VERIFICATION AND REFINEMENT
THEORY OF ACTION INHERITANCE

Verification and Refinement Theory of Action Inheritance for Concurrent Objects

By
Upasana Pujari, M.C.A., B.Sc.(Hons.)

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Computer Science

McMaster University

MASTER OF COMPUTER SCIENCE (2009) McMaster University
(Department of Computing and Software) Hamilton, Ontario

TITLE: Verification and Refinement Theory of Action Inheritance for Concurrent Objects

AUTHOR: Upasana Pujari, M.C.A. (N.I.T., Rourkela, India), B.Sc.(Hons.) (Utkal University, India)

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: vii, 152

Abstract

Lime is an action-based concurrent object-oriented programming language. Lime treats concurrency and object-orientation as a single concern and encapsulates concurrent features within objects. In Lime objects, concurrency is expressed with guarded methods and actions. Inheritance is a characteristic feature of object-oriented programming languages. Lime supports inheritance of methods.

In this thesis we extend class inheritance in Lime to include inheritance of actions. This ensures that autonomous behavior of the class is also inherited. Class inheritance also aids in verification and refinement of classes. We establish class refinement rules for class inheritance. When these rules are satisfied, the subclass is a subtype as well as a refinement of the parent class. Lime uses modules as a means to define classes in terms of action systems. In our research, we extend the modules to support class inheritance. In this extended form, class modularization is useful for verification and class refinement.

Concurrent object-oriented programming languages are affected by the Inheritance Anomaly – a conflict between inheritance and concurrency. We show that Lime's support for atomicity of methods and actions up to method calls allows our model of classes' inheritance to avoid the problem of the Inheritance Anomaly.

Acknowledgements

I am very grateful to my advisor, Dr. Emil Sekerinski, for giving me this opportunity to work on the area that greatly interests me. I sincerely thank Dr. Sekerinski for his thoughtful guidance, constructive criticism and constant support throughout my study.

Many thanks to Dr. Ridha Khedri and Dr. Ryszard Janicki for examining the thesis and for providing insightful comments for further improvement of the thesis.

I would also like to thank my family for their long time support for my studies.

Contents

Chapter 1

Introduction

Inheritance is a key feature of object-oriented programming languages. It allows us to define new classes using existing classes. Each new class, also known as *child class*, can inherit both data and behavior of the existing class, also known as *parent class*. The child class can also extend the data and behavior of the parent class by adding its own data and behavior elements. Inheritance lets us establish a parent-child hierarchical relationship between classes. Since inheritance allows code reuse in the subclasses, it can be a means to faster program development and more reliable programs.

Inheritance lends itself to iterative form of program development. In each iteration, existing classes can be extended to add new functionality. Stepwise refinement, developed in [31], is an approach to program development in which a sequence of refinement steps is applied to the initial abstract specification, transforming it to the final concrete implementation. The program at each of the intermediate steps is more concrete than at the previous step. Each refinement step brings the program closer to its final implementation by adding new functionality while preserving the behavior of the program from the previous step. When inheritance is constrained by Liskov and Wing's requirement for behavioral notion of subtyping, the child class is assured to preserve the behavior of the parent class. When inheritance is restricted to a correctness-preserving form, then it can be applied as a refinement step in program development.

Liskov and Wing's requirement for constraining the behavior of subtypes [19] is stated as:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

In a concurrent system several processes can execute at the same time. These processes can execute on different processors, potentially leading to faster execution of the program. Concurrency also allows programs to be more responsive by reducing waiting time. Since processes in a concurrent system share resources and interact with each other, the behavior of a concurrent system can be quite complex and difficult to replicate.

Lime is an action-based concurrent object-oriented programming language developed in [27, 28]. *Lime* is based on the concepts of object-orientation and action-system model of concurrency. In an object-oriented programming language, objects are self-contained units that evolve independently of each other. Each object has its own state space. In principle, all objects can be concurrent as long as they do not refer to any global variables in common. In *Lime*, both concurrent and object-oriented features are encapsulated within objects. Therefore, in *Lime* an object is considered as a unit of concurrency [27]. Benefitting from its concurrent and object-oriented features, *Lime* presents class structure and process structure as a single design view. *Lime* does not treat concurrency as a separate concern from objects and therefore simplifies the design of concurrent object-oriented programs. *Lime*'s model of concurrent object-oriented programming allows concurrency to be treated as an implementation issue that can be introduced in subclasses.

Concurrency in *Lime* classes is expressed in terms of actions that execute autonomously. An action is a guarded command of the form $A = g \rightarrow S$, where g is a predicate known as the *guard* of the action and S is the *body* of the action. In *Lime*, condition synchronization is achieved through method and action guards. A *Lime* program can have active methods and actions in several objects that can execute in parallel thus achieving intra-object concurrency. *Lime* supports inheritance and behavior specifications in classes [18]. Classes in *Lime* have two kinds of operations — methods and actions. During class inheritance, methods can be inherited and overridden, and new actions can be introduced in the subclass.

Unlike semaphores, monitors or condition variables, which lock an object until an operation completes execution, *Lime* offers a finer-grained model of concurrency by including the following three features — (i) atomicity of operations upto method calls, (ii) allowing the operations to be enabled multiple times and (iii) having several operations enabled at the same time while only one progresses.

Inheritance Anomaly is a conflict between inheritance and concurrent features of Concurrent Object-Oriented Programming languages. Matsuoka and Yonezawa first coined the term *Inheritance Anomaly* in [22]. When inheritance and concurrency are used together, it leads to a breakage of encapsulation — this is the phenomenon known as *Inheritance Anomaly.* The core of the problem is the difficulty in inheriting behavior from a class that combines behavioral and synchronization code in its definition. Several researchers have presented different solutions to this problem. Some of these solutions are discussed in the next chapter.

The goal of this thesis is to extend *Lime* programming language by adding inheritance of actions. In our research we extend class inheritance in *Lime* to include inheritance of actions. Our design of class inheritance also allows introduction of new behavior in terms of new methods in subclasses. We also define the visibility rules for actions. We update the way *Lime* classes are organized into modules and extend the module syntax to support inherited actions, invariants and visibility specifiers. We also specify the class refinement rules for class inheritance. These rules are justified with respect to superposition refinement of action systems. The class refinement rules limit class inheritance with actions to a correctness preserving form. In this constrained from, class inheritance can be viewed as a refinement of concurrent classes. Class inheritance as class refinement is illustrated with formal proof in an example of a sum of number series. We also present class inheritance with actions together with *Lime*'s class structure as a solution to the known manifestations of Inheritance Anomaly.

Chapter 2 provides a brief introduction to inheritance of type-bound actions in Action-Oberon. This chapter also provides an overview of Inheritance Anomaly and gives the existing solutions of the problem. Chapter 3 gives an overview of *Lime* as an action-based concurrent object-oriented programming language. Chapter 4 reviews the verification rules for *Lime* language constructs as well as data refinement and superposition refinement of action systems. In chapter 5, we present modularization of *Lime* classes. In this chapter, we also specify module representation of Data Refinement, Superposition Refinement and *Lime* Class Refinement. The remainder of this thesis focuses on inheritance of actions. Chapter 6 develops the rules and designs the specification for inheritance of actions. The class refinement rules for inheritance of actions are also presented in this chapter. Chapter 7 contains a small collection of *Lime* programs that illustrate the use of inheritance of actions. Chapter 8 of the thesis presents inheritance of actions in *Lime* classes as a means to avoid the problem of *Inheritance Anomaly.* We conclude in Chapter 9 with a discussion on the contributions of this thesis and the future direction in which inheritance of actions might go. In Appendix A, we present a complete proof of verification and refinement of inheritance from a class with actions.

Chapter 2

Related Work

2.1 Inheritance of Actions

Action-Oberon, which is an extension of Oberon-2, provides type-bound actions along with type-bound procedures and plain actions in order to model action systems [3].

Type-bound actions are bound to one or more types. A type bound action declared as $ACTION\ A(t : T)$ is an action A bound to the type T. Every time an object of type T is created, an instance of the action A is also created for the newly created object.

Action-Oberon allows inheritance of type-bound actions. However, the language does not permit overriding of the type-bound actions. Since type-bound actions can be bound to more than one types, overriding of type-bound actions would require multiple dispatch. As discussed in [3] the solution to multiple dispatch would clash with independent extensibility of the system. Instead, Action-Oberon achieves the effect of overriding by replacing the body of the type-bound action with a type-bound procedure and by overriding the type-bound procedure.

We illustrate this process of overriding with the fish example from [3]:

PROCEDURE (*f* : *Fish*) *MoveRight* ;
BEGIN INC (*f*.*x*)
END MoveRight ;

PROCEDURE $(s : Shark)$ *MoveRight* ;
BEGIN $s.MoveRight$; $INC(s.hunger)$
END MoveRight ;

PROCEDURE $(f : Fish)$ *WantToMove*() : *BOOLEAN* ;
BEGIN RETURN TRUE
END WantToMove ;

PROCEDURE $(s : Shark)$ *WantToMove*() : *BOOLEAN* ;
BEGIN RETURN $s.hunger < 10$
END WantToMove ;

ACTION $(me : Fish)$ *MoveRight*
WHEN $me.right$ & $(me.x \neq width)$ & $me.WantToMove$;
BEGIN $me.MoveRight$
END MoveRight ;

In this example, overriding of the action *MoveRight* is effectively achieved by the overridden procedures *MoveRight* for *Fish* and *Shark*. In contrast, *Lime* actions are not bound to multiple types. Therefore we permit actions in *Lime* classes to be inherited and overridden without any translations into procedure or method calls.

2.2 Inheritance Anomaly

Inheritance Anomaly is a specific problematic condition that has been observed in Object-Oriented Concurrent Programming languages. Object-oriented concurrent programming languages draw on the benefits of both concurrency and object-oriented programming. However, object-oriented concurrency and inheritance have conflicting properties. The set of methods that can be invoked on a concurrent object depends on the state of the object. Concurrent objects specify synchronization constraints to restrict the methods that can be invoked on the object in a given state. When a class inherits from a concurrent class, it should inherit the synchronization code of the parent class in order to preserve the concurrent behavior of the class. The problem condition is characterized as — when the child class does inherit the synchronization code it needs to non-trivially redefine the methods of the parent class. This leads to a breach in encapsulation. Therefore, during inheritance, the presence of concurrent features in a class causes a conflict in the object-oriented feature of the class. This conflict between inheritance and concurrent objects is known as *Inheritance Anomaly.* This term was first coined by Matsuoka and Yonezawa in [22].

Since the synchronization scheme of *Lime* is based on guarded methods and guarded actions, so we focus on the occurrence of *Inheritance Anomaly* in methods with guards. In the next sections we discuss two conditions in which *Inheritance Anomaly* can occur in methods with guards as presented in [22].

2.2.1 History-Sensitive Methods

When the synchronization scheme is based on method guards, the synchronization code for the method is a boolean expression known as the *guard* of the method. When a guarded method is invoked, first the guard of the method is evaluated. The method is executed only if its guard evaluates to *true*.

Inheritance Anomaly has been observed in guarded methods when the new method added to the subclass is *history-sensitive.* A method is history-sensitive when its synchronization constraint depends on the past history or trace behavior of the object.

We consider the example of bounded buffer from [22] to illustrate the occurrence of *Inheritance Anomaly* in classes with history-sensitive methods. This implementation of bounded buffer is not exactly a FIFO buffer as *in* and *out* need to be incremented modulo *SIZE*. However, this is a classic example used in the literature for discussions on *Inheritance Anomaly.* Therefore, in our literature review we choose to use the example as such.

class *b_buf* {

```
    int in, out, buf[SIZE] ;
  public :
    void b_buf() { in = out = 0 ; }
    void put(int i ) when ( in < out + SIZE )
      { buf[in] = i ; in + + ; }
    int get() when ( in ≥ out + 1 )
      { int result = buf[out] ; out + + ; return result ; }
  }
  class gb_buf : b_buf {
    bool after_put ;
  public :
    void gb_buf() { in = out = 0 ; after_put = false ; }
    int gget() when ( !after_put && ( in ≥ out + 1 )) {
      int result = buf[out]; out + +; after_put = false; return result; }
    // The following methods are redefined.
    void put(int i ) when ( in < out + SIZE )
      { buf[in] = i ; in + + ; after_put = true ; }
    int get() when ( in ≥ out + 1 ) {
      int result = buf[out]; out + +; after_put = false; return result; }
  }
```

The class *b_buf* represents a bounded buffer. It has a method *put*() that adds an integer value to the buffer when it is not full; and a method *get*() that retrieves an integer value from the buffer when it is not empty. The class *gb_buf* inherits from the bounded buffer class *b_buf* and adds a new history-only sensitive method *gget*(). The method *gget*() is similar to *get*() except that it cannot be invoked immediately after invoking *put*(). The guard of *gget*() needs to check if the method called immediately before it was the *put*() method. The class *gb_buf* adds a new flag, *after_put*, to keep track of the invocations of the *put*() method.

In the class *gb_buf*, addition of the history-sensitive method *gget*(), makes it necessary to redefine the methods *put*() and *get*() in order to set and reset the newly added boolean flag *after_put*. Therefore, addition of a history-sensitive method in the subclass of a concurrent class leads to a breakage in encapsulation, thus resulting in *Inheritance Anomaly.*

2.2.2 Modification of Acceptable State

Inheritance Anomaly can also occur in classes with guarded methods when the acceptable state of a method is changed. The next example is of the bounded buffer along with the *Lock* mixin class from [22]. The *Lock* mixin class changes the acceptable states under which the *put*() and *get*() methods are invoked.

```
class b_buf {
  int in, out, buf[SIZE] ;
public :
  void b_buf() { in = out = 0 ; }
  void put(int i ) when ( in < out + SIZE )
    { buf[in] = i ; in + + ; }
  int get() when ( in ≥ out + 1 )
    { int result = buf[out] ; out + + ; return result ; }
}
class Lock {
  bool locked ;
public :
  void Lock() { locked = false ; }
  void lock() when ( !locked ) { locked = true ; }
  void unlock() when ( locked ) { locked = false ; }
}
// lb_buf inherits from b_buf with Lock mixin class.
class lb_buf : b_buf, Lock {
public :
  void lb_buf() ;
  // The following methods are redefined.
  void put(int i ) when ( !locked && ( in < out + SIZE ) )
    { buf[in] = i ; in + + ; }
```

$\textbf{int}\ get()\ \textbf{when}\ (\,!locked\ \&\&\ (\,in \geq out + 1\,)\,)$
$\{\ \textbf{int}\ result = buf[out]\ ;\ out++\ ;\ \textbf{return}\ result\ ;\ \}$
}

The class *lb_buf* inherits from *b_buf* and is with the mixed-in class *Lock*. In the class *lb_buf*, in addition to their original synchronization constraints, the methods *put*() and *get*() can only be executed when the *locked* attribute is not true, i.e. when the object is not locked. The *put*() and *get*() methods from *b_buf* must be redefined in class *lb_buf* in order to inherit the synchronization code of the two methods and to add to it the *locked* constraints. This is the *Inheritance Anomaly* observed in the case when an existing acceptable state is modified by the presence of a mixin class.

2.2.3 Solutions

The problem of *Inheritance Anomaly* persists in many of the modern concurrent object-oriented programming languages such as *Java* and $C^\sharp$ [23]. In the literature there are several approaches to solving or minimizing the problem of *Inheritance Anomaly*. We discuss two of these approaches relevant to *Inheritance Anomaly* in classes with method guards.

Synchronization Patterns [21]: This approach appeals to the notion of *aspect-oriented programming* paradigm. Aspect-oriented programming focuses on the concept of separating out cross-cutting concerns. Following this approach, the solution to *Inheritance Anomaly* is achieved by separating out the synchronization code from the rest of the implementation of the class. *Synchronization pattern* is one such methodology that expresses the synchronization constraints separately in a concurrency block:

sync_patternname
add_structure ... // additional data structures
add_func ... // additional operations
mutex ... // Locks
sync ... // Synchronization scheme

In this approach there are three building blocks: a *structural block*, a *behavioral block* and a *concurrency block*. The *structural block* describes the relationship between the classes, *behavioral block* describes the operations and the *concurrency block* specifies the synchronization constraints. The *structural*, *behavioral* and the *concurrency blocks* together generate the final program. When a history-sensitive method or a change in acceptable state is introduced in the child class, only the *concurrency block* of the child class needs to be changed.

As discussed in [21, 23], the *concurrency* blocks for the classes *b_buf* and *gb_buf* are given by *b_bufSync* and *gb_bufSync* as:

```
sync_pattern b_bufSync
  add_structure        //empty
  add_func             //empty
  mutex per_object x1
  sync
    operation int get()
      at b_buf exclusive x1
        requires (@ !empty @) false (wait)
    operation void put(int i)
      at b_buf exclusive x1
        requires (@ !full @) false (wait)


sync_pattern gb_bufSync : inherit b_bufSync
  add_structure after_put : boolean
  sync
    operation int get()
      at gb_buf exclusive x1
        requires (@ !empty @) false (wait)
        on_exit (@ after_put = false @)
    operation void put(int i)
      at gb_buf exclusive x1
        requires (@ !full @) false (wait)
```

on_exit (@ *after_put* = *true* @)

operation int gget()

at gb_buf exclusive $x1$

requires (@ !*empty* && !*after_put* @) *false* (*wait*)

on_exit (@ *after_put* = *false* @)

Similarly, in the case of *lb_buf*, we only need to add the expression !*locked* in the *requires* clause of the operations of the *concurrency* block *lb_bufSync*.

We observe that in both the cases, only the *concurrency* block is changed with inheritance of classes. The *structural* and *behavioral* blocks are inherited without any changes to existing methods. Therefore there is no breach of encapsulation and *Inheritance Anomaly* is avoided.

Lime does not appeal to the notion of *aspect-oriented programming.* Therefore, this solution cannot be applied to class inheritance in *Lime*.

Guarded Methods as Conditional Critical Regions [15]: This approach is based on nested guarded method calls with open call mechanism. It also relies on super calls to invoke methods from the parent class. In this approach, a guarded method is interpreted as a Conditional Critical Region (CCR). Each guarded method establishes a critical region for accessing a resource of the class, based on a conditional expression—the *guard* of the method. Nested guarded method are interpreted as nested CCRs.

The solution to *Inheritance Anomaly* is realized by invoking the parent class methods using super calls in a nested guarded method structure. The nested method calls should be open calls — if the execution of a guarded method M reaches a nested method call, the method M should release the critical resource and execution of method M should be suspended at the point of the nested method call. This restriction is essential in order to avoid deadlocks. Further, when the execution of the method is suspended, the invariant for the resource should hold and when the suspended method resumes execution, all the nested method guards upto the point of suspension must evaluate to true.

Solution for History-Sensitive Methods :

```
class b_buf {
  int in, out, buf[SIZE] ;
public :
  void b_buf() { in = out = 0 ; }
  void put(int i ) when ( in < out + SIZE ) { buf[in] = i ; in++ ; }
  int get() when ( in ≥ out + 1 )
    { int result = buf[out] ; out++ ; return result ; }
}
class gb_buf : b_buf {
  bool after_put ;
public :
  void gb_buf() { after_put = false ; }
  int gget() when ( !after_put && ( in ≥ out + 1 )) { return get() ; }
  // The following methods are trivially redefined.
  void put(int i ) when ( in < out + SIZE )
    { super.put(i) ; after_put = true ; }
  int get() when ( in ≥ out + 1 )
    { int result = super.get() ; after_put = false ; return result ; }
}
```

Solution for Modification of Acceptable State :

```
class b_buf {
  int in, out, buf[SIZE] ;
public :
  void b_buf() { in = out = 0 ; }
  void put(int i ) when ( in < out + SIZE ) { buf[in] = i ; in++ ; }
  int get() when ( in ≥ out + 1 )
    { int result = buf[out] ; out++ ; return result ; }
}
class Lock {
  bool locked ;
```

```
public :
    void Lock() { locked = false ; }
    void lock() when (!locked) { locked = true ; }
    void unlock() when ( locked ) { locked = false ; }
}
// lb_buf inherits from b_buf with Lock mixin class.
class lb_buf : b_buf, Lock {
public :
    void lb_buf() ;
    // The following methods are trivially redefined.
    void put(int i ) when (!locked ) { super.put(i) ; }
    int get() when (!locked ) { return super.get() ; }
}
```

In both examples, the methods *put*() and *get*() in the classes *gb_buf* and *lb_buf* achieve their functionality by invoking the *super.put*() and *super.get*() methods. Therefore, the redefinition of these methods is trivial in nature. *Inheritance Anomaly* is avoided in both the examples by using nested guarded methods with open calls. In Chapter 8, we discuss how *Lime* has a similar approach to solving *Inheritance Anomaly* with guarded methods and actions.

In literature, there are other approaches to alleviate the problem of *Inheritance Anomaly.* For example, Jeeg [24] is a dialect of Java based on method guards. It specifies the synchronization constraints in linear temporal logic and appeals to the notion of aspect-oriented programming in its approach to addressing *Inheritance Anomaly.* Fournet et al. [16] approach to dealing with *Inheritance Anomaly* extends join calculus with classes and objects and applies class inheritance for behavioral and synchronization inheritance. JAC [20] is another extension of Java that relies on concurrency annotations to address the problem of *Inheritance Anomaly.* It achieves condition synchronization by using precondition annotation and guard annotation. A language independent aspect-oriented solution to *Inheritance Anomaly* is developed in [30]. This approach uses *Microsoft Intermediate Language* to specify the functional components and aspects. It uses an *aspect model* to decompose and a *weaver* *model* to compose the synchronization constraints and functional components.

Chapter 3

Introduction Of Lime

In an object-oriented programming language, objects are self-contained units that evolve independently of each other. Since it is also an object-oriented language, in *Lime*, an object is considered as a unit of concurrency [27]. *Lime* is an object-oriented language that has concurrent features built into its classes, and is based on the action system formalism.

