiableObjects, ~SS_fresh), "Field 'tail' is not modifiable.");
      this.tail = null;
      this.tail.UpdateOwnersForRep(this, typeof(Dictionary));
      Contract.Assert(this.IsMutable(typeof(Dictionary)) || this.SS_Inv(),
        "Fieldowner of field 'tail' is not mutable or invariant does not hold.");

      // Establish postcondition
      Contract.Assert(this.SS_Inv(), "Object invariant of object 'Dictionary this' does not
          hold.");
      this.AdditivePack(typeof(Dictionary));

      // Fresh check
      GhostVariable<ISet<CCObject>> SS_postStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PostState_Objects");
      (~SS_postStateObjects).AddRange(this.GetAllPeerAndRepObjects());
      Contract.Assert((~SS_postStateObjects).Except(~SS_preStateObjects)
        .IsSubsetOf(~SS_fresh), "Fresh clause violated.");
    }

    [Pure]
    public bool Find(string key, out int val)
    {
      // Method contracts
      Contract.Requires(this.IsPeerValid(), "This must be peer valid");
      Contract.Ensures(this.IsPeerValid(), "This must be peer valid");
      bool returnValue;
      Node n = this.head;
      while (true)
      {
        if (n == null)
        {
          break;
        }

        if (n.key == key)
        {
          val = n.val;
          returnValue = true;
          goto Label_00FA;
        }
        n = n.next;
      }
      val = 0;
      returnValue = false;
    Label_00FA:
      return returnValue;
    }

    public void Insert(string key, int val)
    {
      // Method contracts
      Contract.Requires(this.SS_IsConsistent(), "This must be consistent");
      Contract.Requires(this.IsPeerConsistent(), "This must be peer consistent");
      Contract.Ensures(this.SS_IsConsistent(), "This must be consistent");
      Contract.Ensures(this.IsPeerConsistent(), "This must be peer consistent");
      Contract.Ensures(ContractPP.Fresh(this.GetAllPeerAndRepObjects().Except(
        Contract.OldValue<ISet<CCObject>>(this.GetAllPeerAndRepObjects()))),
        "Fresh clause violated");
      ContractPP.Writes(this.GetOwnedObjects());
      ContractPP.Writes<CCObject>(new IField<CCObject>[] {
        Field<CCObject>.Create(this, "Dictionary.head"),
        Field<CCObject>.Create(this, "Dictionary.tail")
      });
      ContractPP.Writes(this.GetOwnedObjects());

      // GhostVariable initialization
      GhostVariable<ISet<CCObject>> SS_modifiableObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Modifiable_Objects");
      GhostVariable<ISet<IField<CCObject>>> SS_modifiableFields =
        new GhostVariable<ISet<IField<CCObject>>>(new HashSet<IField<CCObject>>(),
          "Modifiable_Fields");
      GhostVariable<ISet<CCObject>> SS_fresh =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Fresh_Objects");
      GhostVariable<ISet<CCObject>> SS_preStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PreState_Objects");
      (~SS_modifiableObjects).AddRange(this.GetOwnedObjects());
      (~SS_modifiableFields).Add(Field<CCObject>.Create(this, "Dictionary.head"));
      (~SS_modifiableFields).Add(Field<CCObject>.Create(this, "Dictionary.tail"));
      (~SS_preStateObjects).AddRange(this.GetAllPeerAndRepObjects());

      // Local unpack
      Contract.Assert(CCSS.IsObjectModifiable(this, ~SS_modifiableFields,
        ~SS_modifiableObjects, ~SS_fresh), "Object 'Dictionary this' is not modifiable.");
      this.Unpack(typeof(Dictionary));

      Node h = this.head;

      // this.head update - frame condition
      Contract.Assert(this != null, "Fieldowner of field 'head' is null.");
      Contract.Assert(this.IsWritable(), "Fieldowner of field 'head' is not writable");
      GhostVariable<IField<CCObject>> SS_gf0 =
        new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this, "Dictionary.head"),
          "Assigned_Field");
      Contract.Assert(CCSS.IsFieldModifiable(~SS_gf0, ~SS_modifiableFields,
        ~SS_modifiableObjects, ~SS_fresh), "Field 'head' is not modifiable.");
      Node SS_nodeConstructorResult = new Node(key, val);
      ((ISet<CCObject>)~SS_fresh).AddRange(SS_nodeConstructorResult.GetRep());
      this.head = SS_nodeConstructorResult;
      this.head.UpdateOwnersForRep(this, typeof(Dictionary));
      Contract.Assert(this.IsMutable(typeof(Dictionary)) || this.SS_Inv(),
        "Fieldowner of field 'head' is not mutable or invariant does not hold.");