Concurrency is introduced in *Lime* classes by adding actions to classes and by adding guards to methods. The level of concurrency in *Lime* is classified as *quasi-concurrent*, as at any given time several methods or actions in an object may be enabled, but only one method or action can be executed. The method or action, that is allowed to progress, is chosen non-deterministically. In an object, methods as well as actions may be enabled repeatedly.

In a *Lime* program, an object corresponds to an action system with procedures as defined in [8, 29]. The *Lime* program itself corresponds to a parallel composition of actions systems. The next section discusses *Lime* syntax and its language features.

3.1 Syntax Of Lime

The formal syntax of the language is given in extended BNF below [28]. The construct $a \mid b$ stands for either a or b, $[a]$ means that a is optional, and $\{a\}$ means that a can be repeated zero or more times:

class	::=	**class** *identifier*
		{ *attribute* | *initialization* | *method* | *action* } **end**
attribute	::=	**var** *variableList*
initialization	::=	**initialization** [(*variableList*)] *statement*
method	::=	**method** *identifier* [(*variableList*[, **res** *variableList*])]
		[**when** *expression* **do**] *statement*
action	::=	**action** *identifier* [**when** *expression* **do**] *statement*
statement	::=	**assert** *expression* |
		designatorList := *expressionList* |
		designatorList :∈ *expression* |
		[*designator* :=] *designator*.*identifier*
		[(*expressionList*)] |
		designator := **new** *identifier* [(*expressionList*)] |
		var *variableList* ; *statement*
		begin *statement* { ; *statement* } **end** |
		if *expression* **then** *statement* [**else** *statement*] |
		while *expression* **do** *statement*
variableList	::=	*identifierList* : *type* { , *identifierList* : *type* }
identifierList	::=	*identifier* { , *identifier* }
designatorList	::=	*designator* { , *designator* }
expressionList	::=	*expression* { , *expression* }

A class is declared by giving it a name and then listing all the attributes (instance variables), initializations, methods, and actions. Both methods and actions must have a name. Methods names need not be unique as methods can be overloaded. However, action names must be unique. Both methods and actions may optionally have a *guard*. The *guard* is a Boolean expression that must only refers to attributes of the object itself and cannot contain method calls. A method or action is *enabled* if its guard is true or missing, otherwise it is *disabled*. A *guarded object* is an object with guarded methods. An object is *active* if its class definition includes one or more actions, otherwise it is *passive*. An active object with at least one enabled action is called an *enabled object*, otherwise it is a *disabled object*.

Lime supports behavior specifications within its class definition. In a *Lime* class, expressions are provided with a specialized assertion language that is useful for writing behavior specifications. [18]

3.2 Methods and Actions

In a *Lime* program, methods can be invoked by name. Only enabled methods can be executed. Actions, on the other hand, cannot be invoked by name. Instead, when an action is enabled, it is executed autonomously. While methods may have value parameters and may return a result, actions do not have any parameters and they do not return any results, but they can affect the state space of the object.

Methods and actions are atomic upto method calls. Execution of an action or method gets suspended at the point where a method is called. Method call may result in control being passed to another object. In that case, in the original object, another enabled action or enabled method call can be executed or a suspended method or action can resume its execution. This is in contrast with Seuss approach [25], where only one call to a guarded method is allowed and that method call has to be the first statement in an action or method.

3.3 An Example in Lime

We consider a simple example to demonstrate the various language features of *Lime*. In this example, the *Lime* class represents a cup that can be filled and emptied.

class *Cup*
  var $state : (full, half, empty)$
  var $f, p : boolean$
  initialization $state, f, p := empty, false, false$
  method *fill*
    when $\neg f \wedge \neg p \wedge (state \neq full)$ **do**
      $f := true$

method *pour*
  when $\neg f \wedge \neg p \wedge (state \neq empty)$ **do**
    $p := true$
action *doFill*
  when f **do**
    begin
      $f := false$;
      if $state = half$ **then** $state := full$
      else if $state = empty$ **then** $state := half$
    end
action *doPour*
  when p **do**
    begin
      $p := false$;
      if $state = full$ **then** $state := half$
      else if $state = half$ **then** $state := empty$
    end
end

var $c : Cup, n : integer$;
begin $n := 0$; **while** $n < 7$ **do** $c :=$ **new** *Cup* **end**

In this example, the Lime class represents a cup that can hold some liquid. At any given time, the cup can be in one of the following three states: *full*, *half* (*half*-full or *half*-empty) and *empty*. Upon initialization, the state of the cup is set to *empty*. The two activities that can be performed on the cup are to fill it up or empty it out. The methods *fill* and *pour* are used to fill the cup with liquid and pour the liquid out of the cup respectively. We assume that the cup is filled up and emptied in steps of half a cup. For example, when fill is called on an empty cup, after the method and the corresponding action are executed, the state of the cup becomes *half* (half-full).

The method *fill* is enabled if the cup is not currently being filled or emptied and also if the cup is not already full. Method *fill* then enables the action *doFill.* Action *doFill* fills up the cup by changing the state from *empty* to *half* or from *half* to *full.* Similarly, the method *pour* is enabled if the cup is not currently being filled or emptied and also if the cup is not already empty. Method *pour* then enables the action *doPour.* Action *doPour* empties the cup by changing the state from *full* to *half* or from *half* to *empty.*

If the Lime program consists of a collection of such cups, at any given time, each cup in the collection can have at the most one method or action executing. Concurrent behavior of the program is expessed as follows: each cup in the collection can have at most one method or action that can execute concurrently with methods and actions being executed by the other cups in the collection. If there are N cups in the collection, then there can be maximum N number of operatons executing concurrently.

3.4 Inheritance in Lime

Apart from the usual benefits of inheritance, class inheritance is also useful in *Lime* programs for introducing concurrency in subclasses. Given a class implementing (part of) a sequential program, a subclass of this class can implement a corresponding concurrent program.

Lime differentiates between subclassing and subtyping [18]. Single subclassing is achieved by the *extend* clause. Multi-subtyping is achieved by the *implement* clause. Subclassing and subtyping together is achieved by the *inherit* clause.

Lime uses three access modifiers to specify the visibility rules: *public*, *protected* and *private*. Public variables and methods are visible to the class, its subclasses and their objects. Protected variables and methods are visible to the class itself and its subclasses. Private variables and methods are only visible to the class itself and not its subclasses.

***D extend C* :**

If a class D extends the class C, then D inherits all non-private variables of C; D also inherits each of the non-private method of C such that it inherits the method's signature and implementation. Since D does not inherit the method's behavior, it can override the method without having to preserve the superclass method's behavior.

D implement $C_1, C_2, \ldots C_n$:
If a class D implements the classes $C_1, C_2, \ldots C_n$, then D preserves the invariants of all the supertypes $(C_1, C_2, \ldots C_n)$; D inherits all non-private variables from all the supertypes $(C_1, C_2, \ldots C_n)$; D also inherits each of the non-private method of all the supertypes, such that it inherits the method's signature and behavior specifications. Since D does not inherit the method's implementation, it should provide the method's implementation or be declared as *abstract*. The method's implementation must preserve the method's inherited behavior specifications.

D inherit C : It is equivalent to *D extend C implement C*.
If a class D inherits from the class C, then D preserves the invariants of C; D inherits all non-private variables of C; D also inherits each of the non-private methods of C, such that it inherits the method's signature, implementation and behavior specifications. D may either keep the method's original implementation or D may choose to override the method to provide a new implementation. The new implementation for the method must preserve the method's inherited behavior specifications.

Class inheritance in *Lime* should reflect *Lime*'s model of concurrency that views object-orientation and concurrency as a single design issue. Therefore, in Chapter 6, we expand class inheritance in *Lime* using inherit clause to include inheritance of actions.

Chapter 4

Mathematical Fundamentals

In this chapter, we review from related work, the verification rules and action system formalism which form the mathematical foundation for verification and refinement in *Lime*. In addition, we specify some new verification rules.

4.1 Verification of Statements

In this section we discuss the statements in *Lime* and their rules for verification as given in [28]. The correctness of all atomic statements of *Lime* can be analysed by using Dijkstra's weakest precondition predicate transformer: $wp(S, c)$ is the weakest precondition such that S terminates and establishes postcondition c. It is also assumed that all statements are monotonic, for any statement S and any Boolean expressions b, c:

$$(b \Rightarrow c) \quad \Rightarrow \quad (wp(S, b) \Rightarrow wp(S, c)) \tag{4.1}$$

At its core, *Lime* has the following atomic statements: the *empty statement* *skip*, the *assertion statement* $\{b\}$, the *guard statement* $[b]$, the *multiple assignment* $x := e$, the *nondeterministic assignment* $x :\in s$, the *nondeterministic choice* $S \sqcap T$ between statements S and T, the *unbounded choice* $\sqcap x \in s \bullet S$ and the *sequential composition* $S\ ;\ T$. $f[x \backslash e]$ denotes the expression f with all free occurrences of x substituted by e, where x is a list of variables and e a list of expressions. These rules apply when the evaluation of all expressions succeeds.

$$
\begin{aligned}
wp(skip, c) &\equiv c && (4.2)\\
wp(\{b\}, c) &\equiv b \wedge c && (4.3)\\
wp([b], c) &\equiv b \Rightarrow c && (4.4)\\
wp(x := e, c) &\equiv c[x \backslash e] && (4.5)\\
wp(x :\in s, c) &\equiv (\forall x \in s \bullet c) && (4.6)\\
wp(S \sqcap T, c) &\equiv wp(S, c) \wedge wp(T, c) && (4.7)\\
wp(\sqcap x \in s \bullet S, c) &\equiv (\forall x \in s \bullet wp(S, c)) \quad x \text{ not free in } c && (4.8)\\
wp(\sqcap x \bullet S, c) &\equiv (\forall x \bullet wp(S, c)) \quad x \text{ not free in } c && (4.9)\\
wp(S \; ; \; T, c) &\equiv wp(S, wp(T, c)) && (4.10)
\end{aligned}
$$

When evaluation of expressions does not succeed, a different set of rules apply factoring in the undefinedness of expressions. These rules are discussed later.

The nondeterministically initialized local variable declaration **var** $x \in s$; S stands for $\sqcap x \in s \bullet S$. The local variable declaration **var** $x : T$; S stands for $\sqcap x \bullet S$, where x ranges over all elements of type T. The local variable declaration with initialization **var** $v = v_0$ stands for **var** $v \in \{v_0\}$. The corresponding derived rules are:

$$
\begin{aligned}
wp(\textbf{var } x \in s \; ; \; S, c) &\equiv \forall x \in s \bullet wp(S, c) \quad x \text{ not free in } c && (4.11)\\
wp(\textbf{var } x : T \; ; \; S, c) &\equiv \forall x \bullet wp(S, c) \quad x \text{ not free in } c && (4.12)\\
wp(\textbf{var } v = v_0, c) &\equiv \forall v \in \{v_0\} \bullet wp(S, c) \quad v \text{ not free in } c && (4.13)
\end{aligned}
$$

The guarded statement **when** b **do** S, the assert statement **assert** b **do** S and the conditional statements are defined as:

$$
\begin{aligned}
\textbf{when } b \textbf{ do } S &\mathrel{\widehat{=}} [b] \; ; \; S && (4.14)\\
\textbf{assert } b \textbf{ do } S &\mathrel{\widehat{=}} \{b\} \; ; \; S && (4.15)\\
\textbf{if } b \textbf{ then } S &\mathrel{\widehat{=}} ([b] \; ; \; S) \sqcap [\neg b] && (4.16)\\
\textbf{if } b \textbf{ then } S \textbf{ else } T &\mathrel{\widehat{=}} ([b] \; ; \; S) \sqcap ([\neg b] \; ; \; T) && (4.17)
\end{aligned}
$$

These are some derived rules:

$$
\begin{aligned}
wp(\textbf{when } b \textbf{ do } S, c) &\equiv b \Rightarrow wp(S, c) && (4.18)\\
wp(\textbf{assert } b \textbf{ do } S, c) &\equiv b \wedge wp(S, c) && (4.19)\\
wp(\textbf{if } b \textbf{ then } S, c) &\equiv (b \Rightarrow wp(S, c)) \wedge (\neg b \Rightarrow c) && (4.20)
\end{aligned}
$$

$$wp(\textbf{if } b \textbf{ then } S \textbf{ else } T, c) \equiv (b \Rightarrow wp(S,c)) \wedge (\neg b \Rightarrow wp(T,c)) \quad (4.21)$$

Given the procedure declaration **procedure** $p(x\ ;\ \textbf{res}\ y)S\ ;\ T$ with body S and scope T, the call $p(e,v)$ within T is defined by **var** $x, y\ ;\ x := e\ ;\ S\ ;\ v := y$

When evaluation of expressions does not succeed, the undefinedness of expressions in statements should be taken into account. Let $\Delta\, e$ represent the definedness of a program expression e. A statement terminates if evaluation of all expressions succeeds. The redefined weakest preconditions for statements factoring in possibly undefined expressions are:

$$wp(\{b\}, c) \equiv \Delta\, b \wedge b \wedge c \quad (4.22)$$
$$wp([b], c) \equiv \Delta\, b \wedge (b \Rightarrow c) \quad (4.23)$$
$$wp(x := e, c) \equiv \Delta\, e \wedge c[x \backslash e] \quad (4.24)$$

The *enabledness domain* of S is defined by $en\,S = \neg wp(S, \mathit{false})$ and the *termination domain* by $tr\,S = wp(S, \mathit{true})$. We have:

$$en(\{b\}\ ;\ S) \equiv en\,S \quad (4.25)$$
$$en([b]\ ;\ S) \equiv b \wedge en\,S \quad (4.26)$$
$$en(\sqcap x \in s \bullet S) \equiv (\exists x \in s \bullet en\,S) \quad (4.27)$$
$$tr(\{b\}\ ;\ S) \equiv b \wedge tr\,S \quad (4.28)$$
$$tr([b]\ ;\ S) \equiv en\,S \quad (4.29)$$

An iteration statement **while** g **do** S can also be expressed as: **do** $g \rightarrow S$ **od** where $g \rightarrow S$ is a guarded command with guard g and body S. Here, S can be executed repeatedly as long as the boolean condition g (the guard) evaluates to true. We can generalize this to the following form for iteration over guarded commands:

$$\begin{array}{lll} \textbf{do} & & g_0 \rightarrow S_0 \\ & \sqcap & g_1 \rightarrow S_1 \\ & \vdots & \\ & \sqcap & g_n \rightarrow S_n \\ \textbf{od} & & \end{array}$$

We can also write the iteration statement as [26]:

$$\textbf{do } (\sqcap i \bullet g_i \rightarrow S_i) \textbf{ od}$$

An iteration statement can be expanded as:

$$\begin{array}{ll} & \textbf{do } g \rightarrow S \textbf{ od} \\ \equiv & \textbf{if } g \textbf{ then} \\ & \quad S \; ; \\ & \quad \textbf{do } g \rightarrow S \textbf{ od} \\ & \textbf{fi} \end{array}$$

We give the proof rules for iteration statement in terms of an invariant I and a variant V as:

$$I \wedge g_i \quad \Rightarrow \quad wp(S_i, I), \forall i \bullet 1 \leq i \leq n \tag{4.30}$$

$$I \wedge G \quad \Rightarrow \quad V > 0 \tag{4.31}$$

$$I \wedge g_i \wedge V = v \quad \Rightarrow \quad wp(S_i, V < v), \forall i \bullet 1 \leq i \leq n \tag{4.32}$$

Then, $I \Rightarrow wp(\textbf{do } (\sqcap i \bullet g_i \rightarrow S_i) \textbf{ od}, I \wedge \neg G)$.

where V is an integer expression, v is an auxiliary integer variable and $G = \bigvee_i g_i$ is the disjunction of the guards of the guarded commands in the iteration statement. Each iteration decreases the value of V and the iteration statement becomes disabled when $V = 0$.

We justify these proof rules based on the treatment of **do** loops in terms of an *invariant* and a *bound function* in [17].

Partial correctness is sufficient for proving invariance properties. Partial correctness is defined in terms of *weakest liberal preconditions* as: Statement S preserves the invariant I means $I \Rightarrow wlp(S, I)$. While the predicate $wp(S, c)$ is the weakest precondition for a statement S to terminate and establish the postcondition c, the weakest liberal precondition $wlp(S, c)$ is the weakest precondition for the statement S to establish c provided S terminates, defined as $wlp(S, c) \equiv tr\ S \Rightarrow wp(S, c)$. If all program expressions are defined, we have:

$$wlp(\{b\}, c) \quad \equiv \quad b \Rightarrow c \tag{4.33}$$

$$wlp([b], c) \quad \equiv \quad b \Rightarrow c \tag{4.34}$$

$$\begin{aligned} wlp(x := e, c) &\equiv c[x \backslash e] && (4.35)\\ wlp(x :\in s, c) &\equiv (\forall x \in s \bullet c) && (4.36)\\ wlp(S\ ;\ T, c) &\Leftarrow wlp(S, wlp(T, c)) && (4.37) \end{aligned}$$

Taking into account the undefinedness of program expressions, we have:

$$\begin{aligned} wlp(\{b\}, c) &\equiv \Delta\, b \wedge b \Rightarrow c && (4.38)\\ wlp([b], c) &\equiv \Delta\, b \wedge b \Rightarrow c && (4.39)\\ wlp(x := e, c) &\equiv \Delta\, e \Rightarrow c[x \backslash e] && (4.40) \end{aligned}$$

The (finite) *conjunctivity* of *wlp* follows from the (finite) conjunctivity of *wp*. These *wlp* rule may be used in invariance proofs:

$$\begin{aligned} wp(S, b \wedge c) &\equiv wp(S, b) \wedge wp(S, c) && (4.41)\\ wlp(S, b \wedge c) &\equiv wlp(S, b) \wedge wlp(S, c) && (4.42) \end{aligned}$$

The declaration of a class C translates to the declaration of a global variable C that holds the set of all objects of class C and for each variable f of type F, a global variable $C.f$ mapping objects of C to values of F:

$$\begin{aligned} &\mathbf{var}\ C : \mathbf{set\ of}\ \mathit{Object} && (4.43)\\ &\mathbf{var}\ C.f : \mathit{Object} \rightarrow F && (4.44) \end{aligned}$$

It is assumed that the type *Object* contains infinitely many elements, including the unique element *nil*. Accessing a variable f of object o, written $o.f$ amounts to applying the function f to o. A variable assignment is equivalent to a function update:

$$\begin{aligned} o.f &= f(o) && (4.45)\\ o.f := e &= f := f[o \leftarrow e]) && (4.46) \end{aligned}$$

The expression $f[a \leftarrow r]$ is used for modifying function f to return r for argument a. We have:

$$\begin{aligned} a.f[a \leftarrow r] &= r && (4.47)\\ b.f[a \leftarrow r] &= b.f, \qquad b \neq a && (4.48) \end{aligned}$$

The nondeterministic assignment $x := ?$, defined as $\sqcap h \bullet x := h$, assigns to x an arbitrary value of its type:

$$wp(x := ?, c) \equiv (\forall x \bullet c) \qquad (4.49)$$

$$wp(o.f := ?, c) \equiv (\forall h \bullet c[f \backslash f[o \leftarrow h]]) \tag{4.50}$$

If I is the body of the initialization of class C, or *skip* if no initialization is declared, M is the body of method *meth* of C, and A is the body of action *act*. Let $C.init$ represent $this.a := ?\ ;\ I$, where a are the variables that are not assigned to in I. The declaration of class C results in the following definitions, for each method *meth* and action *act*:

$$\begin{aligned} C.new &= this :\notin C \cup \{nil\}\ ;\ C := C \cup \{this\}\ ;\ C.init & (4.51)\\ C.meth &= \{this \in C\}\ ;\ M & (4.52)\\ C.act &= (\sqcap this \in C \bullet A) & (4.53) \end{aligned}$$

The class name is also used as a prefix for the method and actions names. The expression $x :\notin s$ replaces $x :\in \overline{s}$, where $\overline{s}$ is the complement of set s. The action $C.act$ is defined in terms of a nondeterministic choice in order to model concurrency through *interleaving*: when two actions operating on disjoint state spaces are enabled, they can be executed in parallel.

A new element of class C is created by finding an unused element of C, adding that to C, and executing the body of the initialization. If v are the formal parameters of the initialization:

$$o := \textbf{new}\ C(e) = \textbf{var}\ this, v\ ;\ v := e\ ;\ C.new\ ;\ o := this \tag{4.54}$$

To illustrate parameter passing with methods calls, an *atomic* method call is defined as follows: Suppose method m of class C is declared with value parameters v and to return a result r. Then an *atomic* call $c.m(e, r)$ for $c \in C$ makes c and e to be the actual value parameters and r the actual result parameter:

$$\begin{aligned} c.meth(e, r) = {} & \textbf{var}\ this, v, result\ ; \\ & this, v := c, e\ ;\ C.meth\ ;\ r := result \end{aligned} \tag{4.55}$$

4.2 Action System Model of Concurrency

The action system formalism [4, 5, 2] is used to model the concurrent behavior of parallel and distributed programs. The behavior of the distributed and

parallel programs is described by the actions that can take place in the processes executing in the system. Each action is a guarded command of the form $A = g \rightarrow S$, where g is a predicate known as the *guard* of the action and S is the *body* of the action. Whenever the guard of the action is true, the action is enabled. Only enabled actions can be chosen for execution. Execution of an enabled action is achieved by executing the body of the action. In an action system, more than one action can be enabled at the same time. Since actions are atomic, these enabled actions can be executed in parallel as long as they do not have any variables in common. The enabled action to be executed next is chosen non-deterministically. One or more enabled actions are executed until the action system terminates. An action system terminates when it does not have any enabled actions.
An action system is a statement of the form:

$$\mathcal{A} \;=\; \text{begin} \;\; \text{var } x := x_0; \text{ do } A_1 \sqcap \ldots \sqcap A_m \text{ od } \;\text{end} \;:\; \text{z}$$

where x are the local variables of $\mathcal{A}$, z are the global variables of $\mathcal{A}$ and A_1, ..., A_m are the actions of $\mathcal{A}$. The local variables and global variables do not overlap, $x \cap z = \oslash$. The state variables, w, of an action system are the union of the local and global variables: w = x $\cup$ z.

Two or more actions are *independent* if they do not have any state variables in common. Otherwise they are termed as *competing* actions. Since actions are atomic, and two or more actions can be executed in parallel only if they are independent actions, therefore parallel execution of an action system is guaranteed to give the same results as a sequential and non-deterministic execution.

4.3 Action System with Procedures

An action systems with procedures [8, 29, 9] is a statement of the form:

$$\begin{array}{lll} \mathcal{A} = \; |[& \textbf{var} & y^*, x := y_0, x_0 \,; \\ & \textbf{proc} & p_1^* = P_1 \,; \;\cdots\, ; \; p_n^* = P_n \,; \end{array}$$

$$
\begin{array}{ll}
& \quad q_1 = Q_1 \ ; \ \cdots \ ; \ q_m = Q_m \ ; \\
& \mathbf{do}\ A_1 \sqcap \cdots \sqcap A_k\ \mathbf{od} \\
]| : z, r &
\end{array}
$$

The *local variables* x are local to the action system $\mathcal{A}$. The variables y^* are *exported global variables* that can be used locally in $\mathcal{A}$ or globally by other action systems in parallel with $\mathcal{A}$. The variables z are *imported global variables* that can be referred to in $\mathcal{A}$ but are not declared in $\mathcal{A}$.

Similarly, the *local procedures* q_i are local to the action system $\mathcal{A}$ and can only be called by actions of $\mathcal{A}$. The procedures p_i^* are *exportable procedures* that can be used locally by actions of $\mathcal{A}$ or by actions of some other action system put in parallel with $\mathcal{A}$. The procedures r are *imported global procedures* that can be called by actions in $\mathcal{A}$ but are not declared in $\mathcal{A}$. The variables x, y, and z are pairwise distinct. The *local* and *global* procedures are also distinct.

4.4 Parallel Composition of Action Systems

Individual action systems can be composed in parallel to construct a larger system of interacting components. $\mathcal{A}$ and $\mathcal{B}$ are two action systems of the form:

$$
\begin{array}{llll}
\mathcal{A} = & |[& \mathbf{var} & v^*, x := v_0, x_0 \ ; \\
& & \mathbf{proc} & p_1^* = P_1 \ ; \ \cdots \ ; \ p_n^* = P_n \ ; \\
& & & d_1 = D_1 \ ; \ \cdots \ ; \ d_i = D_i \ ; \\
& & \multicolumn{2}{l}{\mathbf{do}\ A_1 \sqcap \cdots \sqcap A_k\ \mathbf{od}} \\
&]| : z, r & &
\end{array}
$$

and

$$
\begin{array}{llll}
\mathcal{B} = & |[& \mathbf{var} & w^*, y := w_0, y_0 \ ; \\
& & \mathbf{proc} & q_1^* = Q_1 \ ; \ \cdots \ ; \ q_n^* = Q_n \ ; \\
& & & e_1 = E_1 \ ; \ \cdots \ ; \ e_j = E_j \ ; \\
& & \multicolumn{2}{l}{\mathbf{do}\ B_1 \sqcap \cdots \sqcap B_h\ \mathbf{od}} \\
&]| : z', r' & &
\end{array}
$$

where the module $\mathcal{A}$ has global variables z and global procedures r and the module $\mathcal{B}$ has global variables z' and global procedures r'. In addition, $x \cap y = \oslash$, $v \cap w = \oslash$, $d \cap e = \oslash$ and $p \cap q = \oslash$.
The parallel composition of $\mathcal{A}$ and $\mathcal{B}$ is defined as $\mathcal{C} = \mathcal{A} \sqcap \mathcal{B}$:

$$
\begin{array}{lll}
\mathcal{C} = & |[& \textbf{var} \quad g^*, x, y := g_0, x_0, y_0 \ ; \\
& & \textbf{proc} \quad p_1^* = P_1 \ ; \ \cdots \ ; \ p_n^* = P_n \ ; \ q_1^* = Q_1 \ ; \ \cdots \ ; \ q_m^* = Q_m \ ; \\
& & \qquad\quad d_1 = D_1 \ ; \ \cdots \ ; \ d_i = D_i \ ; \ e_1 = E_1 \ ; \ \cdots \ ; \ e_j = E_j \ ; \\
& & \textbf{do}\ A_1 \sqcap \cdots \sqcap A_k \sqcap B_1 \sqcap \cdots \sqcap B_h \ \textbf{od} \\
&]| : a, b &
\end{array}
$$

where $a = z \cup z' - (v \cup w)$ and $b = r \cup r' - (p \cup q)$ are the *imported global variables* and *imported global procedures* of the action system $\mathcal{C}$. The exported global variables of the two action systems are merged together into $g = v \cup w$.

The assumption here is that the action systems $\mathcal{A}$ and $\mathcal{B}$ do not have any local variables in common, $x \cap y = \oslash$. If the two action systems do have some local variables in common, the common variables can be renamed in one action system to keep the local variables distinct.

In the parallel composition of two action systems, the local variables of the two participating action systems are kept distinct unlike the global variables. The global variables are shared among all the actions in parallel composition in the resultant action system. The resultant action system has all the actions from both action systems in parallel composition as well as all the local and exportable procedures from both the action systems.

4.5 Data Refinement

Data refinement, as discussed in [1, 2], is a method of refinement of an action system which involves a change in the state space of the action system. The technique of data refinement involves replacing the more abstract state variables with more concrete state variables.

Ordinary (algorithmic) refinement of statement S by T, written $S \sqsubseteq T$ is defined as:

$$S \sqsubseteq T \equiv \forall c \bullet wp(S, c) \Rightarrow wp(T, c) \tag{4.56}$$

This implies that T can be used for whatever S can be, but T may be "more deterministic", may have a weaker termination domain, and may have a stronger guard.

Data Refinement of Statements: Data refinement $S \sqsubseteq_R T$ generalizes algorithmic refinement by allowing S and T to operate on different variables, related through *coupling invariant* or *refinement invariant* R. Let S be a statement over variables s and T a statement over variables t, where s and t are disjoint. Let R be a predicate over s and t, known as the *abstraction relation* between the variables. Data refinement of statement S by T through R, written $S \sqsubseteq_R T$ is defined as:

$$S \sqsubseteq_R T \equiv \forall c \bullet R \wedge wp(S, c) \Rightarrow wp(T, \exists s \bullet R \wedge c) \tag{4.57}$$

An equivalent formulation of data refinement is given in terms of the *conjugate weakest precondition* predicate transformer $\overline{wp}$ [13]. *Conjugate weakest precondition* is defined as $\overline{wp}(S, c) \equiv \neg wp(S, \neg c)$. If all program expressions are defined, we have from [28]:

$$\overline{wp}(\{b\}, c) \equiv b \Rightarrow c \tag{4.58}$$
$$\overline{wp}([b], c) \equiv b \wedge c \tag{4.59}$$
$$\overline{wp}(x := e, c) \equiv c[x \backslash e] \tag{4.60}$$
$$\overline{wp}(x :\in s, c) \equiv (\exists x \in s \bullet c) \tag{4.61}$$
$$\overline{wp}(S \; ; \; T, c) \equiv \overline{wp}(S, \overline{wp}(T, c)) \tag{4.62}$$
$$\overline{wp}(S \sqcap T, c) \equiv \overline{wp}(S, c) \vee \overline{wp}(T, c) \tag{4.63}$$
$$\overline{wp}(\sqcap x \in s \bullet S, c) \equiv (\exists x \in s \bullet \overline{wp}(S, c)) \qquad x \text{ not free in } c \tag{4.64}$$

If program expressions are possibly undefined we have:

$$\overline{wp}(\{b\}, c) \equiv \Delta\, b \wedge b \Rightarrow c \tag{4.65}$$
$$\overline{wp}([b], c) \equiv \Delta\, b \Rightarrow b \wedge c \tag{4.66}$$
$$\overline{wp}(x := e, c) \equiv \Delta\, e \Rightarrow c[x \backslash e] \tag{4.67}$$

In [13] data refinement is formulated in terms of the *conjugate weakest precondition* predicate transformer $\overline{wp}$, as:

$$S \sqsubseteq_R T \equiv R \wedge tr\, S \Rightarrow wp(T, \overline{wp}(S, R)) \tag{4.68}$$

In case S and T have variables x in common, the definition of data refinement needs to be generalized to account for the common variables x. Let $S[x\backslash\overline{x}]$ stand for statement S with variables x substituted by variables $\overline{x}$. It is assumed that $\overline{x}$ are fresh variables:

$$S \sqsubseteq_R T \equiv R \wedge tr\ S \Rightarrow wp(T[x\backslash\overline{x}], \overline{wp}(S, R \wedge x = \overline{x})) \tag{4.69}$$

A useful special case is the refinement of *skip*:

$$skip \sqsubseteq_R T \equiv R \Rightarrow wp(T, R) \tag{4.70}$$

Components of a sequential composition can be refined individually:

$$S_0 \sqsubseteq_R T_0 \wedge S_1 \sqsubseteq_R T_1 \Rightarrow S_0\ ;\ S_1 \sqsubseteq_R T_0\ ;\ T_1 \tag{4.71}$$

Data Refinement of Actions: [1, 2] Let A and A' be actions on the state variables x, z and x', z respectively. Let $R(x, x', z)$ be an *abstraction relation* between the variables. If A is defined as $gA \rightarrow sA$ and A' is defined as $gA' \rightarrow sA'$, then $A \sqsubseteq_R A'$, if

(a) $\mathrm{R} \wedge \mathrm{gA}' \Rightarrow \mathrm{gA}$

(b) $(\forall\ \mathrm{c} \cdot \mathrm{R} \wedge \mathrm{gA}' \wedge \mathrm{wp}(\mathrm{sA}, \mathrm{c}) \Rightarrow \mathrm{wp}(\mathrm{sA}', \exists\ \mathrm{x} \cdot \mathrm{R} \wedge \mathrm{c}))$

The first condition represents the refinement of guards. It requires that a refinement step may strengthen the guard of an action. The second condition represents the refinement of bodies. It requires that a refinement step expand the domain of termination and decrease the non-determinism of the body.

Data Refinement of Action Systems with procedures: [8, 29, 9] Let $\mathcal{A}$ and $\mathcal{A}'$ be two action systems of the form:

$$\begin{array}{lll} \mathcal{A} = & |[\ \ \mathbf{var} & z^*, x := z_0, x_0\ ; \\ & \quad\ \mathbf{proc} & p_1^* = P_1\ ;\ \cdots\ ;\ p_n^* = P_n\ ; \\ & & q_1 = Q_1\ ;\ \cdots\ ;\ q_m = Q_m \\ & \quad\ \mathbf{do}\ A_1 \sqcap \cdots \sqcap A_k\ \mathbf{od} & \\ &]|\ :\ u, r & \end{array}$$

$$
\begin{array}{lll}
\mathcal{A}' = & \| [& \textbf{var} \quad z^*, x' := z_0, x'_0 \,; \\
& & \textbf{proc} \quad p^*_1 = P'_1 \,; \;\cdots;\; p^*_n = P'_n \,; \\
& & \qquad\quad q_1 = Q'_1 \,; \;\cdots;\; q_m = Q'_m \\
& & \textbf{do}\; A'_1 \sqcap \cdots \sqcap A'_k \sqcap H_1 \sqcap \cdots \sqcap H_j \;\textbf{od} \\
&] | : u, r &
\end{array}
$$

where H_j are the stuttering actions which correspond to a *skip* statement on the global state of $\mathcal{A}$. It is assumed that every exportable global procedure p is locally enabled. Then $\mathcal{A} \sqsubseteq_R \mathcal{A}'$ if there exists an *abstraction relation* R(x, x$'$, z, u, f) on local variables x and x', exported global variables z, imported global variables u and the formal parameters f of the exportable global procedures p, such that

(a) Initialization : R(x_0, x'_0, z_0, u, f),

(b) Procedures : $P_i \sqsubseteq_R P'_i$,

(c) Procedure enabledness : $R \wedge gP_i \Rightarrow gP'_i \vee gA' \vee gH$,

(d) Main actions : $A \sqsubseteq_R A'$,

(e) Continuation condition : $R \wedge gA \Rightarrow gA' \vee gH$,

(f) Auxiliary actions : $\mathit{skip} \sqsubseteq_R H$,

(g) Internal Convergence : $R \Rightarrow \mathit{wp}(\textbf{do}\; H \;\textbf{od}\;, \mathit{true})$,

(h) Non-interference : $R \wedge \mathit{wp}(E, \mathit{true}) \Rightarrow \mathit{wp}(E, R)$ for every action E of an action system $\mathcal{E}$ where $\mathcal{A}$ occurs in a parallel composition with the action system $\mathcal{E}$.

where $\bigvee_i \mathit{en}\, A_i = gA$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}$, $\bigvee_i \mathit{en}\, A'_i = gA'$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}'$, $\bigvee_i \mathit{en}\, H_i = gH$ is the disjunction of enabledness domains of the auxiliary actions of $\mathcal{A}'$ and $\sqcap_i H_i = H$ denotes the combined action for the auxiliary actions of $\mathcal{A}'$.

Condition (a) requires that initialization establish the abstraction relation. Condition (b) requires that the body of each global procedure P_i is data refined by the body of the corresponding global procedure P'_i. Condition (c) requires that whenever R holds, if a global procedure P_i is enabled in $\mathcal{A}$ then either the corresponding global procedure P'_i or an action in $\mathcal{A}'$ is enabled. Condition (d) requires that action A is data refined by the action A' using R. Condition (e) requires that whenever R holds, the continuation condition of $\mathcal{A}$ implies the continuation condition of $\mathcal{A}'$. Condition (f) requires that the stuttering action H acts as *skip* statement on global variables u, z. The stuttering actions do not have any effect on the global state of the action system being refined. Condition (g) requires that execution of stuttering actions in isolation, must terminate. Condition (h) requires that upon parallel composition of the action systems $\mathcal{A}$ and $\mathcal{E}$, interleaved execution of actions from $\mathcal{E}$ preserve the abstraction relation.

4.6 Superposition Refinement of Action Systems with Procedures

Superposition refinement [7] of action systems as a special case of data refinement. With superposition refinement new functionalities are added while preserving the original computation of the action system. Adding new functionalities involves adding new state variables and modifying existing actions or adding new actions or both.

In the superposition refinement of $\mathcal{A}$, an action system with procedures, the refining action system $\mathcal{A}'$ adds new local variables and new actions to modify the new variables, but it has to retain all the old variables from $\mathcal{A}$. The action system $\mathcal{A}'$ may also modify the existing actions and global procedures of the action system $\mathcal{A}$.

Let $\mathcal{A}$ and $\mathcal{A}'$ be two action systems of the form:

$$\begin{array}{lll} \mathcal{A} = & |[\ \mathbf{var} & z^*, x := z_0, x_0 \ ; \\ & \quad \mathbf{proc} & p_1^* = P_1 \ ; \ \cdots ; \ p_n^* = P_n \ ; \\ & & q_1 = Q_1 \ ; \ \cdots ; \ q_m = Q_m \end{array}$$

$$
\begin{array}{lll}
 & & \textbf{do } A_1 \sqcap \cdots \sqcap A_k \textbf{ od} \\
 &]| : u, r & \\
 & & \\
\mathcal{A}' = & |[& \textbf{var} \quad z^*, x, x' := z_0, x_0, x'_0 \; ; \\
 & & \textbf{proc} \quad p_1^* = P'_1 \; ; \; \cdots \; ; \; p_n^* = P'_n \; ; \\
 & & \qquad\quad q_1 = Q'_1 \; ; \; \cdots \; ; \; q_m = Q'_m \\
 & & \textbf{do } A'_1 \sqcap \cdots \sqcap A'_k \sqcap B_1 \sqcap \cdots \sqcap B_j \textbf{ od} \\
 &]| : u, r &
\end{array}
$$

Both $\mathcal{A}$ and $\mathcal{A}'$ have the same global variables. $\mathcal{A}'$ has new local variables x'. $\mathcal{A}'$ also retains the old local variables x. For each old action A_i in $\mathcal{A}$ there is a corresponding action A'_i in $\mathcal{A}'$. In addition, $\mathcal{A}'$ also has *auxiliary actions* B_j that do not correspond to any actions in $\mathcal{A}$. The existing actions or global procedures of $\mathcal{A}$ may be modified by corresponding actions or global procedures of $\mathcal{A}'$.

If R(x, x', z, u, f) is an *abstraction relation* on local variables x and x', exported global variables z, imported global variables u and the formal parameters f of the global procedures p, then $\mathcal{A} \sqsubseteq_R \mathcal{A}'$ under the following conditions:

(a) Initialization : R(x_0, x'_0, z_0, u, f),

(b) Procedures : $P_i \sqsubseteq_R P'_i$,

(c) Procedure enabledness : $R \wedge gP_i \Rightarrow gP'_i \vee gA' \vee gB$,

(d) Main actions : $A \sqsubseteq_R A'$,

(e) Continuation condition : $R \wedge gA \Rightarrow gA' \vee gB$,

(f) Auxiliary actions : $\mathit{skip} \sqsubseteq_R B$,

(g) Internal Convergence : $R \Rightarrow \mathit{wp}(\textbf{do } B \textbf{ od}, \mathit{true})$,

(h) Non-interference : $R \wedge \mathit{wp}(E, \mathit{true}) \Rightarrow \mathit{wp}(E, R)$ for every action E of an action system $\mathcal{E}$ where $\mathcal{A}$ occurs in a parallel composition with the action system $\mathcal{E}$.

where $\bigvee_i en\, A_i = gA$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}$, $\bigvee_i en\, A'_i = gA'$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}'$, $\bigvee_i en\, B_i = gB$ is the disjunction of enabledness domains of the auxiliary actions of $\mathcal{A}'$ and $\sqcap_i B_i = B$ denotes the combined action for the auxiliary actions of $\mathcal{A}'$.

These conditions can be summarized as follows: Initialization must establish the abstraction relation, R. The global procedures and actions of $\mathcal{A}$ are data refined by the corresponding global procedures and actions of $\mathcal{A}'$. Whenever a global procedure is enabled in $\mathcal{A}$, either the corresponding global procedure or an action of $\mathcal{A}'$ is enabled. When R holds, the continuation condition of $\mathcal{A}$ implies the continuation condition of $\mathcal{A}'$. The auxiliary actions B_j do not have any effect on the old variables x,z, when R(x, x′, z) holds. Also, each auxiliary action establishes the abstraction relation. The execution of auxiliary actions B_j eventually terminates. Execution of the action system $\mathcal{A}$ in parallel composition with another action system $\mathcal{E}$ preserves the refinement invariant for execution of any action of $\mathcal{E}$

Chapter 5

Concurrency and Modularization

On one hand we have classes that model object-oriented features and on the other hand we have action systems that model concurrent behavior of programs. In order to model concurrent object-oriented programs, *Lime* classes are translated into a form analogous to action systems. We achieve this by extending modules to correspond to action systems with procedures and by translating each class in *Lime* into an equivalent module representation.

5.1 Action Systems and Modules

An action system with procedures includes variables, procedures and actions. The variables in the action system can be *exported global variables*, *imported global variables* or *local variables*. The procedures in the action system can be *exportable global procedures*, *imported global procedures* or *local procedures*.

Consider the action system $\mathcal{A}$:

$$
\begin{array}{lll}
\mathcal{A} = & |[& \textbf{var} \quad y^*, x := y_0, x_0 \ ; \\
& & \textbf{proc} \quad p_1^* = P_1 \ ; \ \cdots \ ; \ p_n^* = P_n \ ; \\
& & \qquad\quad q_1 = Q_1 \ ; \ \cdots \ ; \ q_m = Q_m \ ; \\
& & \textbf{do} \ A_1 \sqcap \cdots \sqcap A_k \ \textbf{od} \\
&]| : z, r &
\end{array}
$$

In $\mathcal{A}$, x are the local variables, y^* are the *exported global variables* and z are *imported global variables*, q_i are the *local procedures*, p_i^* are the *exportable global procedures* and r are the *imported global procedures.*

An action system can be represented as a module in *Lime* using the syntax for module definition in [27]. We extend this definition of modules to add visibility modifiers to its variables and procedure declarations. The *local* members of the action system can be modeled by *private* members of the module, the *exported global* members of the action system can be modeled by *public* members of the module and the *imported global* members (u) of the action system can be modeled by importing another module where u are declared as *public* members. A module also names the actions.

Since a class in *Lime* can specify a class invariant, the module representation of the class must have a corresponding invariant. Therefore we extend the definitions of modules to introduce invariants.

We define a module as:

module M **import** $N, \ldots$
  private var $v \in V$
  public var $w \in W$
  $\ldots$
  invariant I
  private procedure $p(\ldots)\ P$
  public procedure $q(\ldots)\ Q$
  $\ldots$
  action $a\ A$
  $\ldots$
end

A module is well-formed if —

(a) within the module, statements only refer to public variables and public procedures of other modules.

(b) the invariant refer to only the variables of the module and of directly or indirectly imported modules. It may also refer to private variables of other modules

A module is correct if —

(a) *Initialization*: the initialization establishes the invariant
$v \in V \wedge w \in W \Rightarrow I$

(b) *Procedure Correctness*: all public procedures preserve the invariant
$\{I\}Q\{I\}$

(c) *Action Correctness*: all actions preserve the invariant
$\{I\}A\{I\}$

If these three conditions are satisfied, then I is the module invariant.

The invariant for a module is a predicate $I(x, y, z, f)$ over the private variables, public variables, imported public variables and formal parameters of the public procedures.