      // this.head.next update - frame condition
      Contract.Assert(this.head != null, "Fieldowner of field 'next' is null.");
      Contract.Assert(this.head.IsWritable(), "Fieldowner of field 'next' is not writable");
      GhostVariable<IField<CCObject>> SS_gf2 =
        new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this.head, "Node.next"),
          "Assigned_Field");
      Contract.Assert(CCSS.IsFieldModifiable(~SS_gf2, ~SS_modifiableFields,
        ~SS_modifiableObjects, ~SS_fresh), "Field 'next' is not modifiable.");
      this.head.next = h;
      this.head.next.UpdateOwnersForPeer(this.head);
      Contract.Assert(this.head.IsMutable(typeof(Node)) || this.head.SS_Inv(),
        "Fieldowner of field 'next' is not mutable or invariant does not hold.");

      if (this.tail == null)
      {
        // this.tail update - frame condition
        Contract.Assert(this != null, "Fieldowner of field 'tail' is null.");
        Contract.Assert(this.IsWritable(), "Fieldowner of field 'tail' is not writable");
        GhostVariable<IField<CCObject>> SS_gf3 =
          new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this, "Dictionary.tail"),
            "Assigned_Field");
        Contract.Assert(CCSS.IsFieldModifiable(~SS_gf3, ~SS_modifiableFields,
          ~SS_modifiableObjects, ~SS_fresh), "Field 'tail' is not modifiable.");
        this.tail = this.head;
        this.tail.UpdateOwnersForRep(this, typeof(Dictionary));
        Contract.Assert(this.IsMutable(typeof(Dictionary)) || this.SS_Inv(),
          "Fieldowner of field 'tail' is not mutable or invariant does not hold.");
      }

      // Local pack
      Contract.Assert(this.SS_Inv(), "Object invariant of object 'Dictionary this' does not
           hold.");
      this.Pack(typeof(Dictionary));

      // Fresh check
      GhostVariable<ISet<CCObject>> SS_postStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PostState_Objects");
      ((ISet<CCObject>)~SS_postStateObjects).AddRange(this.GetAllPeerAndRepObjects());
      Contract.Assert((~SS_postStateObjects).Except((~SS_preStateObjects)).
        IsSubsetOf(~SS_fresh), "Fresh clause violated.");
    }

    public static void InsertTest(Dictionary d)
    {
      // Method contracts
      Contract.Requires((d != null) && d.SS_IsConsistent(),
        "Object 'Dictionary d' must be non-null and consistent");
      Contract.Requires((d != null) && d.IsPeerConsistent(),
        "Object 'Dictionary d' must be non-null and peer consistent");
      Contract.Ensures((d != null) && d.SS_IsConsistent(),
        "Object 'Dictionary d' must be non-null and consistent");
      Contract.Ensures((d != null) && d.IsPeerConsistent(),
        "Object 'Dictionary d' must be non-null and peer consistent");
      Contract.Ensures(ContractPP.Fresh(d.GetAllPeerAndRepObjects().Except(
        Contract.OldValue<ISet<CCObject>>(d.GetAllPeerAndRepObjects()))),
        "Fresh clause violated");
      ContractPP.Writes(d.GetRep());

      // GhostVariable initialization
      GhostVariable<ISet<CCObject>> SS_modifiableObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Modifiable_Objects");
      GhostVariable<ISet<IField<CCObject>>> SS_modifiableFields =
        new GhostVariable<ISet<IField<CCObject>>>(new HashSet<IField<CCObject>>(),
          "Modifiable_Fields");
      GhostVariable<ISet<CCObject>> SS_fresh =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Fresh_Objects");
      GhostVariable<ISet<CCObject>> SS_preStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PreState_Objects");
      Dictionary SS_DictionaryOldReference = null;
      if (d != null)
      {
        ((ISet<CCObject>)~SS_modifiableObjects).AddRange(d.GetRep());
        ((ISet<CCObject>)~SS_preStateObjects).AddRange(d.GetAllPeerAndRepObjects());
        SS_DictionaryOldReference = d;
      }