With these additions, now the module is formally a structure with a set of imported modules, a set of private variable declarations, a set of public variable declarations, an invariant, a set of public procedure declarations, a set of private procedure declarations and a named set of actions. The action system with procedures $\mathcal{A}$ is represented as a module of the form:

module $\mathcal{A}$ **import** G
  private var $x := x_0$
  public var $y := y_0$
  invariant $I(x, y, z)$
  public procedure $p_1(s_1 : S_1,$ **res** $t_1 : T_1)$
    P_1
  ...
  public procedure $p_n(s_n : S_n,$ **res** $t_n : T_n)$
    P_n
  private procedure $q_1(u_1 : U_1,$ **res** $v_1 : V_1)$
    Q_1
  ...
  private procedure $q_m(u_m : U_m,$ **res** $v_m : V_m)$
    Q_m

action a_1

A_1

...

action a_k

A_k

end

In the module $\mathcal{A}$, the *imported global variables* z and the *imported global procedures* r are imported from the module G where z and r are *public* variables and procedures respectively.

As we are introducing invariants in modules, we define refinement in terms of invariants as well. We consider first the data refinement of statement S by statement T through the abstraction relation R as:

$$S \sqsubseteq_R T \equiv R \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, R))$$

We decompose the *abstraction relation* R into an abstract invariant I over the variables of S and a coupling invariant J over the variables of S and T. We define the data refinement of statement S by statement T through I and J as:

$$S \sqsubseteq_J^I T \equiv S \sqsubseteq_{I \wedge J} T$$

and

$$S \sqsubseteq_J^I T \equiv I \wedge J \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, I \wedge J))$$

Theorem 1. *If S preserves I, then* $S \sqsubseteq_J^I T \equiv I \wedge J \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, J))$

Proof:

$$\begin{array}{ll}
 & S \sqsubseteq_J^I T \\
\equiv & < \textit{definition} > \\
 & \quad I \wedge J \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, I \wedge J)) \\
\equiv & < \textit{conjunctivity} > \\
 & \quad I \wedge J \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, I) \wedge \overline{wp}(S, J)) \\
\equiv & < \text{T preserves } \overline{wp}(S, I) \text{ as } \overline{wp}(S, I) \text{ does not refer to variables of T;} \\
 & \quad \text{conjunctivity} > \\
 & \quad I \wedge J \wedge tr\ S \Rightarrow \overline{wp}(S, I) \wedge wp(T, \overline{wp}(S, J))
\end{array}$$

$\equiv$ $<$ assumption: S preserves I $>$

$I \wedge J \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, J))$ $\square$

Let S be a statement over variables v and I a predicate over v. Let T be a statement over variables w and J a predicate over both v and w, where v and w are disjoint. Using the formulation for data refinement based on the conjugate weakest precondition predicate transformer $\overline{wp}$ defined as $\overline{wp}(S, c) \equiv \neg wp(S, \neg c)$ [13], and from Theorem 1, statement S is refined by T through I and J, written $S \sqsubseteq_I^J T$, is defined by:

$$S \sqsubseteq_J^I T \equiv I \wedge J \wedge tr\ S \Rightarrow wp(T, \overline{wp}(S, J)) \tag{5.1}$$

In case S and T have variables x in common, the definition needs to be extended. Let S be a statement over variables x and T be a statement over variables x and y. Let $T[x \backslash \overline{x}]$ stand for statement T with variables x substituted by variables $\overline{x}$. Assume that $\overline{x}$ are fresh variables:

$$S \sqsubseteq_J^I T \equiv I \wedge J \wedge tr\ S \Rightarrow wp(T[x \backslash \overline{x}], \overline{wp}(S, J \wedge x = \overline{x})) \tag{5.2}$$

We present an equivalent notation for $S \sqsubseteq_J^I T$ when S and T have variables x in common:

$$S(x) \leq_{J(x,y)}^{I(x)} T(x, y) \ \hat{=}\ S(x) \sqsubseteq_{J(\overline{x},y) \wedge x = \overline{x}}^{I(x)} T(\overline{x}, y) \tag{5.3}$$

The special case is the refinement of *skip* is defined by:

$$skip \leq_J^I T \equiv I \wedge J \Rightarrow wp(T, J) \tag{5.4}$$

Components of a sequential composition can be refined individually:

$$S_0 \leq_J^I T_0 \wedge S_1 \leq_J^I T_1 \Rightarrow S_0 \ ; \ S_1 \leq_J^I T_0 \ ; \ T_1 \tag{5.5}$$

Invariant preservation is defined in terms of refinement as:

$$I\,\{S\}\,I \equiv S \leq_I S \tag{5.6}$$

5.1.1 Module Representation of Parallel Composition of Action Systems

Consider the module representation of two actions systems $\mathcal{A}$ and $\mathcal{B}$:

module $\mathcal{A}$ **import** G
  private var $x := x_0$
  public var $v := v_0$
  invariant $I(x, v, z, f_A)$
  public procedure $p_1(s_1 : S_1$, **res** $t_1 : T_1)$
    P_1
  $\ldots$
  public procedure $p_n(s_n : S_n$, **res** $t_n : T_n)$
    P_n
  private procedure $d_1(u_1 : U_1$, **res** $l_1 : L_1)$
    D_1
  $\ldots$
  private procedure $d_i(u_i : U_i$, **res** $l_i : L_i)$
    D_i
  action a_1
    A_1
  $\ldots$
  action a_k
    A_k
end

and

module $\mathcal{B}$ **import** H
  private var $y := y_0$
  public var $w := w_0$
  invariant $J(y, w, z', f_B)$
  public procedure $q_1(s'_1 : S'_1$, **res** $t'_1 : T'_1)$
    Q_1

...
public procedure $q_m(s'_m : S'_m,$ **res** $t'_m : T'_m)$
Q_m
private procedure $e_1(u'_1 : U'_1,$ **res** $l'_1 : L'_1)$
E_1
...
private procedure $e_j(u'_j : U'_j,$ **res** $l'_j : L'_j)$
E_j
action b_1
B_1
...
action b_h
B_h
end

where the module $\mathcal{A}$'s global variables z and global procedures r are imported from module G; the module $\mathcal{B}$'s global variables z' and global procedures r' are imported from module H. f_A are the formal parameters of the public procedures of module $\mathcal{A}$ and f_B are the formal parameters of the public procedures of module $\mathcal{B}$. In addition, $x \cap y = \oslash$, $v \cap w = \oslash$, $d \cap e = \oslash$ and $p \cap q = \oslash$. $I(x, v, z, f_A)$ is the module invariant for $\mathcal{A}$ and $J(y, w, z', f_B)$ is the module invariant for $\mathcal{B}$.

The module representation of parallel composition of $\mathcal{A}$ and $\mathcal{B}$ is defined as $\mathcal{C} = \mathcal{A} \sqcap \mathcal{B}$:

module $\mathcal{C}$ **import** G, H
public var $g := g_0$
private var $x, y := x_0, y_0$
invariant $K(x, y, g, a, f)$
public procedure $p_1(s_1 : S_1,$ **res** $t_1 : T_1)$
P_1
...
public procedure $p_n(s_n : S_n,$ **res** $t_n : T_n)$

P_n
public procedure $q_1(s'_1 : S'_1,$ **res** $t'_1 : T'_1)$
Q_1
...
public procedure $q_m(s'_m : S'_m,$ **res** $t'_m : T'_m)$
Q_m
private procedure $d_1(u_1 : U_1,$ **res** $l_1 : L_1)$
D_1
...
private procedure $d_i(u_i : U_i,$ **res** $l_i : L_i)$
D_i
private procedure $e_1(u'_1 : U'_1,$ **res** $l'_1 : L'_1)$
E_1
...
private procedure $e_j(u'_j : U'_j,$ **res** $l'_j : L'_j)$
E_j
action a_1
A_1
...
action a_k
A_k
action b_1
B_1
...
action b_h
B_h
end

where $g = v \cup w$ are the *exported global variables* of $\mathcal{C}$. $a = z \cup z' - (v \cup w)$ are the *imported global variables* and $b = r \cup r' - (p \cup q)$ are the *imported global procedures* of the action system $\mathcal{C}$. $f = f_A \cup f_B$ are the formal parameters of the public procedures of $\mathcal{C}$.

5.1.2 Module Representation of Data Refinement

Let $\mathcal{A}$ and $\mathcal{A}'$ be two action systems of the form:

module $\mathcal{A}$ import G
  private var $x := x_0$
  public var $z := z_0$
  invariant $I(x, z, u, f)$
  public procedure $p_1(s_1 : S_1,$ res $t_1 : T_1)$
    P_1
  ...
  public procedure $p_m(s_m : S_m,$ res $t_m : T_m)$
    P_m
  private procedure $q_1(u_1 : U_1,$ res $v_1 : V_1)$
    Q_1
  ...
  private procedure $q_n(u_n : U_n,$ res $v_n : V_n)$
    Q_n
  action a_1
    A_1
  ...
  action a_k
    A_k
end

module $\mathcal{A}'$ import G
  private var $x' := x'_0$
  public var $z := z_0$
  invariant $J(x, x', z, u, f)$
  public procedure $p_1(s_1 : S_1,$ res $t_1 : T_1)$
    P'_1
  ...
  public procedure $p_m(s_m : S_m,$ res $t_m : T_m)$
    P'_m

private procedure $q_1(u_1 : U_1,$ **res** $v_1 : V_1)$
Q_1'
...
private procedure $q_n(u_n : U_n,$ **res** $v_n : V_n)$
Q_n'
action a_1
A_1'
...
action a_k
A_k'
action h_1
H_1
...
action h_l
H_l
end

It is assumed that every exportable global procedure p_i is locally enabled. I(x, z, u, f) and J(x, x', z, u, f) are the invariants of $\mathcal{A}$ and $\mathcal{A}'$ respectively, on local variables x and x', exported global variables z, imported global variables u and the formal parameters f of the exported global procedures p. Then $\mathcal{A}$ is data refined by $\mathcal{A}'$, written $\mathcal{A} \sqsubseteq_J^I \mathcal{A}'$, if the following conditions hold:

(a) Initialization : $I(x_0, z_0, u, f) \wedge J(x_0, x_0', z_0, u, f)$,

(b) Procedures : $P_i \sqsubseteq_J^I P_i'$,

(c) Procedure enabledness : $I \wedge J \wedge gP_i \Rightarrow gP_i' \vee gA' \vee gH$,

(d) Main actions : $A \sqsubseteq_J^I A'$,

(e) Continuation condition : $I \wedge J \wedge gA \Rightarrow gA' \vee gH$,

(f) Auxiliary actions : $skip \sqsubseteq_J^I H$,

(g) Internal Convergence : $I \wedge J \Rightarrow wp($**do** H **od** $, true)$,

(h) Non-interference : $I \wedge J \wedge wp(E, true) \Rightarrow wp(E, J)$ for every action E of an action system $\mathcal{E}$ where $\mathcal{A}$ occurs in a parallel composition with the action system $\mathcal{E}$.

where $\bigvee_i en\ A_i = gA$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}$, $\bigvee_i en\ A'_i = gA'$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}'$, $\bigvee_i en\ H_i = gH$ is the disjunction of enabledness domains of the auxiliary actions of $\mathcal{A}'$ and $\sqcap_i H_i = H$ denotes the combined action for the auxiliary actions of $\mathcal{A}'$.

5.1.3 Module Representation of Superposition Refinement

In superposition refinement of action systems [7] new non-public state variables may be added; existing actions or procedures may be modified and new actions or procedures may be added in the concrete action system.
Let $\mathcal{A}$ and $\mathcal{A}'$ be two action systems of the form:

module $\mathcal{A}$ **import** G
  private var $x := x_0$
  public var $z := z_0$
  invariant $I(x, z, u, f)$
  public procedure $p(s_1 : S_1,$ **res** $t_1 : T_1)$
    P_1
  ...
  public procedure $p(s_m : S_m,$ **res** $t_m : T_m)$
    P_m
  private procedure $q_1(u_1 : U_1,$ **res** $v_1 : V_1)$
    Q_1
  ...
  private procedure $q_n(u_n : U_n,$ **res** $v_n : V_n)$
    Q_n
  action a_1
    A_1

...
action a_k
A_k
end

module $\mathcal{A}'$ **import** G
private var $x, x' := x_0, x'_0$
public var $z := z_0$
invariant $I(x, x', z, u, f)$
public procedure $p_1(s_1 : S_1,$ **res** $t_1 : T_1)$
P'_1
...
public procedure $p_m(s_m : S_m,$ **res** $t_m : T_m)$
P'_m
private procedure $q_1(u_1 : U_1,$ **res** $v_1 : V_1)$
Q'_1
...
private procedure $q_n(u_n : U_n,$ **res** $v_n : V_n)$
Q'_n
action a_1
A'_1
...
action a_k
A'_k
action b_1
B_1
...
action b_l
B_l
end

Both $\mathcal{A}$ and $\mathcal{A}'$ have the same exported global variables. $\mathcal{A}'$ has new local variables x'. $\mathcal{A}'$ also retains the old local variables x. For each old action A_i in

$\mathcal{A}$ there is a corresponding action A'_i in $\mathcal{A}'$. In addition, $\mathcal{A}'$ also has *auxiliary actions* B_j that do not correspond to any actions in $\mathcal{A}$. The existing actions or global procedures of $\mathcal{A}$ may be modified by corresponding actions or global procedures of $\mathcal{A}'$.

I(x, z, u, f) and J(x, x', z, u, f) are the invariants of $\mathcal{A}$ and $\mathcal{A}'$ respectively, on local variables x and x', exported global variables z, imported global variables u and the formal parameters f of the exported global procedures p. Then $\mathcal{A}$ is refined under superposition by $\mathcal{A}'$, written $\mathcal{A} \sqsubseteq_J^I \mathcal{A}'$, if the following conditions hold:

(a) Initialization : $I(x_0, z_0, u, f) \wedge J(x_0, x'_0, z_0, u, f)$,

(b) Procedures : $P_i \sqsubseteq_J^I P'_i$,

(c) Procedure enabledness : $I \wedge J \wedge gP_i \Rightarrow gP'_i \vee gA' \vee gB$,

(d) Main actions : $A \sqsubseteq_J^I A'$,

(e) Continuation condition : $I \wedge J \wedge gA \Rightarrow gA' \vee gB$,

(f) Auxiliary actions : $skip \sqsubseteq_J^I B$,

(g) Internal Convergence : $I \wedge J \Rightarrow wp(\textbf{do } B \textbf{ od}, true)$,

(h) Non-interference : $I \wedge J \wedge wp(E, true) \Rightarrow wp(E, J)$ for every action E of an action system $\mathcal{E}$ where $\mathcal{A}$ occurs in a parallel composition with the action system $\mathcal{E}$.

where $\bigvee_i en\, A_i = gA$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}$, $\bigvee_i en\, A'_i = gA'$ is the disjunction of enabledness domains of the main actions of $\mathcal{A}'$, $\bigvee_i en\, B_i = gB$ is the disjunction of enabledness domains of the auxiliary actions of $\mathcal{A}'$ and $\sqcap_i B_i = B$ denotes the combined action for the auxiliary actions of $\mathcal{A}'$.

5.2 Modules and Module Refinement

In *Lime*, module representation is used to represent a program in an action system format in order to model the concurrent behavior of the program. Module actions are executed atomically — either an action is enabled and can be executed to completion or it is not enabled, which is unlike class actions that are atomic only up to method calls [27].

A *Lime* program P is a collection of modules M_i representing the classes and a main statement S. We write the program P as:

program P **import** $M_1, M_2, \ldots$; S

A program is well-formed if —

(a) the statement S refers to only the public variables and public procedures of the imported modules.

(b) the import structure of modules is acyclic.

In this formulation, a program is an action system in which S is the initialization, and the action system contains all variables, actions and procedures of all (transitively) imported modules. Thus, a program with imported modules is a parallel composition of action systems represented by the imported modules.

$$
\begin{array}{l}
\textit{program}\ P\ \textbf{import}\ M_1, \ldots\ ;\ S \equiv \\
\quad \textbf{var}\ v, w, \ldots\ ; \\
\quad \textbf{procedure}\ p(\ldots)\ P, \textbf{procedure}\ q(\ldots)\ Q, \ldots\ ; \\
\quad S\ ; \\
\quad \textbf{do}\ A_i \sqcap \ldots\ \textbf{od}
\end{array}
$$

The invariant of the program is a predicate I such that S establishes I and all actions A_i preserve I. This condition is only on actions and not on procedures as procedures are ultimately called from actions and actions are atomic.

Let M be a module with invariant I. A module N does not interfere with M if initialization of N preserves I and all actions of N preserve I. Again, this condition is only on actions and not procedures.

Theorem 2. *Let M be a module with invariant I. If all other modules do not interfere with M, then I is an invariant of the program.*

Proof: Let a program P be defined as: *program* P **import** $M, N_1, N_2, \ldots ;\ S$

(a) Initialization of M establishes I and each action and procedure of M preserves I, as I is the invariant of module M.

(b) Each of the other modules N_i imported in P does not interfere with the module M. Therefore, for each module N_i of P, the initialization of N_i preserves I and all the actions of N_i preserve I.

Since program P is a parallel composition of the modules $M, N_1, N_2, \ldots$, from (a) and (b) above, the initialization S of the program P establishes the invariant I and all actions of the program P preserve the invariant I. Thus, I is the invariant of the program.

□

Some special cases in which a module invariant can become a program invariant:

(a) If the invariant I of M refers to only the private variables of M.

(b) If the invariant I of M refers to public variables of M, and the modules directly importing M do not interfere with M.

(c) If the invariant I of M refers to variables of an imported module N, and all the modules directly importing N do not interfere with M.

Module Refinement Let M be a module of the form:

> **module** M **import** $\ldots$
>   **private var** $v \in V$
>   **public var** $w \in W$
> $\ldots$
>   **invariant** $I(v, w, u, f)$
>   **public procedure** $q_1(\ldots)\ Q_1$
> $\ldots$

action a_1 A_1

...

end

The module M is refined by another module M'. The invariant J of module M' establishes the relation between the variables of M and M'. u are the *global imported variables*, f are the formal parameters of the public variables.

module M' **import** ... **refines** M
  private var $v' \in V'$
  public var $w \in W$
  invariant $J(v, v', w, u, f)$
  public procedure $q_1(\ldots)$ Q_1'
  ...
  action a_1 A_1'
  ...
  action b_1 B_1'
  ...
end

Module M' is well-formed if it includes the public variables, public procedures, and actions of M (though with possibly different bodies) and is otherwise a well-formed module. Module M is refined by module M', written $M \sqsubseteq M'$, if

(a) *Initialization*: $v \in V \wedge v' \in V' \wedge w \in W \Rightarrow J$

(b) *Procedure Refinement*: $Q_i \sqsubseteq_J^I Q_i', \ldots$

(c) *Procedure Enabledness*: $I \wedge J \wedge \mathit{en}\ Q_i \Rightarrow \mathit{en}\ Q_i' \vee \mathit{en}\ A' \vee \mathit{en}\ B'$

(d) *Main Action Refinement*: $A_i \sqsubseteq_J^I A_i'$

(e) *Main Action Enabledness*: $I \wedge J \wedge \mathit{en}\ A_i \Rightarrow \mathit{en}\ A' \vee \mathit{en}\ B'$

(f) *Auxiliary Action Refinement*: $\mathit{skip} \sqsubseteq_J^I B'$

(g) *Auxiliary Action Termination*: $I \wedge J \Rightarrow \mathit{tr}(\mathbf{do}\ B'\ \mathbf{od})$

where $en\,A'$ stands for disjunction of enabledness domain of all A'_i actions and $en\,B'$ stands for disjunction of enabledness domain of all B'_i actions.

Theorem 3. *Let P be a program that imports (directly or indirectly) module M. Let module M' be a refinement of module M. If all other modules do not interfere with M' and M' does not interfere with all other modules, then P is refined by replacing M with M'.*

Proof: Let a program P be defined as: *program* P **import** $M, N_1, N_2, \ldots N_k$; S. Let the set $\mathcal{N} = \{N_1, N_2, \ldots N_k\}$ denote the set of other modules other than M in program P.

Let the program P' be obtained by replacing the module M by M' in program P. The program P' is defined as: *program* P' **import** $M', N_1, N_2, \ldots N_k$; S

For each module $N_i \in \mathcal{N}$, module M' does not interfere with module N_i; module N_i does not interfere with module M', by assumption. Therefore, program P' preserves the invariant of program P.

Module M' is a refinement of module M, $M \sqsubseteq M'$, by assumption. For each module $N_i \in \mathcal{N}$, $N \sqsubseteq N$, as refinement relation is reflexive. Therefore, program P (which is a parallel composition of modules M and N_i) is refined by program P' (which is a parallel composition of modules M' and N_i), $P \sqsubseteq P'$.
$\square$

If the invariant J of M' refers to only the private variables of M and M', then no other module can interfere with M'. If M' does not import any modules, then M' does not interfere with other modules.

5.3 Classes and Class Refinement

Lime classes are translated into action systems formalism by defining the class within a module using module syntax with procedures and actions.

A class in *Lime* contains a set of variable declarations, an invariant, an initialization statement, a set of method declarations, and a named set of actions. The variables and methods of the class can be declared with *public,* *private* or *default* visibility. We write a class C in *Lime* as:

> **class** C
>   **var** $u : U$
>   **private var** $v : V$
>   **public var** $w : W$
>   **invariant** I
>   **initialization** $K(e : E)$
>   **method** $m(q : Q,$ **res** $b : B)\, M$
>   **private method** $n(s : S,$ **res** $g : G)\, N$
>   **public method** $o(t : T,$ **res** $h : H)\, O$
>   ...
>   **action** $a\, A$
>   ...
> **end**

The variables and methods with no access modifiers associated with them are known as *default variables* and *default methods* respectively. In class C above, variable u is a default variable and method m is a default method. Default members of a class are not visible outside of the module where the class is defined. Within the module containing the class definition, the default members of the class are visible to all members of the module.

As discussed in [27], each *Lime* class can be defined within a module with one module variable for each class variable, one procedure for each method, a procedure for initialization and an extra variable for the objects populating that class. The class invariant is translated into the invariant for the class declaration within the module. Each action in the class is translated into a corresponding action in the module. The variables map each object of the class to the corresponding variable values. Each procedure takes an additional value parameter, *this*, for the object to which the procedure is applied. In contrast, for an action, *this* is assigned nondeterministically any object of that class before the action is executed. All objects are of type *Object*. It is assumed that type *Object* is infinite and contains an unique element *nil*. Class initialization is translated to a procedure *new* that takes an additional result parameter *this*. The parameter *this* returns a newly created object which is not *nil* and which is not in the set of existing objects of this class. $x :\notin s$ is used as a shorthand for $x :\in \overline{s}$.

The class definition for C within a module amounts to following module declarations:

> **private var** C : **set of** *Object* $= \{\}$
> **var** $u : Object \rightarrow U$
> **private var** $v : Object \rightarrow V$
> **public var** $w : Object \rightarrow W$
> **invariant** $(\forall this \in C \bullet I)$
> **public procedure** $C.is(x : Object$; **res** $r : boolean)$
> $\quad r := x \in C$
> **procedure** $C.new(e : E,$ **res** $this : Object)$
> $\quad this :\notin C \cup \{nil\}$; $C := C \cup \{this\}$; K
> **procedure** $C.m(this : Object, q : Q,$ **res** $b : B)$
> $\quad \{this \in C\}$; M
> **private procedure** $C.n(this : Object, s : S,$ **res** $g : G)$
> $\quad \{this \in C\}$; N
> **public procedure** $C.o(this : Object, t : T,$ **res** $h : H)$
> $\quad \{this \in C\}$; O
> **action** $C.a$
> $\quad$ **var** $this :\in C \bullet A$

In the class definition within the module we introduce a procedure $C.is$ for a class C, which performs a type test. It is commonly invoked as r := x is C rather than C.is(x, r). This procedure is useful for performing a type test from outside of a module. However, this also means that refinement only works for programs that do not contain type tests in general: If a program includes the test x is C and C is replaced by C', even if C' is a refinement the resulting program is not.

If a class is defined as private in a module, then all variables and procedures become private to the module. If C is defined public, then private and default variables and methods become private variables and procedures outside of the module and public variables and methods become public variables and procedures. The default and public variables of a class may be modified by other procedures and methods of the module.

A class is well-formed if the resulting module is well-formed and the private variables and methods of the class are only referred to within the class. A class is correct if following three conditions hold:

(a) *Constructor Invariant Preservation*:
$\{\forall this \in C \bullet I\}$ $(this :\notin C \cup \{nil\}\ ;\ C := C \cup \{this\}\ ;\ K)$ $\{\forall this \in C \bullet I\}$

(b) *Public Method Invariant Preservation*:
$\{\forall this \in C \bullet I\}$ $(\{this \in C\}\ ;\ M)$ $\{\forall this \in C \bullet I\}$

(c) *Actions Invariant Preservation*:
$\{\forall this \in C \bullet I\}$ $(\sqcap this \in C \bullet A)$ $\{\forall this \in C \bullet I\}$

Theorem 4. *Suppose I is an invariant of class C in module M. If (after translating all methods to procedures) all other procedures and all other actions of M preserve $(\forall this \in C \bullet I)$, then $(\forall this \in C \bullet I)$ is a module invariant.*

Proof: Within the module M, the initialization of C with $\{\}$ trivially establishes $(\forall this \in C \bullet I)$. All methods and actions of C preserve $(\forall this \in C \bullet I)$, since $(\forall this \in C \bullet I)$ is the invariant of class C. By assumption, all other procedures (after translation) and actions of M preserve $(\forall this \in C \bullet I)$ as well. Therefore, $(\forall this \in C \bullet I)$ is an invariant of the module M. $\square$

As a consequence, if I is only over the private variables of C, then $(\forall this \in C \bullet I)$ is a module invariant if it is a class invariant. In general a method needs to not only to preserve the class invariant, but also the globally stated module invariant.

We allow a module to contain more than one class definitions. In that case, a module is allowed to have several invariant clauses. Each class defined within the module contributes one invariant clause. All the invariant clauses from all the classes in the module are conjoined to form the module invariant. Each class defined within a module has to preserve the module invariant.

Class Refinement Class refinement is based on the notion of *data refinement* of action systems with procedures [28]. A class D refining another class C is well formed if it includes the public variables, public methods, and actions of C (though with different bodies).

Consider the classes C and D declared below:

class C
  var $u : U$
  private var $v : V$
  public var $w : W$
  invariant I
  initialization K
  method m_1 M_1
  $\dots$
  public method o_1 O_1
  $\dots$
  action a_1 A_1
  $\dots$
end

class D **refines** C
  var $u' : U'$
  private var $v' : V'$
  public var $w : W$
  invariant J
  initialization K'
  method m_1 M_1'
  $\dots$
  public method o_1 O_1'
  $\dots$
  action a_1 A_1'
  $\dots$
  action b_1 B_1
  $\dots$

end

In order to establish the conditions under which class C is refined by class D, we translate the *Lime* classes into action systems with procedures. This translation into action systems is achieved by defining the classes C and D in a module M. Class refinement is verified under the conditions for data refinement *between* action systems. We can then reason about refinement of class C by class D in terms of refinement of the corresponding class declarations within the module M under the conditions for data refinement.

The class definition of C within a module M amounts to following module declarations:

private var C : **set of** $\mathit{Object} = \{\}$
var $u : \mathit{Object} \rightarrow U$
private var $v : \mathit{Object} \rightarrow V$
public var $w : \mathit{Object} \rightarrow W$
invariant $(\forall \mathit{this} \in C \bullet I)$
procedure $C.\mathit{new}($**res** $\mathit{this} : \mathit{Object})$
  $\mathit{this} :\notin C \cup \{\mathit{nil}\}\ ;\ C := C \cup \{\mathit{this}\}\ ;\ K$
procedure $C.m_1(\mathit{this} : \mathit{Object})$
  $\{\mathit{this} \in C\}\ ;\ M_1$
  ...
public procedure $C.o_1(\mathit{this} : \mathit{Object})$
  $\{\mathit{this} \in C\}\ ;\ O_1$
  ...
action $C.a_1$
  var $\mathit{this} :\in C \bullet A_1$
  ...

The class definition of D within a module M' amounts to following module declarations:

private var D : **set of** $\mathit{Object} = \{\}$
var $u' : \mathit{Object} \rightarrow U'$
private var $v' : \mathit{Object} \rightarrow V'$

public var $w : Object \to W$
invariant $(\forall this \in D \bullet J)$
procedure $D.new($**res** $this : Object)$
  $this :\notin D \cup \{nil\}$; $D := D \cup \{this\}$; K'
procedure $D.m_1(this : Object)$
  $\{this \in D\}$; M_1'
  ...
public procedure $D.o_1(this : Object)$
  $\{this \in D\}$; O_1'
  ...
action $D.a_1$
  var $this :\in D \bullet A_1'$
  ...
action $D.b_1$
  var $this :\in D \bullet B_1$
  ...

We extend the definition of Class Refinement from [28] so that the abstract class is refined by the concrete class through the class invariants instead of the refinement invariant.

Definition 5.1 (Class Refinement). *Let C be a class with default variables u, private variables v, and public variables w. Let D be a class with default variables u', private variables v', and public variables w. We assume that both classes have the same method names and parameter and return types, and that each action defined in C is also defined in D. However, class D may have additional actions, called auxiliary actions, B. Let $I(u, v, w)$ and $J(u, u', v, v', w)$ be the class invariants of classes C and D respectively. Class C is refined by D through I and J, written $C \sqsubseteq_J^I D$, if following conditions hold:*

(a) Program Initialization: *When no objects exists, the invariants holds:*

$$C = \{\} \wedge D = \{\} \Rightarrow I \wedge J$$

(b) Object Creation: *The creation of a C object is refined by the creation of a D object:*

$$C.new \sqsubseteq^I_J D.new$$

(c) Method Refinement: *Every public method o_i of C is refined by the corresponding method in D:*

$$C.o_i \sqsubseteq^I_J D.o_i$$

Method Enabledness: *For every public method o_i in C, either the corresponding method of D or* some *action in D is enabled:*

$$I \wedge J \wedge en\ C.o_i \wedge tr\ C.o_i \Rightarrow (en\ D.o_i \vee en\ D.a \vee en\ D.b)$$

(d) Main Action Refinement: *Every action a_i of C is refined by the corresponding action in D:*

$$C.a_i \sqsubseteq^I_J D.a_i$$

Main Action Enabledness: *For every action a_i in C,* some *action in D is enabled:*

$$I \wedge J \wedge en\ C.a_i \wedge tr\ C.a_i \Rightarrow (en\ D.a \vee en\ D.b)$$

(e) Auxiliary Action Refinement: *Every new action b_i of D refines skip:*

$$skip \sqsubseteq^I_J D.b_i$$

Auxiliary Action Termination: *The computation of auxiliary actions terminates eventually:*

$$I \wedge J \Rightarrow wp(\mathbf{do}\ D.b\ \mathbf{od}\ , true)$$

where $\bigvee_i en\, D.a_i = en\, D.a$ *is the disjunction of enabledness domains of the main actions of* D, $\bigvee_i en\, D.b_i = en\, D.b$ *is the disjunction of enabledness domains of the auxiliary actions of* D *and* $\sqcap_i D.b_i = D.b$ *denotes the combined action for the auxiliary actions of* D.

The conditions for data refinement of action systems with procedures and invariants require that the invariants hold even before any instances of the classes are created. The creation of object instances preserve the invariants. Each public method and main action in C is data refined by the corresponding public method and main action of D. This means that the corresponding methods and actions of D have the same effect on the state space of C as the methods and actions of C. The auxiliary actions of D do not have any effect in the state space of C. Each method and action of D also preserves the invariants. The continuation condition for C implies the continuation condition for D. In other words, whenever the action system representing C terminates, the action system representing D terminates. Therefore, refinement increases the domain of termination. The auxiliary actions D do not have any effect in the state space of C, and their computation eventually terminates. So the auxiliary actions do not introduce non-termination in class D.

This definition of class refinement is based on the concept of the refinement of action systems with procedures, in [9, 10, 11] and the treatment of object identities in [27].

In general, a module can consist of several classes. It is possible to simultaneously refine more than one class. Consider for example,

module M
  // class definition for class C_1
  // class definition for class C_2
  // class definition for class D_1
  // class definition for class D_2
end

Module M contains class definitions for C_1, C_2, D_1 and D_2 where classes D_1 and D_2 simultaneously refine classes C_1 and C_2 respectively.

In the next chapter we present the design issues and rules for class inheritance in Lime that includes inheritance of actions and allows new methods to be added in the inheriting class.

Chapter 6

Inheritance of Actions

6.1 Rationale for Inheritance of Actions

In *Lime*, inheritance of classes using inherit clause establishes a subtype relationship between the child class and the parent class. One of the requirements for a subtype relation between the classes is that a child class object can be substituted for a parent class object.

When only syntactic conformance is taken into account, inheritance for subtyping is limited to matching the method signatures of the parent and child classes. For semantic conformance, subtyping by inheritance must include preservation of behavior of the objects in subtyping relation. A child class that is a subtype of a parent class must preserve the behavior of the parent class so that an instance of the child class can replace an instance of the parent class without any change in the observable behavior.

The goal of our research is to extend the design of class inheritance in *Lime* to include inheritance of actions. As the first step, we need to establish the necessity of inheritance of actions during class inheritance, so that a subtype relation holds between the child and parent classes.

6.1.1 To inherit actions or not

Let us consider a *Lime* class with only methods and no actions. The behavior of this class is expressed in terms of the behavior specifications of its methods.

In this case, class inheritance with *inherit* clause as specified in [18] is sufficient for ensuring behavior preservation from parent to child class.

In *Lime*, a class definition can include methods as well as actions. Part of the functionality of this class is achieved by the execution of enabled methods. The remaining part of the functionality is achieved by the autonomous execution of enabled actions. Therefore, behavior of this class is expressed in terms of the behavior specifications of its methods and actions. Class inheritance restricted to inheritance of methods alone, is not sufficient for ensuring behavior preservation from parent to child class. In that case, the child class inherits and preserves only the behavior specified by the methods of the parent class. Instead, the child class should be able to inherit and preserve the reactive as well as autonomous behavior of the parent class by inheriting methods as well as actions from the parent class.

To illustrate our point, we present the following example of a *Card*.

The class *Card* represents the membership card to a club. The class stores information on the cardholder, issue date, expiry date, and the status of the membership. For simplicity, we focus only on the expiry date and membership status. The variable *dtOfExpiry* represents the date of expiry for the card. The variable *status* represents the membership status for the cardholder. The *status* can be *valid*, *renew* (card membership needs to be renewed before expiry date) and *invalid* (when the membership has expired without being renewed by the expiry date).

It is assumed that the *Lime* program containing this class has access to a method that returns the current date and stores it in a global variable *Today*. The value *nullDate* is used as the nil value for Date variable which sets all the fields of the Date variable to zeros. Without specifying the implementation, it is assumed that two Date values can be assigned to each other, compared and subtracted. The result of an assignment operation is that individual fields of the *Date* (day, month and year) are assigned to the corresponding fields of the date on the right hand side of the assignment statement. The result of subtraction operation is the number of days between the two dates. A *Date* value can also be compared with the *nullDate*.

class *Card*
  public var *dtOfExpiry* : *Date*
  public var *status* : (*valid*, *renew*, *invalid*)
  public var *c* : *boolean*
  initialization *dtOfExpiry*, *status*, *c* := *NullDate*, *valid*, *false*
  public method *setExpiry*(*de* : *Date*)
    when $de \neq NullDate$ **do**
      dtOfExpiry := *de*
  public method *chkExpiry*
    when $dtOfExpiry \neq NullDate$ **do**
      c := *true*
  action *doCheck*
    when *c* **do**
      begin
        assert $Today \neq NullDate$;
        c := *false* ;
        if $dtOfExpiry < Today$ **then** *status* := *invalid*
        else if $dtOfExpiry - Today > 10$ **then** *status* := *valid*
        else *status* := *renew*
      end
end

Upon initialization, *dtOfExpiry* is set to *NullDate* and *status* is set to *valid*. The method *setExpiry* sets the date of expiry in the dtOfExpiry variable. It is enabled only if the date parameter to the method is not *NullDate*. Method *chkExpriy* is enabled only if *dtOfExpiry* is not *NullDate*, Method *chkExpiry* enables the action *doCheck*. Action *doCheck* first asserts that the variable *Today* does not contain a *NullDate*. Then *doCheck* sets the card's *status* to *valid*, *invalid* or *renew* according to the following rule – if *dtOfExpiry* is less than *Today* the *status* should be set to *invalid*; if *dtOfExpiry* is more than 10 days ahead of *Today* the *status* should be set to *valid*; otherwise if *dtOfExpiry* is somewhere between *Today* and the next 10 days, the *status* should be set to *renew.* For brevity, the implementation of the method renew is not specified here.

The class *CardIM* inherits from the class *Card* in order to add calculations for membership renewal fees. If the inheritance is limited to inheritance of methods only, then *doCheck* cannot be part of the definition of *CardIM.*

> **class** *CardIM* **inherit** *Card*
>   **public var** $fees, days : integer$
>   **initialization** $fees, days := 0, 0$
>   **public method** *chkExpiry*
>   **when** $dtOfExpiry \neq NullDate$ **do**
> $fees, c := 400, true$
> **end**

When *chkExpiry* is invoked on a *Card* object, it enables the action *doCheck* which computes the card's *status.* When *chkExpiry* is invoked on a *CardIM* object, the fees variable is set to the basic fees, but the *status* of the card is not computed. Therefore, if a *Card* object is replaced by a *CardIM* object, there is a change in the observable behavior. In this case, inheritance does not establish a subtype relationship.

Therefore inheritance of classes in *Lime* should also include inheritance of actions. The class *CardIMA* inherits the methods and actions of the class *Card* as follows:

> **class** *CardIMA* **inherit** *Card*
>   **public var** $fees, days : integer$
>   **initialization** $fees, days := 0, 0$
>   **public method** *chkExpiry*
>   **when** $dtOfExpiry \neq NullDate$ **do**
> $fees, c := 400, true$
>   **action** *doCheck*
>   **when** $fees = 400 \wedge c$ **do**
>   **begin**

assert $Today \neq NullDate$;
$c := false$;
if $dtOfExpiry < Today$ **then** $status, fees := invalid, fees + 15$
else
  begin
    $days := dtOfExpiry - Today$;
    if $days > 10$ **then** $status, fees := valid, 0$
    else $status, fees := renew, fees - days$
  end
end
end

The class *CardIMA* adds a variable *fees* which stores the amount of membership fees for the cardholder. Since methods can be inherited, class *CardIMA* inherits and overrides the method *chkExpiry* from the class *Card*. It is enabled if *dtOfExpiry* is not *NullDate*, and sets the basic fees for one year's membership to 400 dollars and also enables the action *doCheck*.

Action *doCheck* asserts that the variable *Today* does not contain a *NullDate*. It sets the cardholder's status according to the same rules as in the class *Card*. Action *doCheck* also calculates the amount of fees according to the status - if status is 'invalid' then the cardholder has to pay an additional fine of 15 dollars; if the status is 'renew' then the cardholder has to pay the annual fee less an amount proportional to the number of days left to expiry; otherwise if the status is valid then the cardholder does not have to renew or pay any fees.

6.1.2 To override actions or not

Let us again consider the classes *Card* and *CardIMA* from the previous section. The class *CardIMA* introduces membership fees as an additional variable. The methods and actions of *CardIMA* should implement additional functionality to calculate the membership fees while preserving the behavior of the corresponding methods and actions of *Card*. The method *chkExpiry* in *CardIMA* overrides the corresponding parent class method to set the value of *fees* to 400. Similarly, the action *doCheck* overrides the corresponding parent class action to add computation of *fees*.

We give the following example to illustrate the necessity for overriding actions when the child class action provides an alternate implementation.

Class *SumSquare* takes two positive integers a and b and calculates the square of their sum as result $= a^2 + b^2 + 2ab$. Class *sumSquareR* inherits from *SumSquare* and provides an alternate (more efficient) way of calculating the square of their sum as result $= (a + b)^2$.

class *SumSquare*
  public var a, b, ss : *integer*
  public var r, f : *boolean*
  initialization $a, b, ss, r, f := 0, 0, 0, \mathit{false}, \mathit{false}$
  public method *setAB*(c, d : *integer*)
    begin
      assert $\{c > 0 \land d > 0\}$;
      $a, b := c, d$
    end
  public method *calcSumSquare*
    $ss, r := 0, \mathit{true}$
  public method *getSumSquare*(**res** *result* : *integer*)
    when f **do**
      $\mathit{result}, f := ss, \mathit{false}$
  action *doSumSquare*
    when r **do**
      $ss, r, f := a^2 + b^2 + 2 * a * b, \mathit{false}, \mathit{true}$
end

Class SumSquareR inherits and overrides the action *doSumSquare* as follows:

class *SumSquareR* **inherit** *SumSquare*
  action *doSumSquare*
    when r **do**
      $ss, r, f := (a + b)^2, \mathit{false}, \mathit{true}$

end

Therefore, a child class action should be able to override the corresponding parent class action when –

(1) the child class action implements additional functionality or

(2) the child class action provides alternate implementation.

6.2 Role of Action Name and Guard

For each inherited method, the name and type signature of the parent class method and the corresponding child class method must match. The method names should match because methods are identified and invoked by their name. The type signature of the methods should match, otherwise it would be a case of overloading and not overriding.

Actions, on the other hand, do not take any parameters and are not invoked by name. However, action names are still useful for distinguishing between overridden actions and newly added actions. Class inheritance also needs to identify the corresponding child class action for each inherited parent class action. Therefore, for each inherited action, the name of the parent class action and the corresponding child class action must match.

An action can execute autonomously when its guard evaluates to true. In order to preserve the observable behavior of the parent class, the guard of a child class action must establish a specific relationship with the guard of the corresponding parent class action. For now, we state that the guard can be strengthened in the child class action. We will establish the details of the relationship between the two guards during the design of the class refinement rules.

6.3 Subclass Action and Superclass Action

When a child class action overrides the corresponding parent class action it preserves the behavior of the parent class action while, possibly, providing additional functionality. In a child class the overridden action should replace the corresponding parent class action. If the child class does not specify a corresponding action, the parent class action is used instead.

In Lime, a parent class method can be invoked from within the body of the corresponding child class method. This is achieved by a *super.mtdName* call in the child class method. The child class method can then specify additional functionality to augment the behavior of the parent class method.

Similarly, a child class action can override and augment the behavior of a parent class action. However, if the parent class action accesses private variable of the class, then the child class action cannot override to augment and duplicate the behavior of the parent class action as it cannot access private members of the parent class. If the child class action could invoke the parent class action, it would solve this problem. However, actions cannot be invoked. Therefore we present a construct $\oplus$ (*'fusion'* operator) which is specified as follows:

If the parent class action aa is of the form **when** $g1$ **do** $S1$ and the corresponding child class action aa is of the form **when** $g2$ **do** $S2$ where $S2$ is the additional functionality specified in the child class action then the fusion of the two actions is defined as

$$\mathit{super.aa} \oplus \mathit{this.aa} \equiv \textbf{when } g1 \wedge g2 \textbf{ do } S1 \ ; \ S2$$

.