      // Dictionary.Insert method call - frame condition
      GhostVariable<ISet<CCObject>> SS_methodModifiableObjects_Insert =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(),
          "Method_Insert_Modifiable_Objects");
      GhostVariable<ISet<IField<CCObject>>> SS_methodModifiableFields_Insert =
        new GhostVariable<ISet<IField<CCObject>>>(new HashSet<IField<CCObject>>(),
          "Method_Insert_Modifiable_Fields");
      GhostVariable<ISet<CCObject>> SS_methodPreStateObjects_Insert =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(),
          "Method_Insert_PreState_Objects");
      GhostVariable<ISet<CCObject>> SS_methodPostStateObjects_Insert =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(),
          "Method_Insert_PostState_Objects");
      if (d != null)
      {
        (~SS_methodModifiableObjects_Insert).AddRange(d.GetOwnedObjects());
        (~SS_methodModifiableFields_Insert).Add(Field<CCObject>.Create(d, "Dictionary.head"));
        (~SS_methodModifiableFields_Insert).Add(Field<CCObject>.Create(d, "Dictionary.tail"));
        (~SS_methodPreStateObjects_Insert).AddRange(d.GetRep());
      }
      Contract.Assert(CCSS.IsModifiesClauseSatisfied(~SS_methodModifiableFields_Insert,
        ~SS_methodModifiableObjects_Insert, ~SS_modifiableFields, ~SS_modifiableObjects,
        ~SS_fresh), "Method invocation of method 'Dictionary.Insert(String,Int32)' violates the
             modifies clause of the enclosing method");
      d.Insert("foo", 123);
      if (d != null)
      {
        (~SS_methodPostStateObjects_Insert).AddRange(d.GetRep());
      }
      (~SS_fresh).AddRange((~SS_methodPostStateObjects_Insert).Except(
        ~SS_methodPreStateObjects_Insert));

      // Fresh check
      GhostVariable<ISet<CCObject>> SS_postStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PostState_Objects");
      if (SS_DictionaryOldReference != null)
      {
        ((ISet<CCObject>)~SS_postStateObjects).AddRange(
          SS_DictionaryOldReference.GetAllPeerAndRepObjects());
      }
      Contract.Assert((~SS_postStateObjects).Except(~SS_preStateObjects).
        IsSubsetOf(~SS_fresh), "Fresh clause violated.");
    }

    [Pure]
    public new bool SS_Inv()
    {
      return this.Always()
        && ((this.head == null && this.tail == null) || (this.head != null && this.tail != null
            ))
        && CCSS.NullOrOwnerIs(this.head, this, typeof(Dictionary))
        && CCSS.NullOrOwnerIs(this.tail, this, typeof(Dictionary));
    }

    [ContractInvariantMethod]
    private void SS_InvCheck()
    {
      Contract.Invariant(this.SS_Inv());
    }

    [Pure]
    public override bool SS_IsConsistent()
    {
      return (this.SS_Inv() && base.SS_IsConsistent());
    }

    [Pure]
    public new bool SS_IsPartiallyConsistent(Type frame)
    {
      return (this.SS_Inv() && base.SS_IsPartiallyConsistent(frame));
    }
}


internal class Node : CCObject
{
  public string key;
  /*[Peer]*/
  public Node next;
  public int val;

  public Node(string key, int val)
  {
    // Contructor contracts
    Contract.Ensures(this.SS_IsPartiallyConsistent(typeof(Node)),
      "This must be partially consistent");
    Contract.Ensures(ContractPP.Fresh(this.GetAllPeerAndRepObjects().Except(
      Contract.OldValue<ISet<CCObject>>(this.GetAllPeerAndRepObjects()))),
      "Fresh clause violated");
    ContractPP.Writes(this.GetRep());

    // GhostVariable initialization
    GhostVariable<ISet<CCObject>> SS_modifiableObjects =
      new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Modifiable_Objects");
    GhostVariable<ISet<IField<CCObject>>> SS_modifiableFields =
      new GhostVariable<ISet<IField<CCObject>>>(new HashSet<IField<CCObject>>(),
        "Modifiable_Fields");
    GhostVariable<ISet<CCObject>> SS_fresh =
      new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "Fresh_Objects");
    GhostVariable<ISet<CCObject>> SS_preStateObjects =
      new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PreState_Objects");
    (~SS_modifiableObjects).AddRange(this.GetRep());
    (~SS_preStateObjects).AddRange(this.GetAllPeerAndRepObjects());

    // this.key update - frame condition
    Contract.Assert(this != null, "Fieldowner of field 'key' is null.");
    Contract.Assert(this.IsWritable(), "Fieldowner of field 'key' is not writable");
    GhostVariable<IField<CCObject>> SS_gf0 =
      new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this, "Node.key"),
        "Assigned_Field");
    Contract.Assert(CCSS.IsFieldModifiable(~SS_gf0, ~SS_modifiableFields,
      ~SS_modifiableObjects, (ISet<CCObject>)~SS_fresh), "Field 'key' is not modifiable.");
    this.key = key;
    Contract.Assert(this.IsMutable(typeof(Node)) || this.SS_Inv(),
      "Fieldowner of field 'key' is not mutable or invariant does not hold.");

    // this.val update - frame condition
    Contract.Assert(this != null, "Fieldowner of field 'val' is null.");
    Contract.Assert(this.IsWritable(), "Fieldowner of field 'val' is not writable");
    GhostVariable<IField<CCObject>> SS_gf1 =
      new GhostVariable<IField<CCObject>>(Field<CCObject>.Create(this, "Node.val"),
        "Assigned_Field");
    Contract.Assert(CCSS.IsFieldModifiable(~SS_gf1, ~SS_modifiableFields,
      ~SS_modifiableObjects, (ISet<CCObject>)~SS_fresh), "Field 'val' is not modifiable.");
    this.val = val;
    Contract.Assert(this.IsMutable(typeof(Node)) || this.SS_Inv(),
      "Fieldowner of field 'val' is not mutable or invariant does not hold.");

    // Establish postcondition
    Contract.Assert(this.SS_Inv(), "Object invariant of object 'Node this' does not hold.");
    this.AdditivePack(typeof(Node));
```

```
      // Fresh check
      GhostVariable<ISet<CCObject>> SS_postStateObjects =
        new GhostVariable<ISet<CCObject>>(new HashSet<CCObject>(), "PostState_Objects");
      (~SS_postStateObjects).AddRange(this.GetAllPeerAndRepObjects());
      Contract.Assert((~SS_postStateObjects).Except((~SS_preStateObjects)).
        IsSubsetOf(~SS_fresh), "Fresh clause violated.");
    }

    [Pure]
    public new bool SS_Inv()
    {
      return this.Always() && CCSS.NullOrOwnerSame(this.next, this);
    }

    [ContractInvariantMethod]
    private void SS_InvCheck()
    {
      Contract.Invariant(this.SS_Inv());
    }

    [Pure]
    public override bool SS_IsConsistent()
    {
      return (this.SS_Inv() && base.SS_IsConsistent());
    }

    [Pure]
    public new bool SS_IsPartiallyConsistent(Type frame)
    {
      return (this.SS_Inv() && base.SS_IsPartiallyConsistent(frame));
    }
  }