Why do we choose to implement the body of the fusion action as $S1 \ ; \ S2$ and not as $S1 \sqcap S2$? Consider the case when $S1$ and $S2$ may be composed of one or more method calls. Then $S1 \sqcap S2$ is equivalent to putting the constituent method calls in a parallel composition. Since actions in *Lime* are atomic up to method calls, in the implementation such a parallel composition of method calls is translated into sequential composition of the method calls. Therefore, we choose to implement the body of the fusion action as $S1 \ ; \ S2$.
Let us consider the following example,

class A

```
  private var a : integer
  initialization a := 1
  action aa
    when a > 0 do
      begin
        a := a + 1 ; x.p()
      end
end
class B inherit A
  private var b : integer
  initialization b := 1
  invariant a = b
  action aa
    when b > 0 do
      begin
        //super.aa ;
        b := b + 1 ; y.q()
      end
end
```

As the parent class action *aa* cannot be invoked as *super.aa*, so class B can be rewritten using the $\oplus$ operator as:

```
class B inherit A
  private var b : integer
  initialization b := 1
  invariant a = b
  action aa ⊕ super.aa
    when b > 0 do
      begin
        b := b + 1 ; y.q()
      end
end
```

The action aa $\oplus$ super.aa is implemented as

$$
\begin{array}{l}
\textbf{when } b > 0 \land a > 0 \textbf{ do} \\
\quad \textbf{begin} \\
\quad\quad a := a + 1 \ ; \ x.p() \ ; \ b := b + 1 \ ; \ y.q() \\
\quad \textbf{end}
\end{array}
$$

6.4 Visibility Rules for Actions

In *Lime*, visibility rules are applied to variables and methods in order to specify access control. The two access modifiers used now are: *public* and *private*. A private method can be invoked only from within the class itself and a public method can be invoked from a class, its subclasses and their objects. A method declared without any access modifiers is a *default* method. A default method is visible to all members of a module but it is not visible outside of the module.

Since actions execute autonomously, the usual meaning of these access modifiers cannot apply to actions. We categorize actions into *final* and *public* actions. A *final* action is defined as an action that can be inherited but cannot be overridden. A *public* action is defined as an action that can be inherited and overridden. By default, actions in *Lime* are public. One scenario when an action can be declared as *final* is when the action refers to *private* variables or invokes a *private method* of the class. Another application of *final* actions is to ensure that a critical behavior does not change from parent to child class while still being available to the child class.

In *Lime*, classes are translated into modules to give the classes an action system representation. In modules, either a member (variable or procedure) can be exported (*public* member) or not (*private* member). However, in its current syntax, modules cannot export a member for a specific purpose, as in the case of *protected* members. Therefore, we longer support *protected* variables or *protected* methods.

The syntax of *Lime* is extended to include specification of access modifiers (This is not the full syntax for *Lime*; we have only shown that part of the syntax that has been affected by introduction of access modifiers). Also included in the syntax are the inheritance clause:

class	::=	**class** *identifier* { **extend** *identifier* |
		inherit *identifier* | **implement** *identifier* }
		{ *attribute* | *initialization* | *method* | *action* } **end**
attribute	::=	[*accessMod*] **var** *variableList*
initialization	::=	**initialization** [(*variableList*)] *statement*
method	::=	[*accessMod*] **method** *identifier* [(*variableList*
		[, **res** *variableList*])] [**when** *expression* **do**]
		statement
action	::=	[*accessModAction*] **action** *identifier*
		[**when** *expression* **do**] *statement*
accessMod	::=	**public** | **private**
accessModAction	::=	**public** | **final**

6.5 Module Representation of Class with Inherited Actions

Since we have extended class inheritance in *Lime* to include inheritance of actions, the translation of classes into modules must account for inherited actions. Consider a class D that inherits some of the methods and actions of a class C. The class definition for the classes C and D are:

class C
  var $u : U$
  private var $v : V$
  public var $w : W$
  invariant I
  initialization K
  method $p_1\ P_1$
  $\ldots$
  public method $m_1\ M_1$
  $\ldots$
  public method $o_1\ O_1$
  $\ldots$
  action $s_1\ S_1$
  $\ldots$
  action $t_1\ T_1$
  $\ldots$
end

class D **inherit** C
  var $u' : U'$
  private var $v' : V'$
  invariant J
  initialization K'
  public method $o_1\ O_1'$
  $\ldots$
  public method $n_1\ N_1$
  $\ldots$
  action $t_1\ T_1'$
  $\ldots$
  action $b_1\ B_1$
  $\ldots$
end

Class D may have its own *private* and *default* methods. However, we are concerned with the *public* methods as D can only inherit *public* methods of C.

The class definition for C within a module M amounts to following module declarations:

private var C : **set of** $Object := \{\}$
var $u : Object \rightarrow U$
private var $v : Object \rightarrow V$
public var $w : Object \rightarrow W$
invariant $(\forall this \in C \bullet I)$
procedure $C.new($**res** $this : Object)$
  $this :\notin C \cup \{nil\}\ ;\ C := C \cup \{this\}\ ;\ K$
procedure $C.p_1(this : Object)$
  $\{this \in C\}\ ;\ P_1$

$\dots$

public procedure $C.m_1(\mathit{this} : \mathit{Object})$
$\{\mathit{this} \in C\}\ ;\ M_1$

$\dots$

public procedure $C.o_1(\mathit{this} : \mathit{Object})$
$\{\mathit{this} \in C\}\ ;\ O_1$

$\dots$

action $C.s_1$
var $\mathit{this} :\in C \bullet S_1$

$\dots$

action $C.t_1$
var $\mathit{this} :\in C \bullet T_1$

$\dots$

The class definition for D within the module M amounts to following module declarations:

private var D : **set of** $\mathit{Object} := \{\}$
var $u' : \mathit{Object} \to U'$
private var $v' : \mathit{Object} \to V'$
public var $w : \mathit{Object} \to W$
invariant $(\forall \mathit{this} \in D \bullet J)$
procedure $D.\mathit{new}($**res** $\mathit{this} : \mathit{Object})$
$C.\mathit{new}(\mathit{this})\ ;\ D := D \cup \{\mathit{this}\}\ ;\ K'$
procedure $D.p_1(\mathit{this} : \mathit{Object})$
$\{\mathit{this} \in D\}\ ;\ C.P_1$

$\dots$

public procedure $D.m_1(\mathit{this} : \mathit{Object})$
$\{\mathit{this} \in D\}\ ;\ C.m_1$

$\dots$

public procedure $D.o_1(\mathit{this} : \mathit{Object})$
$\{\mathit{this} \in D\}\ ;\ O_1'$

$\dots$

public procedure $D.n_1(\mathit{this} : \mathit{Object})$

$\{this \in D\}$; N_1

...

action $D.s_1$

var $this :\in C \bullet S_1$

...

action $D.t_1$

var $this :\in D \bullet T_1'$

...

action $D.b_1$

var $this :\in D \bullet B_1$

...

In the module declarations, every time an object of class D is created, the object is added to the set of objects D as well as to the set of objects C. The set of objects C contains all objects that have been created as instances of class C along with all objects that have been created as instances of class D. On the other hand, the set of objects D contains only those objects that have been created as instances of class D. Therefore, the sets of objects satisfy the relation: $C \supseteq D$.

Let C' be a subset of C such that C' contains only those objects that have been created as instances of class D. Then $C' = D$. Class D is now defined in terms of C' as:

private var D : **set of** $Object := \{\}$
private var C' : **set of** $Object := \{\}$
var $u' : Object \rightarrow U'$
private var $v' : Object \rightarrow V'$
public var $w : Object \rightarrow W$
invariant $(\forall this \in D \bullet J)$
procedure $D.new($**res** $this : Object)$
$C.new(this)$; $C' := C' \cup \{this\}$;
$D := D \cup \{this\}$; K'
procedure $D.p_1(this : Object)$

$\{this \in D\}$; $C'.P_1$

...

public procedure $D.m_1(this : Object)$

$\{this \in D\}$; $C'.m_1$

...

public procedure $D.o_1(this : Object)$

$\{this \in D\}$; O'_1

...

public procedure $D.n_1(this : Object)$

$\{this \in D\}$; N_1

...

action $D.s_1$

var $this :\in C' \bullet S_1$

...

action $D.t_1$

var $this :\in D \bullet T'_1$

...

action $D.b_1$

var $this :\in D \bullet B_1$

...

Inherited Class with *super* Calls : A class D inherits from the class C such that D contains super calls for methods or actions.

class C	**class** D **inherit** C
var $u : U$	**var** $u' : U'$
private var $v : V$	**private var** $v' : V'$
public var $w : W$	**invariant** J
invariant I	**initialization** K'
initialization K	**method** $p_1\ P_1'$
method $p_1\ P_1$	...
...	**public method** o_1
public method $m_1\ M_1$	**begin** $super.o_1$; O_1' **end**
...	...
public method $o_1\ O_1$	**public method** $n_1\ N_1$
...	...
action $s_1\ S_1$	**action** $t_1 \oplus super.t_1$
...	T_1'
action $t_1\ T_1$	...
...	**action** $b_1\ B_1$
end	...
	end

The class definition of class D within a module amounts to the module declarations:

private var D : **set of** $Object := \{\}$
private var C' : **set of** $Object := \{\}$
var $u' : Object \to U'$
private var $v' : Object \to V'$
public var $w : Object \to W$
invariant $(\forall this \in D \bullet J)$
procedure $D.new($**res** $this : Object)$
  $C.new(this)$; $C' := C' \cup \{this\}$;
  $D := D \cup \{this\}$; K'
public procedure $D.m_1(this : Object)$

$\{this \in D\}$; $C'.m_1$

...

public procedure $D.o_1(this : Object)$

$\{this \in D\}$; $C'.o_1$; O'_1

...

public procedure $D.n_1(this : Object)$

$\{this \in D\}$; N_1

...

action $D.s_1$

var $this :\in C' \bullet S_1$

...

action $D.t_1$

var $this :\in D \bullet T_1$; T'_1

...

action $D.b_1$

var $this :\in D \bullet B_1$

...

6.6 Class Inheritance and Class Refinement

From a syntactic point of view, subtyping involves matching the method signatures of the parent and child classes. However, substitutability of objects must take into account the behavior of the objects. The subtype relation between the parent and child classes should be such that the child class must imitate the behavior of the parent class [12]. From a semantic point of view, subtyping requires that a parent class object can be replaced by a child class object without any change in the observable behavior. This requirement is addressed by semantic conformance of behavior during subtyping in [19, 6]. This form of subtyping while preserving behavior includes matching the type signatures as well as the behavior specifications of the parent and child classes.

Both class inheritance and refinement appeal to the notion of child class or refined class being able to replace parent class or original (less refined) class while preserving the behavior of the class it is inheriting from or refining.

We have presented the various design features of class inheritance with actions that establishes a subtype relationship between the child class and its parent class. The child class is guaranteed to preserve the behavior of the corresponding parent class so that an instance of parent class can be replaced by an instance of the child class.

Taking it one step further, if we restrict class inheritance so that it preserves total correctness, then the child class will also be a refinement of the parent class. In the total correctness preserving form, class inheritance establishes a *subtype* as well as a *refinement* relationship between the child class and its parent class.

In order to establish the conditions under which parent class C is refined by child class D, we translate the *Lime* classes into action systems with procedures. This translation into action systems is achieved by declaring the classes C and D in a module M. We can then reason about refinement of class C by class D in terms of refinement of the module representation of class C by the module representation of class D within module M.

In the next section we present the class refinement rules for inheritance of classes in *Lime*. If each of the C' objects and D objects satisfy these rules for class refinement rules for inheritance, then a D object can replace a C object without any change in observable behavior. Therefore, we can conclude that class C is refined by class D; also class D is a subtype of class C.

6.7 Class Refinement Rules for Inheritance

Superposition refinement of action systems requires that the global state space of the action systems remain unchanged and that the local state space of the concrete action system expands the local state space of the abstract action system with new local variables. In other words, if the action system $\mathcal{A}'$ refines the action system $\mathcal{A}$ under the conditions of superposition refinement, then $\mathcal{A}'$ has all the local variables x of $\mathcal{A}$ and additionally $\mathcal{A}'$ has new local variables x'.

In case of class inheritance, we require that the global state space of the parent and child classes remain unchanged. This condition is satisfied as the child class inherits the public variables of the parent class. The child class can refer to the default variables of the parent class as both the classes are defined in the same module. The child class operates on the same local state space as the parent class and expands it by adding new local variables (private and default variables). Therefore, class refinement for class inheritance in *Lime* is verified under the conditions of superposition refinement of modules as presented in section 5.1.3.

Definition 6.1 (Class Refinement for Inheritance). *Let C be a class with default variables u, private variables v and public variables w. Let D be a class with default variables u', private variables v', and class D inherits from class C. C operates on the state space determined by u, v, w. D operates on the state space determined by the variables u, u', v, v', w, either directly or indirectly through superclass method calls. Class D inherits all public variables w, all public methods, M, and all actions, A, from class C. We assume that the public methods in both classes have the same method names and parameters and return types. Class D may have additional actions, called auxiliary actions, $auxA$, and new methods, $newM$. Let $I(u, v, w)$ be the invariant of class C and $J(u, u', v, v', w)$ be the invariant of class D. Class C is refined by D through I and J, written $C \leq_J^I D$, if following conditions hold:*

(a) Program Initialization: $C = \{\} \wedge D = \{\} \Rightarrow I \wedge J$

(b) Object Creation: $C.new \leq_J^I D.new$

For every state in the statespace after creation of C and D objects,

(c) Main Method Refinement: $C.M_i \leq_J^I D.M_i$

Main Method Enabledness:

$I \wedge J \wedge en\ C.M_i \wedge tr\ C.M_i \Rightarrow (en\ D.M_i \vee en\ D.A \vee en\ D.auxA)$

(d) New Method Refinement: $\{I \wedge J\}\, D.newM_i\, \{I \wedge J\} \equiv D.newM_i \leq_J^I D.newM_i$

(e) Main Action Refinement: $C.A_i \leq_J^I D.A_i$

Main Action Enabledness: $I \wedge J \wedge en\, C.A_i \wedge tr\, C.A_i \Rightarrow en\, D.A \vee en\, D.auxA$

(f) Auxiliary Action Refinement: $skip \leq^I_J D.auxA_i$

Auxiliary Action Termination: $I \wedge J \Rightarrow wp(\textbf{do}\ D.auxA\ \textbf{od}\ , true)$

where $\bigvee_i en\, D.A_i = en\, D.A$ *is the disjunction of enabledness domains of the main actions of* D, $\bigvee_i en\, D.auxA_i = en\, D.auxA$ *is the disjunction of enabledness domains of the auxiliary actions of* D *and* $\sqcap_i D.auxA_i = D.auxA$ *denotes the combined action for the auxiliary actions of* D. □

Condition (a) states that when no objects exist, the invariants of C and D hold. Condition (b) states that creation of a C object is refined by creation of a D object. First part of condition (c) states that every public method M_i of C is refined by the corresponding method in D. Second part of condition (c) states that for every public method M_i in C, either the corresponding method of D or *some* action in D is enabled. Condition (d) states that every new method $newM_i$ of D preserves the invariants of C and D. First part of condition (e) states that every action A_i of C is refined by the corresponding action in D. Second part of condition (e) states that for every action A_i in C, *some* action in D is enabled. First part of condition (f) states that every new action $auxA_i$ of D refines *skip*. Second part of condition (f) states that the computation of auxiliary actions terminates eventually.

As discussed in section 6.5, let C' be a subset of C such that C' contains only those objects that have been created as instances of class D. In other words, C' considers only those objects that participate in the inheritance relationship between C and D classes. Then, the refinement can be more precisely established between the classes C and D as:

$$C \leq^{I(C,u,v,w,z,f)}_{J(C,D,u,u',v,v',w,z,f)} D \equiv C \leq^{I(C',u,v,w,z,f)}_{J(C',D,u,u',v,v',w,z,f)} D \tag{6.1}$$

Condition (b) on object creation does not include a check for enabledness, as we assume that initializations are always enabled: the syntactic structure of initializations does not allow for guards.

Condition (f) implies that the auxiliary actions are *stuttering* actions: they refine *skip*, they do not cause any state change in C. The auxiliary actions eventually terminate, they do not introduce (observable) divergence.

This definition of class refinement is based on the concept of the refinement of action systems with procedures, in [9, 10, 11] and the treatment of object identities in [27].

6.8 Discussion

The conditions for superposition refinement of action systems with procedures and invariants require that the invariants hold even before any instances of the classes are created. The creation of object instances preserve the invariants. Each of the methods and actions of class C that are inherited, are refined by the corresponding methods and actions of class D. The inherited (and possibly overridden) methods and actions of child class D preserve the behavior of the corresponding methods and actions of the parent class C. Each new method in D must preserve the invariants of C and D. This ensures that the new methods do not introduce any inconsistencies in behavior in the presence of subtype aliasing and when the objects are shared by multiple users. This is based on the notion of consistent methods of [6]. The auxiliary actions in the child class D act as *skip* statement in the state space of the parent class C. The execution of the auxiliary actions must terminate. Thus auxiliary actions do not introduce non-termination, when there was no pre-existing divergence in the parent class. When all these conditions are satisfied, the child class D is a subtype as well as a refinement of the parent class C.

In Appendix A we present a simple *Lime* examples with class inheritance and its proof of correctness as a refinement step.

Chapter 7

Lime Examples

In this chapter, we present a small collection of *Lime* programs to illustrate the features of the language. In particular, we highlight the use of class refinement and class inheritance including inheritance of actions. Each of the following three sections contains one *Lime* example with explanation.

7.1 Food Court

In this example we model the behavior of a food court. This example is motivated by the Ticket Algorithm. In the abstract implementation, the food court has NSHOP number of shops and NCUST number of customers. When customers enter the food court they choose one of the shops in the food court. The customer is then added to the chosen shop's list of customers. At any point in time, the customers are served in no particular order. A shop has MAX number of customers. The shop is in *busy* state as long as there are customers waiting to be served, otherwise the shop is *idle*. The customer can be in three states: *entered*, when the customer enters the food court, *waiting*, when the customer has chosen a shop and is waiting to be served, and *served*,when the customer has been served by the chosen shop.

class *Customer*
  var *s* : *Shop*

```
    var state : (entered, waiting, served)
    invariant Inv_C
    initialization
      s, state := nil, entered
    public method getState(res st : (entered, waiting, served))
      st := state
    public method chooseShop(sh : Shop)
      when sh ≠ nil do
        begin
          s := sh ;
          s.addCustomer(this) ;
          state := waiting
        end
    action getServed
      when state = waiting do
        state := served
end

class Shop
    var noC : integer
    var state : (idle, busy)
    var n : integer
    var C : array MAX of Customer
    invariant Inv_S
    initialization
      begin
        noC, n, state := 0, 0, idle ;
        while n < MAX do
          C[n], n := nil, n + 1 ;
      end
    public method addCustomer(c : Customer)
      begin
        assert c ≠ nil ;
```

```
        state, C[noC], noC := busy, c, noC + 1 ;
      end
    action checkState
      when state = busy do
        var j : integer
        var finished : boolean
        begin
          j, finished := 1, true ;
          while j ≤ noC do
            finished, j := finished ∧ C[j − 1].getState() = served, j + 1;
          if finished then state := idle
        end
end


// Main Program : FOOD COURT
var Sh : array NSHOP of Shop
var Cust : array NCUST of Customer
var m, p, j : integer
invariant Inv_P
begin
  m, p := 0, 0 ;
  while m < NSHOP do
    begin
      Sh[m] := new Shop ;
      m := m + 1
    end
  while p < NCUST do
    begin
      Cust[p] := new Customer ;
      j := rand(0, NSHOP − 1) ;
      assert Sh[j] ≠ nil ;
      Cust[p].chooseShop(Sh[j]) ;
```

$p := p + 1$
end
end

The class *CustomerI* inherits from the class *Customer* and the class *ShopI* inherits from the class *Shop*. In this more concrete implementation, when the customer chooses a shop, the customer gets a token from the shop that is held in the *turn* variable of *CustomerI* class. The customer is served when the token held by the customer matches the next token to be served by the shop. The main program now uses the classes *CustomerI* and *ShopI* instead of *Customer* and *Shop*.

class *CustomerI* **inherit** *Customer*
var $turn, next : integer$
invariant Inv_{CI}
initialization
// Here implicitly, the initialization of class Customer is called first.
$turn, next := 0, -1$
action *getServed*
when $turn = next$ **do**
begin
$state := served$;
$s.updateNext$
end
end

class *ShopI* **inherit** *Shop*
var $number, next : integer$
invariant Inv_{SI}
initialization
// Here implicitly, the initialization of class Shop is called first.
$number, next := 1, 1$
method $addCustomer(c : Customer)$

var cI : $CustomerI$
begin
assert $c \neq nil$;
if c **is** $CustomerI$ **then**
begin
$cI := c$;
$cI.turn, cI.next, number := number, next, number + 1$;
$C[noC], noC, state := cI, noC + 1, busy$
end
end
method $updateNext$
var i : $integer$
begin
$i, next := 0, next + 1$;
while $i < noC$ **do**
$C[i].next := next$
end
action $checkState$
when $state = busy \wedge next = number \wedge next > 1$ **do**
var j : $integer$
var $finished$: $boolean$
begin
$j, finished := 1, true$;
while $j < noC$ **do**
$j, finished := j + 1, finished \wedge C[j-1].getState() = served$;
if $finished$ **then** $state := idle$
end
end

// Main Program : FOOD COURT
var Sh : **array** $NSHOP$ **of** $Shop$
var $Cust$: **array** $NCUST$ **of** $Customer$

var $m, p, j : integer$
invariant Inv_{PI}
begin
  $m, p := 0, 0$;
  while $m < NSHOP$ **do**
    begin
      $Sh[m] :=$ **new** $ShopI$;
      $m := m + 1$
    end
  while $p < NCUST$ **do**
    begin
      $Cust[p] :=$ **new** $CustomerI$;
      $j := rand(0, NSHOP - 1)$;
      assert $Sh[j] \neq nil$;
      $Cust[p].chooseShop(Sh[j])$;
      $p := p + 1$
    end
end

The *default* variables of a parent class are visible to the child class. To distinguish between the values assigned to the *default* variables in an instance of the parent class and in an instance of the child class, we rename the *default* variables as follows:

For an instance of the parent class, each *default* variable names *vName* is written as $vName_0$. For an instance of the child class, each *default* variable names *vName* is written as $vName_1$. This naming scheme is used only for specifying invariants.