}
```

Listings

List of Figures

List of Tables

Bibliography

[1] *An Interview with the Microsoft Code Contracts Team.* URL: http://devjourney.com/blog/code-contracts-part-8-an-interview-with-the-microsoft-code-contracts-team/.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. "Verification of Object-Oriented Programs with Invariants". In: *Journal of Object Technology* 3 (2004), p. 2004.

[3] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A modular reusable verifier for object-oriented programs". In: *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science.* Springer-Verlag, 2006, pp. 364–387.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# Programming System: An Overview". In: *Construction and Analysis of Safe, Secure and Interoperable Smart devices.* Vol. 3362. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 49–69.

[5] *Boogie.* URL: http://boogie.codeplex.com/.

[6] *Code Contracts User Manual.* Microsoft Corporation. 2012.

[7] *CodeContracts.* URL: http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

[8] *Dafny.* URL: http://research.microsoft.com/en-us/projects/dafny/.

[9] Ádám Darvas and K. Rustan M. Leino. "Practical reasoning about invocations and implementations of pure methods". In: *Proceedings of the 10th international conference on Fundamental approaches to software engineering.* FASE '07. Braga, Portugal: Springer-Verlag, 2007, pp. 336–351.

[10] Krishna Kishore Dhara and Gary T. Leavens. "Forcing behavioral subtyping through specification inheritance". In: *Proceedings of the 18th international conference on Software engineering.* ICSE '96. Berlin, Germany: IEEE Computer Society, 1996, pp. 258–267.

[11] Manuel Fähndrich. "Static verification for code contracts". In: *Proceedings of the 17th international conference on Static analysis.* SAS '10. Perpignan, France: Springer-Verlag, 2010, pp. 2–5.

[12] Manuel Fähndrich and Francesco Logozzo. "Static contract checking with abstract interpretation". In: *Proceedings of the 2010 international conference on Formal verification of object-oriented software.* FoVeOOS '10. Paris, France: Springer-Verlag, 2011, pp. 10–30.

[13] Manuel Fähndrich and Songtao Xia. "Establishing object invariants with delayed types". In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications.* OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 337–350.

[14] Nicu G. Fruja and Peter Müller. *Translating Invariant Proofs between Spec# and JML.* Tech. rep. ETH Zurich, 2007.

[15] Ioannis T. Kassios. *Dynamic Frames and Automated Verification.* Tutorial for the 2nd COST Action IC0701 Training School, Limerick 6/11, Ireland. 2011.

[16] Ioannis T. Kassios. “The Dynamic Frames Theory”. In: *Formal Aspects of Computing* 23.3 (2011), pp. 267–289.

[17] K. Rustan M. Leino. “Specification and verification of object-oriented software”. In: *Marktoberdorf International Summer School*. Lecture Notes. 2008.

[18] K. Rustan M. Leino. “This is Boogie 2”. In: *Manuscript KRML 178*. 2008.

[19] K. Rustan M. Leino and Peter Müller. “A verification methodology for model fields”. In: *European Symposium on Programming (ESOP)*. Ed. by P. Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer-Verlag, 2006, pp. 115–130.

[20] K. Rustan M. Leino and Peter Müller. *Modular verification of global module invariants in object-oriented programs*. Tech. rep. 459. ETH Zurich, 2004.

[21] K. Rustan M. Leino and Peter Müller. “Modular verification of static class invariants”. In: *Formal Methods (FM)*. Ed. by J. Fitzgerald, I. Hayes, and A. Tarlecki. Vol. 3582. Lecture Notes in Computer Science. Springer-Verlag, 2005, pp. 26–42.

[22] K. Rustan M. Leino and Peter Müller. “Object Invariants in Dynamic Contexts”. In: *European Conference on Object-Oriented Programming (ECOOP)*. Ed. by M. Odersky. Vol. 3086. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 491–516.

[23] K. Rustan M. Leino and Peter Müller. “Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs”. In: *Advanced Lectures on Software Engineering—LASER Summer School 2007/2008*. Ed. by Peter Müller. Vol. 6029. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 91–139.

[24] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. “Flexible Immutability with Frozen Objects”. In: *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*. VSTTE ’08. Toronto, Canada: Springer-Verlag, 2008, pp. 192–208.

[25] K. Rustan M. Leino and Wolfram Schulte. “Exception Safety for C#”. In: *IEEE International Conference on Software Engineering and Formal Methods* (2004), pp. 218–227.

[26] *Modopt, method signatures, and incomplete specs oh my!* URL: http://codebetter.com/gregyoung/2007/06/08/modopt-method-signatures-and-incomplete-specs/.

[27] *Moles*. URL: http://research.microsoft.com/en-us/projects/moles/.

[28] *Parameterized Test Patterns for Microsoft Pex*. Microsoft Corporation. 2010.

[29] *Spec#*. URL: http://specsharp.codeplex.com/.

[30] Nikolai Tillmann and Jonathan de Halleux. “Pex - White Box Test Generation for .NET”. In: *Tests and Proofs*. Ed. by Bernhard Beckert and Reiner Hähnle. Vol. 4966. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 134–153.

[31] *Z3 Theorem Prover*. URL: http://research.microsoft.com/en-us/um/redmond/projects/z3/.