The invariants Inv_C, Inv_S, and Inv_P for the classes *Customer*, *Shop* and the corresponding main program respectively are given as:

$$Inv_C : state_0 = waiting \Rightarrow s_0 \neq nil \wedge this \in s_0.C_0$$
$$Inv_S : (state_0 = busy \Rightarrow (\forall k \bullet 0 \leq k < noC_0 \Rightarrow C_0[k] \neq nil)) \wedge$$
$$(state_0 = idle \Rightarrow (\forall h \bullet 0 \leq h < noC_0 \Rightarrow C_0[h].state_0 = served))$$
$$Inv_P : (\forall p \in Cust \bullet p.state_0 = waiting \Rightarrow p.s_0 \neq nil \wedge p \in p.s_0.C_0) \wedge$$

$(\forall q \in Sh \bullet (q.state_0 = busy \Rightarrow (\forall r \bullet 0 \leq r < q.noC_0 \Rightarrow q.C_0[r]$
$\neq nil)) \wedge (q.state_0 = idle \Rightarrow (\forall t \bullet 0 \leq t < q.noC_0 \Rightarrow$
$q.C_0[t].state_0 = served)))$

The invariants Inv_{CI}, Inv_{SI}, and Inv_{PI} for the classes *CustomerI*, *ShopI* and the corresponding main program respectively are given as:

$$\begin{aligned}
Inv_{CI} :\ & (s_0 = s_1) \wedge (state_0 = state_1) \wedge \\
& (state_1 = waiting \Rightarrow s_1 \neq nil \wedge this \in s_1.C_1) \wedge \\
& (state_1 = served \Rightarrow turn_1 \leq s_1.next) \\
Inv_{SI} :\ & (noC_0 = noC_1) \wedge (state_0 = state_1) \wedge (n_0 = n_1) \wedge (C_0 = C_1) \wedge \\
& (state_1 = busy \Rightarrow (\forall k \bullet 0 \leq k < noC_1 \Rightarrow C_1[k] \neq nil)) \wedge \\
& (state_1 = idle \Rightarrow (\forall h \bullet 0 \leq h < noC_1 \Rightarrow C_1[h].state_1 = served)) \\
& \wedge (\forall u, v \bullet 0 \leq u, v < noC_1 \wedge u < v \Rightarrow C_1[u].turn_1 < C_1[v].turn_1) \\
Inv_{PI} :\ & (\forall p \in Cust \bullet (p.s_0 = p.s_1) \wedge (p.state_0 = p.state_1) \wedge \\
& (p.state_1 = waiting \Rightarrow p.s_1 \neq nil \wedge p \in p.s_1.C_1) \wedge \\
& (p.state_1 = served \Rightarrow p.turn_1 \leq p.s_1.next)) \wedge \\
& (\forall q \in Sh \bullet (q.noC_0 = q.noC_1) \wedge (q.state_0 = q.state_1) \wedge (q.n_0 = \\
& q.n_1) \wedge (q.C_0 = q.C_1) \wedge (q.state_1 = busy \Rightarrow (\forall k \bullet 0 \leq k < \\
& q.noC_1 \Rightarrow q.C_1[k] \neq nil)) \wedge (q.state_1 = idle \Rightarrow (\forall h \bullet 0 \leq h < \\
& q.noC_1 \Rightarrow q.C_1[h].state_1 = served)) \wedge (\forall u, v \bullet 0 \leq u, v < q.noC_1 \\
& \wedge u < v \Rightarrow q.C_1[u].turn_1 < q.C_1[v].turn_1))
\end{aligned}$$

In this example, the class *Customer* is refined by the class *CustomerI*. The refinement invariant R_C for this refinement is given by $R_C \equiv Inv_C \wedge Inv_{CI}$. Similarly, the class *Shop* is refined by the class *ShopI*. The refinement invariant R_S for this refinement is given by $R_S \equiv Inv_S \wedge Inv_{SI}$.

7.2 Collection of Elements

In this example we start with an abstract implementation of a *Bag* into which elements can be inserted and deleted by the *add* and *remove* operations respectively. The bag also offers the *isEmpty*, *hasMember* and *getTotal* operations. The *isEmpty* operation returns *true* if the bag doesn't have any elements, *false* otherwise. The *hasMember* operation checks if a given element is a member of the bag. The *getTotal* operation returns the sum of all elements of the bag.

class *Bag*
  var b : **bag of** *integer*
  var *sum* : *integer*
  invariant Inv_{Bag}
  initialization
    $b, sum := [], 0$
  public method *isEmpty*(**res** r : *boolean*)
  $r := b = []$
  public method *add*(e : *integer*)
    $b, sum := b + [e], sum + e$
  public method *remove*(e : *integer*)
    var r : *boolean*
    begin
      hasMember(e, r) ;
      if r **then**
        $b, sum := b - [e], sum - e$
    end
  public method *hasMember*(e : *integer*, **res** *found* : *boolean*)
    $found := e \in b$
  public method *getTotal*(**res** s : *integer*)
    $s := sum$
end

The invariant Inv_{Bag} for the class *Bag* is given as:

$$Inv_{Bag} : sum = \Sigma e \in b \bullet e$$

As a refinement of the class *Bag*, we present the class *Tree* that supports inserting an element, deleting an element and sum of all elements by the operations *add*, *remove* and *getTotal* respectively. Class *Tree* also offers the operations *isEmpty* and *hasMember* for checking, respectively, if the tree is empty and if the given element is a member of the tree. The sum of all elements is obtained by traversing the tree and adding all the elements encountered. We first define the class *Node* which serves as the building block for *Tree*. The algorithm for class *Tree* is based on the binary search tree algorithm from [14].

```
class Node
  var lNode, rNode, pNode : Node
  var data, d : integer
  var state : (idle, adding, searching, deleting)
  initialization (e : integer)
    lNode, rNode, pNode, data, d, state := nil, nil, nil, e, 0, idle
  public method add(e : integer)
    when state = idle do
      begin
        if e ≤ data ∧ lNode = nil then
          begin
            lNode := new Node(e) ;
            lNode.pNode := this
          end
        else if e > data ∧ rNode = nil then
          begin
            rNode := new Node(e) ;
            rNode.pNode := this
          end
        else
          state, d := adding, e
      end
  action doAddElement
    when state = adding do
      begin
        state := idle ;
        if d ≤ data then
          lNode.add(d)
```

```
        else
          rNode.add(d)
      end
  public method search(e : integer, res n : Node)
    when state = idle do
      begin
        if e = data then
          n := this
        else if e < data ∧ lNode ≠ nil then
          begin
            state := searching ;
            lNode.search(e, n)
          end
        else if e > data ∧ rNode ≠ nil then
          begin
            state := searching ;
            rNode.search(e, n)
          end
        else
          n := nil
        state := idle
      end
  public method next(res n : Node)
    var curNode : Node
    begin
      curNode := this ;
      if curNode.rNode ≠ nil then
        begin
          n := curNode.rNode ;
          while n.lNode ≠ nil do
            n := n.lNode
        end
      else
```

```
            begin
              n := curNode.pNode ;
              while n ≠ nil ∧ curNode = n.rNode do
                curNode, n := n, n.pNode
            end
        end
      public method calcTotal(res s : integer)
        var l, r : integer
        begin
          l, r, s := 0, 0, data ;
          if lNode ≠ nil then
            begin
              lNode.calcTotal(l) ;
              s := s + l
            end
          if rNode ≠ nil then
            begin
              rNode.calcTotal(r) ;
              s := s + r
            end
        end
end

class Tree
  public var root : Node
  invariant Inv_Tree
  initialization
    root := nil
  public method isEmpty(res r : boolean)
    r := (root = nil)
  public method add(e : integer)
    if root = nil then
      root := new Node(e)
```

```
        else
          root.add(e)
      public method remove(e : integer)
        when root ≠ nil do
          var r, x, y : Node
          begin
            r := nil ;
            root.search(e, r) ;
            if r ≠ nil then
              begin
                r.state := deleting ;
                if r.lNode = nil ∨ r.rNode = nil then
                  y := r
                else
                  r.next(y)
                if y.lNode ≠ nil then
                  x := y.lNode
                else
                  x := y.rNode
                if x ≠ nil then
                  x.pNode := y.pNode
                if y.pNode = nil then
                  root := x
                else
                  if y = y.pNode.lNode then
                    y.pNode.lNode := x
                  else
                    y.pNode.rNode := x
                if y ≠ r then
                  r.data := y.data
                r.state := idle
              end
          end
```

public method *hasMember*(e : *integer*, **res** *found* : *boolean*)
  var n : *Node*
  begin
    $n :=$ *nil* ;
    if *root* = *nil* **then**
      found := *false*
    else
      begin
        root.search(e, n) ;
        found := $n \neq$ *nil*
      end
  end
public method *getTotal*(**res** s : *integer*)
  begin
    $s := 0$;
    if *root* $\neq$ *nil* **then**
      root.calcTotal(s)
  end
end

As a second refinement, each node in a tree stores the sum of the subtree rooted at that node. Therefore, instead of traversing the entire tree to calculate the sum, we just need to retrieve the sum stored at the root node. Class *TreeST* inherits from the class *Tree* and the class *NodeST* inherits from the class *Node*. The class *NodeST* defines a variable *subtotal* that stores the sum of the subtree rooted at that node. Class *TreeST* calculates the sum by retrieving the value for *root.subtotal*. The class *NodeST* serves as the building block for *TreeST*. Class *Node* is refined by *NodeST* and the class *Tree* is refined by *TreeST*.

class *NodeST* **inherit** *Node*
  var *subtotal* : *integer*
  initialization (e : *integer*)
  // Here implicitly, the initialization of class Node is called first.

$subtotal := e$
public method $add(e : integer)$
  when $state = idle$ **do**
    var $x : NodeST$
    begin
      $x := nil$;
      if $e \leq data \wedge lNode = nil$ **then**
        begin
          $x :=$ **new** $NodeST(e)$;
          $lNode, lNode.pNode := x, this$
        end
      else if $e > data \wedge rNode = nil$ **then**
        begin
          $x :=$ **new** $NodeST(e)$;
          $rNode, rNode.pNode := x, this$
        end
      else
        $state, d := adding, e$
      if $x \neq nil$ **then**
        begin
          while $x.pNode \neq nil$ **do**
            if $x.pNode$ **is** $NodeST$ **then**
              $x, x.subtotal := x.pNode, x.subtotal + e$
        end
    end
action $doAddElement$
  when $state = adding$ **do**
    begin
      $state := idle$;
      if $d \leq data$ **then**
        $lNode.add(d)$
      else
        $rNode.add(d)$

```
        end
    public method search(e : integer, res n : NodeST)
      when state = idle do
        begin
          if e = data then
            n := this
          else if e < data ∧ lNode ≠ nil then
            begin
              state := searching ;
              lNode.search(e, n)
            end
          else if e > data ∧ rNode ≠ nil then
            begin
              state := searching ;
              rNode.search(e, n)
            end
          else
            n := nil
          state := idle
        end
    public method next(res n : NodeST)
      var curNode : NodeST
      begin
        curNode := this ;
        if curNode.rNode ≠ nil then
          begin
            if curNode.rNode is NodeST then
              n := curNode.rNode ;
            while n.lNode ≠ nil do
              if n.lNode is NodeST then
                n := n.lNode
          end
        else
```

```
          begin
            n := curNode.pNode ;
            while n ≠ nil ∧ curNode = n.rNode do
              if n.pNode is NodeST then
                curNode, n := n, n.pNode
          end
      end
    public method calcTotal(res s : integer)
      s := this.subtotal
  end

  class TreeST inherit Tree
    invariant Inv_TreeST
    public method add(e : integer)
      if root = nil then
        root := new NodeST(e)
      else
        root.add(e)
    public method remove(e : integer)
      when root ≠ nil do
        var r, x, y : NodeST
        begin
          r := nil ;
          root.search(e, r) ;
          if r ≠ nil then
            begin
              r.state := deleting ;
              if r.lnode = nil ∨ r.rNode = nil then
                y := r
              else
                r.next(y)
              if y.lNode ≠ nil ∧ y.lNode is NodeST then
                x := y.lNode
```

```
else if y.rNode is NodeST then
  x := y.rNode
if x ≠ nil then
  x.pNode := y.pNode
if y.pNode = nil then
  root := x
else
  begin
    if y = y.pNode.lNode ∧ y.pNode.lNode is NodeST
    then
      y.pNode.lNode := x
    else if y.pNode.rNode is NodeST then
      y.pNode.rNode := x
    var oldVal, newVal : integer
    var temp : NodeST
    oldVal, newVal, temp := y.subtotal, 0, y ;
    if x ≠ nil newVal := x.subtotal ;
    while temp.pNode ≠ nil do
      if temp.pNode is NodeST then
        begin
          temp := temp.pNode ;
          temp.subtotal := temp.subtotal − oldVal +
                           newVal
        end
  end
if y ≠ r then
  begin
    var rOldData : integer
    var temp : NodeST
    rOldData, temp := r.data, r ;
    r.data := y.data ;
    r.subtotal := r.subtotal − rOldData + y.data ;
    while temp.pNode ≠ nil do
```

if $temp.pNode$ **is** $NodeST$ **then**
begin
$temp := temp.pNode$;
$temp.subtotal := temp.subtotal + y.data - rOldData$
end
end
$r.state := idle$
end
end
end

In order to specify the invariants, we rename the variables in classes *Tree*,*Node*,*TreeST* and *NodeST* using the same naming scheme as in the Food Court example in the previous section.
The invariant Inv_{Tree} for the class *Tree* is given as:

$$Inv_{Tree} : (this.getTotal() = \Sigma n \in nodeOf(this) \bullet n.data_0) \wedge this.getTotal() = sum$$

The Inv_{TreeST} for the class *TreeST* is given as:

$$Inv_{TreeST} : (\forall m \in nodeOf(this) \bullet m.data_0 = m.data_1) \wedge (root_1.subtotal_1 = \Sigma n \in nodeOf(this) \bullet n.data_1) \wedge (root_1.subtotal_1 = sum) \wedge (\forall p \in nodeOf(this) \bullet p.subtotal_1 = \Sigma c \in inSubtree(p) \bullet c.data_1)$$

where $nodeOf(t)$ returns the set of nodes in the tree t and $inSubtree(p)$ returns the set of nodes in the subtree rooted at the node p. Functions $nodeOf()$ and $inSubtree()$ are used only for specifying the invariants.

In this example, the class *Bag* is refined by the class *Tree*. The refinement invariant R_B for this refinement is given by $R_B \equiv Inv_{Bag} \wedge Inv_{Tree}$. The class *Tree* is refined by the class *TreeST*. The refinement invariant R_T for this refinement is given by $R_T \equiv Inv_{Tree} \wedge Inv_{TreeST}$.

Chapter 8

Inheritance Anomaly

In Chapter 2, we discussed the problem of *Inheritance Anomaly* and its manifestations in concurrent classes with guarded methods. In this chapter, we present the approach taken by *Lime* to avoid the problem of *Inheritance Anomaly* using its guarded methods and guarded actions.

There are two cases in which *Inheritance Anomaly* can occur in classes with guarded methods: when the child class introduces a history-sensitive method and when an acceptable state is modified in the child class. The synchronization code of the parent class needs to be inherited and augmented with additional conditions that reflect the modifications in state of the child class. The child class should be able to achieve this without non-trivial re-definition of its methods, so that *Inheritance Anomaly* is avoided. As a solution, we present how class inheritance in *Lime* avoids the problem of *Inheritance Anomaly*.

8.1 History-only Sensitive Methods

As discussed in chapter 2, *Inheritance Anomaly* can occur in the presence of history-sensitive methods. We consider the example of bounded buffer from [22]. The class *b_buf* represents a bounded buffer. The method *put*() adds an integer value to the buffer when it is not full; and the method *get*() retrieves an integer value from the buffer when it is not empty.

The *Lime* class implementing *b_buf* is defined as:

class *b_buf*
  var *buf* : **array of** *integer*
  var *in*, *out*, *n*, *size* : *integer*
  initialization $(m : integer)\ in, out, n, size := 0, 0, 0, m$
  method $put(x : integer)$
    when $n < size$ **do**
      $in, buf[in], n := (in + 1) mod\ size, x, n + 1$
  method *get*(**res** $x : integer$)
    when $n > 0$ **do**
      $out, x, n := (out + 1) mod\ size, b[out], n - 1$
end

The class *gb_buf* inherits from the bounded buffer class *b_buf*. The history-sensitive method *gget*() of class *gb_buf* cannot be invoked immediately after the method *put*(). The method *gget*() retrieves an integer value from the buffer when it is not empty. The class *gb_buf* uses a flag, *after_put*, to keep track of the invocations of the *put*() method. The *Lime* class implementing *gb_buf* is defined as:

class *gb_buf* **inherit** *b_buf*
  var *after_put* : *boolean*
  initialization $(m : integer)$
    begin *super*(m) ; *after_put* := *false* **end**
  method *gget*(**res** $x : integer$)
    when $\neg after\_put \wedge n > 0$ **do**
      begin *super.get*(x) ; *after_put* := *false* **end**
  method $put(x : integer)$
    when $n < size$ **do**
      begin *super.put*(x) ; *after_put* := *true* **end**
  method *get*(**res** $x : integer$)
    when $n > 0$ **do**
      begin *super.get*(x) ; *after_put* := *false* **end**
end

In the class *gb_buf*, the methods *put*() and *get*() are overridden in order to set and reset the newly added boolean flag *after_put*. Since *put*() and *get*() in *gb_buf* use super calls to invoke the *put*() and *get*() methods of *b_buf*, it is ensured that there is no breakage in encapsulation. The programmer of *gb_buf* class does not need to have access to the implementation of *put*() and *get*() methods of *b_buf*.

In the presence of history-sensitive method *gget*(), the methods *put*() and *get*() of *gb_buf* have been redefined but it is achieved in such a manner that there is no breach in encapsulation and the redefinition is trivial. Therefore, *Inheritance Anomaly* does not occur in *Lime* classes implementing *b_buf* and *gb_buf*.

8.2 Modification of Acceptable States

The second case where *Inheritance Anomaly* has been observed in classes with guarded methods is when the acceptable states of methods in the class are modified. We illustrate this with the example of bounded buffer along with the Lock mixin class from [22]. The class *lb_buf* inherits from *b_buf* and extends the class *Lock*. In the class *lb_buf*, the methods *put*() and *get*() can only be executed when the *locked* attribute is not *true*.

In Lime, we use the extends clause to implement the Lock mix-in class. When a class A extends a class B, then the methods of A can refer to the attributes of B. However, A does not inherit the methods of B and it cannot make super-calls to methods of B.

The *Lime* classes for *Lock* and *lb_buf* are defined as:

class *Lock*
  var *locked* : *boolean*
  initialization *locked* := *false*
  method *lock*
    when $\neg$*locked* **do** *locked* := *true*
  method *unlock*
    when *locked* **do** *locked* := *false*

end

class *lb_buf* **inherit** *b_buf* **extend** *Lock*
  method $put(x : integer)$
    when $\neg locked \wedge n < size$ **do** $super.put(x)$
  method $get($**res** $x : integer)$
    when $\neg locked \wedge n > 0$ **do** $super.get(x)$
end

We observe that the methods *put*() and *get*() from the superclass are redefined in the *lb_buf* class. But, instead of using the implementation of *put*() and *get*() from the superclass and redefining them, we choose to invoke the superclass method from within the corresponding child class method. Therefore, in this case, the redefinition of the methods is trivial and inheritance does not lead to a breakage in encapsulation.

8.3 Solution to Inheritance Anomaly with Guarded Actions

So far we have discussed a solution to *Inheritance Anomaly* using guarded methods in *Lime* classes. We consider the following example to illustrate a solution to *Inheritance Anomaly* using guarded actions of *Lime* classes:

We define a class $G1$ that stores the coordinates of two points. It also has a method *draw*() that enables an action *drawLine* to draw a line between two distinct points and a method *reset*() that resets the coordinates of the two points to (0,0).

class $G1$ {
  int x_1, y_1, x_2, y_2 ;
  void $G1()$ {
    $x_1 = 0$; $y_1 = 0$; $x_2 = 0$; $y_2 = 0$;
  }

```
void setPtAandB(int a_1, b_1, a_2, b_2) {
  if !( a_1 == 0 && b_1 == 0 && a_2 == 0 && b_2 == 0 )
    x_1 = a_1 ; y_1 = b_1 ; x_2 = a_2 ; y_2 = b_2 ;
}
void draw() {
  if !( x_1 == x_2 && y_1 == y_2 )
    // draw a line between PtA(x_1, y_1) and PtB(x_2, y_2)
}
void reset() {
  // resets the coordinates of the two points to (0,0)
  x_1 = 0 ; y_1 = 0 ; x_2 = 0 ; y_2 = 0 ;
}
}
```

Next, we define a class $G2$ that inherits from the class $G1$. In addition to the functionalities inherited from $G1$, the class $G2$ also introduces a method $drawQ1()$ which checks that the two distinct points are in the first quadrant of the X-Y plane, and draws a line between them. We assume that the two points for $drawQ1()$ cannot be on the X and Y axes. The additional restriction is that $drawQ1()$ cannot be called after $draw()$.

The class $G2$ is defined as:

```
class G2 : G1 {
bool afterDraw ;
  void G2() {
    x_1 = 0 ; y_1 = 0 ; x_2 = 0 ; y_2 = 0 ; afterDraw = false ;
  }
  void setPtAandB(int a_1, b_1, a_2, b_2) {
    if !( a_1 == 0 && b_1 == 0 && a_2 == 0 && b_2 == 0 )
      x_1 = a_1 ; y_1 = b_1 ; x_2 = a_2 ; y_2 = b_2 ; afterDraw = false ;
  }
  void draw() {
    if !( x_1 == x_2 && y_1 == y_2 ) {
```

$afterDraw = true$;
// draw a line between $PtA(x_1, y_1)$ and $PtB(x_2, y_2)$
}
}
void $reset()$ {
// resets the coordinates of the two points to (0,0)
$x_1 = 0$; $y_1 = 0$; $x_2 = 0$; $y_2 = 0$; $afterDraw = false$;
}
void $drawQ1()$ {
if $!afterDraw$ && $!(x_1 == x_2$ && $y_1 == y_2)$ &&
$(x_1 > 0$ && $y_1 > 0$ && $x_2 > 0$ && $y_2 > 0)$
// draw a line between $PtA(x_1, y_1)$ and $PtB(x_2, y_2)$
}
}

We observe that in class $G2$, the methods $setPtAandB()$, $draw()$ and $reset()$ of $G1$ need to be redefined in order to account for the changes in synchronization constraints due to the newly added method $drawQ1()$. The presence of the history-sensitive method $drawQ1()$ introduces *Inheritance Anomaly* in this example.

The *Lime* classes implementing $G1$ and $G2$ are defined as:

class $G1$
var $x_1, y_1, x_2, y_2 : integer$
var $dL : boolean$
initialization $x_1, y_1, x_2, y_2, dL := 0, 0, 0, 0, false$
method $setPtAandB(a_1, b_1, a_2, b_2 : integer)$
when $\neg(a_1 = 0 \wedge b_1 = 0 \wedge a_2 = 0 \wedge b_2 = 0)$ **do**
$x_1, y_1, x_2, y_2 := a_1, b_1, a_2, b_2$
method $draw$
when $\neg(x_1 = x_2 \wedge y_1 = y_2)$ **do**
$dL := true$
method $reset$

// resets the coordinates of the two points to (0,0)
$x_1, y_1, x_2, y_2 := 0, 0, 0, 0$
action *drawLine*
when *dL* **do**
begin
$dL :=$ *false* ;
// draw a line between $PtA(x_1, y_1)$ and $PtB(x_2, y_2)$
end
end

class $G2$ **inherit** $G1$
var *afterDraw* : *boolean*
initialization begin *super*() ; *afterDraw* := *false* **end**
method *setPtAandB*$(a_1, b_1, a_2, b_2$: *integer*)
when $\neg(\, a_1 = 0 \wedge b_1 = 0 \wedge a_2 = 0 \wedge b_2 = 0\,)$ **do**
begin *super.setPtAandB* ; *afterDraw* := *false* **end**
method *draw*
when $\neg(\, x_1 = x_2 \wedge y_1 = y_2\,)$ **do**
begin *super.draw* ; *afterDraw* := *true* **end**
method *reset*
// resets the coordinates of the two points to (0,0)
begin *super.reset* ; *afterDraw* := *false* **end**
method *drawQ*1
when $\neg$*afterDraw* $\wedge \neg(\, x_1 = x_2 \wedge y_1 = y_2\,) \wedge$
$(\, x_1 > 0 \wedge y_1 > 0 \wedge x_2 > 0 \wedge y_2 > 0\,)$
super.draw
end

As actions are inherited along with methods, class $G2$ does not need to redefine the action *drawLine*. Even though all the methods of $G2$ are redefined, the redefinition is achieved by superclass method calls and does not result in a breakage in encapsulation. Therefore *Inheritance Anomaly* does not occur in this example.

In the process of inheritance, if a child class action in *Lime* has to redefine the parent class action, it can be achieved through the $\oplus$ operator. We consider *Lime* classes $C1$ and $C2$ to illustrate this case. Class $C1$ defines the actions a and b. Class $C2$ inherits from class $C1$ and adds another action d. Action d is a history-sensitive action and cannot be executed immediately after action a. To achieve this condition, child class $C2$ introduces a boolean flag $afterA$. The classes $C1$ and $C2$ are defined as:

class $C1$
  var $p : P$
  initialization I
  action a
    when gA **do**
      A
  action b
    when gB **do**
      B
end

class $C2$ **inherit** $C1$
  var $q : Q$
  var $afterA : boolean$
  initialization begin $super()$; $afterA := false$; J **end**
  action $a \oplus super.a$
    when gA' **do**
      $afterA := true$
  action $b \oplus super.b$
    when gB' **do**
      $afterA := false$
  action d
    when $\neg afterA$ **do**
      D
end

Here, *action* $a \oplus super.a$ is interpreted as,

$$\textbf{when } gA' \wedge gA \textbf{ do begin } A \;;\; afterA := true \textbf{ end}$$

and *action* $b \oplus super.b$ is intepreted as,

$$\textbf{when } gB' \wedge gB \textbf{ do begin } B \;;\; afterA := false \textbf{ end}$$

Even though in class $C2$ the actions a and b are redefined, the redefinition is trivial and does not cause a breakage in encapsulation. Therefore, *Inheritance Anomaly* does not occur in this example.

In *Lime*, both guarded methods and guarded actions help avoid the problem of *Inheritance Anomaly.* This solution to *Inheritance Anomaly* is applicable only if the method calls are open calls, i.e., when the execution of a method or action encounters a method call, control is transferred to the object that would execute this method call, and the original method or action releases lock on the object so that another operation can be initiated on the object. In *Lime*, methods and actions are atomic upto method calls — when an action or method execution encounters a method call *nestedM*, it releases its exclusive control on the object and passes control to another object containing the method *nestedM*. Therefore, actions and methods in *Lime* classes are executed as open calls. However, for verification purposes, *Lime* considers the module actions and methods execute atomically upto completion, establishing closed calls.

The capability of *Lime* classes to invoke guarded methods and actions from their superclasses either using super calls or by using the $\oplus$ operator combined with the execution of the guarded methods and actions as open calls together provides a solution to the problem of *Inheritance Anomaly.*

Chapter 9

Conclusions

The design of class inheritance in *Lime* now includes inheritance of actions. During class inheritance, both actions and public methods can be inherited and overridden. The only exception are the *final* actions which can only be inherited. Class inheritance now helps preserve the reactive as well as autonomous behavior of the parent class. Inheritance of Lime classes fits the requirements for superposition refinement of action systems. Therefore, we have developed class refinement rules for class inheritance based on the notion of superposition refinement of action systems. Under these conditions, class inheritance is also a class refinement. We can now achieve stepwise refinement approach of program development by successive application of class inheritance. Each inheritance step can introduce some additional functionality in the child class while preserving the behavior of the parent class thus taking the program from a more abstract to a more concrete representation.

Classes in *Lime* are translated into modules. The module representations of the classes resemble action systems with procedures. We have added invariants and visibility specifiers to *Lime* modules. We have also broadened modules to support classes with inherited and possibly overridden actions and methods. The class refinement and verification rules can now be applied to the module representation of *Lime* classes in order to formally prove the refinement relationship between the classes.

Lime class structure allows invoking superclass methods and accessing superclass action via the newly added fusion ($\oplus$) operator. This class structure is useful in presenting the extended model of class inheritance in *Lime* as a means to avoid the problem of *Inheritance Anomaly.* This is possible mainly because *Lime* supports atomicity of methods and actions only up to method calls.

For verification and refinement we assume atomicity of methods and actions. An interesting direction for future research would be to model verification and refinement for the case when atomicity of methods and actions is limited only up to method calls. At present, module syntax of *Lime* does not support importing or exporting of variables for a specific purpose. Therefore, the module syntax cannot model *protected* variables. Modularization of *Lime* classes can be further improved by extending module syntax to provide a means for modeling *protected* variables. We leave implementation of inheritance of actions as future work.

Appendix A

Verification and Refinement of Inherited Lime Classes

In this chapter, we present a simple example of class inheritance in *Lime*. In the example, the class $C1$ inherits from and refines the class $C0$. This chapter also includes a complete formal proof of correctness for this refinement step based on the conditions for superposition refinement of classes from 6.7.

A.1 Sum of number series to n

In this example, we calculate the sum of first n positive integers. This is achieved first by adding all the n integers one by one. Then, in the child class which is also a refinement of the parent class, the sum of first n positive integers is calculated by using the formula $\frac{n(n+1)}{2}$ instead. This example is along the lines of the vector summation example from [28].

A.1.1 Class Definitions

In the class $C0$, the sum of the first n positive integers is calculated by the method *calcSum*. The method *calcSum* calculates the sum by adding the integers one at a time in repeated execution of the action *doSum*. The method *setN* sets the value of n. The method *getSum* returns the sum in the *result* parameter. The invariant for class $C0$ is

$$I_{C0} : (s \geq 0) \wedge (0 \leq m \leq n) \wedge (n = 0 \Rightarrow s = 0).$$

class $C0$
  var n, s, m : *integer*
  initialization
    $n, s, m := 0, 0, 0$
  public method $setN(k : integer)$
    begin
      assert $k \geq 0$;
      $n := k$
    end
  public method *calcSum*
    when $n > 0$ **do**
      $s, m := 0, n$
  public method $getSum($**res** $result : integer)$
    when $m = 0$ **do**
      $result := s$
  action *doSum*
    when $m > 0$ **do**
      $s, m := s + m, m - 1$
end

The class $C1$ inherits from the class $C0$. The public methods *setN*, *calcSum*, and *getSum* are inherited from $C0$. However, class $C1$ overrides the action *doSum* to use the formula $\frac{n(n+1)}{2}$ for calculating the sum of first n positive integers. In $C1$, the sum is calculated by a single execution of the action *doSum*. The invariant for class $C1$ is

$$J_{C1} : (s \geq 0) \wedge (0 \leq m \leq n) \wedge (n = C0.n) \wedge (n = 0 \Rightarrow s = 0) \wedge \\ (m < n \Rightarrow s = C0.s + \frac{C0.m(C0.m + 1)}{2})$$

class $C1$ **inherit** $C0$

action *doSum*
  when $m > 0$ **do**
    $s, m := \frac{m(m+1)}{2}, 0$
end

A.1.2 Module Definitions

The class definition of class $C0$ within a module amounts to the module declarations:

private var $C0$: **set of** *Object* $:= \{\}$
var n, s, m : $Object \rightarrow integer$
invariant $(\forall this \in C0 \bullet I_{C0})$
procedure $C0.new($**res** $this : Object)$
  $this :\notin C0 \cup \{nil\}$; $C0 := C0 \cup \{this\}$;
  $this.n, this.s, this.m := 0, 0, 0$
public procedure $C0.setN(this : Object, k : integer)$
  $\{this \in C0\}$;
  $\{k \geq 0\}$; $this.n := k$
public procedure $C0.calcSum(this : Object)$
  $\{this \in C0\}$;
  $[this.n > 0]$; $this.s, this.m := 0, this.n$
public procedure $C0.getSum(this : Object,$ **res** $result : integer)$
  $\{this \in C0\}$;
  $[this.m = 0]$; $result := this.s$
public action $C0.doSum$
  var $this :\in C0 \bullet$
    $[this.m > 0]$; $this.s, this.m := this.s + this.m, this.m - 1$

The class definition of class $C1$ within a module amounts to the module declarations:

private var $C1$: **set of** *Object* $:= \{\}$
private var $C0'$: **set of** *Object* $:= \{\}$
var n, s, m : $Object \rightarrow integer$

invariant $(\forall this \in C1 \bullet J_{C1})$
procedure $C1.new($**res** $this : Object)$
$C0.new(this)$; $C0' := C0' \cup \{this\}$;
$C1 := C1 \cup \{this\}$
public procedure $C1.setN(this : Object, k : integer)$
$\{this \in C1\}$; $C0'.setN(this, k)$
public procedure $C1.calcSum(this : Object)$
$\{this \in C1\}$; $C0'.calcSum(this)$
public procedure $C1.getSum(this : Object,$ **res** $result : integer)$
$\{this \in C1\}$; $C0'.getSum(this, result)$
public action $C1.doSum$
var $this :\in C1 \bullet$
$[this.m > 0]$; $this.s, this.m := \dfrac{this.m(this.m+1)}{2}, 0$

Since $C1$ inherits variables n, s and m from $C0$, we rename the variables as follows: $C0.n = n_0$, $C0.s = s_0$, $C0.m = m_0$, $C1.n = n_1$, $C1.s = s_1$, $C1.m = m_1$
Using these renamed variables, the invariants of classes $C0$ and $C1$ can be written as:

$$I : \forall p \in C0 \bullet (p.s_0 \geq 0) \wedge (0 \leq p.m_0 \leq p.n_0) \wedge (p.n_0 = 0 \Rightarrow p.s_0 = 0).$$

and

$$J : \forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2})$$

Combining these two class invariants, we have the invariant R given as:

$$\begin{aligned} R &\equiv I \wedge J \\ &\equiv C0' = C1 \wedge \forall p \in C1 \bullet (p.s_0 \geq 0) \wedge (p.s_1 \geq 0) \wedge (p.n_1 = p.n_0) \wedge \\ &\quad (0 \leq p.m_0 \leq p.n_0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_0 = 0 \Rightarrow p.s_0 = 0) \\ &\quad \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge \\ &\quad (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2}) \end{aligned}$$

A.1.3 Verification and Refinement

According to the rules for class inheritance in *Lime*, when class $C1$ inherits from class $C0$, the class $C1$ also refines the class $C0$ under the conditions of superposition refinement, written as $C0 \leq^I_J C1$. In order to establish this, the following class refinement conditions must hold:

(a) *Program Initialization:*

$$C0' = \{\} \wedge C1 = \{\} \Rightarrow I \wedge J$$

(b) *Object Creation:*

$$C0.new \leq^I_J C1.new$$

(c-1) *Main Method Refinement - setN :*

$$C0.setN \leq^I_J C1.setN$$

Main Method Enabledness - setN :

$$(I \wedge J \wedge en\ C0.setN \wedge tr\ C0.setN) \Rightarrow$$

$$(en\ C1.setN \vee en\ C1.doSum)$$

(c-2) *Main Method Refinement - calcSum :*

$$C0.calcSum \leq^I_J C1.calcSum$$

Main Method Enabledness - calcSum :

$$(I \wedge J \wedge en\ C0.calcSum \wedge tr\ C0.calcSum) \Rightarrow$$

$$(en\ C1.calcSum \vee en\ C1.doSum)$$

(c-3) *Main Method Refinement - getSum :*

$$C0.getSum \leq^I_J C1.getSum$$

Main Method Enabledness - getSum :

$$(I \wedge J \wedge en\ C0.getSum \wedge tr\ C0.getSum) \Rightarrow$$

$$(en\ C1.getSum \vee en\ C1.doSum)$$

(d) *New Method Refinement:* Class C1 does not define any new methods.

(e) *Main Action Refinement - doSum :*

$$C0.doSum \leq_J^I C1.doSum$$

Main Action Enabledness - doSum :

$$I \wedge J \wedge en\ C0.doSum \wedge tr\ C0.doSum \Rightarrow en\ C1.doSum$$

(f) *Auxiliary Action Refinement:* Class C1 does not define any auxiliary actions.

A.2 Detail Proof

Program Initialization:

$$C0' = \{\} \wedge C1 = \{\} \Rightarrow I \wedge J$$

Since at the point of program initialization, both $C0'$ and $C1$ are empty, so the above condition holds trivially over the empty sets.

Object Creation:

$$C0.new \leq_J^I C1.new$$

$$\equiv\ this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0 := 0,0,0 \leq_J^I this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0,0,0; C0' := C0' \cup \{this\}; C1 := C1 \cup \{this\}$$

$(Using\ 5.2,\ 5.3 \text{ and } R = I \wedge J)$

$\equiv$ $R \wedge tr(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0 := 0,0,0) \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0,0,0; C0' := C0' \cup \{this\}; C1 := C1 \cup \{this\}, \overline{wp}(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0 := 0,0,0, J))$

(Step-1)

$tr(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0 := 0,0,0)$

$(Using\ tr\ S \equiv wp(S, true))$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0 := 0,0,0, true)$

$(Using\ \ 4.10)$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, wp(this.n_0, this.s_0, this.m_0 := 0,0,0, true))$

$(Using\ \ 4.46,\ 4.5)$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, true[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]])$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, true)$

$(Using\ \ 4.10)$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}, wp(C0 := C0 \cup \{this\}, true))$

$(Using\ \ 4.5)$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}, true[C0 \backslash C0 \cup \{this\}])$

$\equiv$ $wp(this :\notin C0 \cup \{nil\}, true)$

$(Using\ \ 4.6)$

$\equiv \ \forall this \notin C0 \cup \{nil\} \bullet true$

$\equiv \ true$

Substituting this result in (Step-1), we have

$C0.new \leq^I_J C1.new$

$\equiv \ R \wedge true \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1,$
$this.s_1, this.m_1 := 0, 0, 0; C0' := C0' \cup \{this\}; C1 := C1 \cup \{this\},$
$\overline{wp}(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0$
$:= 0, 0, 0, J))$

$\equiv \ R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1,$
$this.m_1 := 0, 0, 0; C0' := C0' \cup \{this\}; C1 := C1 \cup \{this\}, \overline{wp}(this$
$:\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0 := 0, 0, 0,$
$J))$ ***(Step-2)***

$\overline{wp}(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_0, this.s_0, this.m_0$
$:= 0, 0, 0, J)$

$(Using\ 4.62)$
$\equiv \ \overline{wp}(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, \overline{wp}(this.n_0, this.s_0,$
$this.m_0 := 0, 0, 0, J))$

$(Using\ 4.46,\ 4.60)$
$\equiv \ \overline{wp}(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, J[n_0 \backslash n_0[this \leftarrow 0],$
$s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]])$

$(Using\ 4.62)$
$\equiv \ \overline{wp}(this :\notin C0 \cup \{nil\}, \overline{wp}(C0 := C0 \cup \{this\}, J[n_0 \backslash n_0[this \leftarrow 0],$
$s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]]))$

$(Using\ 4.60)$

$$\equiv \overline{wp}(this :\notin C0 \cup \{nil\}, J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}])$$

$(Using\ 4.61)$

$$\equiv \exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}]$$

Substituting this result in (Step-2), we have

$$C0.new \leq_J^I C1.new$$

$$\equiv R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0,0,0; C0' := C0' \cup \{this\}; C1 := C1 \cup \{this\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]] [C0 \backslash C0 \cup \{this\}]))$$

$(Using\ 4.10)$

$$\equiv R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0,0,0; C0' := C0' \cup \{this\}, wp(C1 := C1 \cup \{this\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}])))$$

$(Using\ 4.5)$

$$\equiv R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0,0,0; C0' := C0' \cup \{this\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]] [C0 \backslash C0 \cup \{this\}][C1 \backslash C1 \cup \{this\}]))$$

$(Using\ 4.10)$

$$\equiv R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0,0,0, wp(C0' := C0' \cup \{this\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]] [C0 \backslash C0 \cup \{this\}][C1 \backslash C1 \cup \{this\}])))$$

$(Using\ \ 4.5)$

$$\equiv \ R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}; this.n_1, this.s_1, this.m_1 := 0, 0, 0, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}] [C1 \backslash C1 \cup \{this\}][C0' \backslash C0' \cup \{this\}]))$$

$(Using\ \ 4.10)$

$$\equiv \ R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, wp(this.n_1, this.s_1, this.m_1 := 0, 0, 0, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}] [C1 \backslash C1 \cup \{this\}][C0' \backslash C0' \cup \{this\}])))$$

$(Using\ \ 4.46,\ 4.5)$

$$\equiv \ R \Rightarrow wp(this :\notin C0 \cup \{nil\}; C0 := C0 \cup \{this\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}][C1 \backslash C1 \cup \{this\}][C0' \backslash C0' \cup \{this\}][n_1 \backslash n_1[this \leftarrow 0], s_1 \backslash s_1[this \leftarrow 0], m_1 \backslash m_1[this \leftarrow 0]]))$$

$(Using\ \ 4.10)$

$$\equiv \ R \Rightarrow wp(this :\notin C0 \cup \{nil\}, wp(C0 := C0 \cup \{this\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]] [C0 \backslash C0 \cup \{this\}][C1 \backslash C1 \cup \{this\}][C0' \backslash C0' \cup \{this\}] [n_1 \backslash n_1[this \leftarrow 0], s_1 \backslash s_1[this \leftarrow 0], m_1 \backslash m_1[this \leftarrow 0]])))$$

$(Using\ \ 4.5)$

$$\equiv \ R \Rightarrow wp(this :\notin C0 \cup \{nil\}, (\exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}] [C1 \backslash C1 \cup \{this\}][C0' \backslash C0' \cup \{this\}][n_1 \backslash n_1[this \leftarrow 0], s_1 \backslash s_1[this \leftarrow 0], m_1 \backslash m_1[this \leftarrow 0]]))$$

$(Using\ \ 4.6)$

$$\equiv \ R \Rightarrow \forall this \notin C0 \cup \{nil\} \bullet \exists this \notin C0 \cup \{nil\} \bullet J[n_0 \backslash n_0[this \leftarrow 0], s_0 \backslash s_0[this \leftarrow 0], m_0 \backslash m_0[this \leftarrow 0]][C0 \backslash C0 \cup \{this\}]$$

$$[C1\backslash C1 \cup \{this\}][C0'\backslash C0' \cup \{this\}][n_1\backslash n_1[this \leftarrow 0], s_1\backslash s_1[this \leftarrow 0], m_1\backslash m_1[this \leftarrow 0]] \qquad \textbf{\textit{(Step-3)}}$$

$$J[n_0\backslash n_0[this \leftarrow 0], s_0\backslash s_0[this \leftarrow 0], m_0\backslash m_0[this \leftarrow 0]] [C0\backslash C0 \cup \{this\}][C1\backslash C1 \cup \{this\}][C0'\backslash C0' \cup \{this\}] [n_1\backslash n_1[this \leftarrow 0], s_1\backslash s_1[this \leftarrow 0], m_1\backslash m_1[this \leftarrow 0]]$$

$$\equiv \forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2}) [n_0\backslash n_0[this \leftarrow 0], s_0\backslash s_0[this \leftarrow 0], m_0\backslash m_0[this \leftarrow 0]][C0\backslash C0 \cup \{this\}] [C1\backslash C1 \cup \{this\}][C0'\backslash C0' \cup \{this\}][n_1\backslash n_1[this \leftarrow 0], s_1\backslash s_1[this \leftarrow 0], m_1\backslash m_1[this \leftarrow 0]]$$

$$\equiv \forall p \in C1 \cup \{this\} \bullet (p.s_1[this \leftarrow 0] \geq 0) \wedge (0 \leq p.m_1[this \leftarrow 0] \leq p.n_1[this \leftarrow 0]) \wedge (p.n_1[this \leftarrow 0] = p.n_0[this \leftarrow 0]) \wedge (p.n_1[this \leftarrow 0] = 0 \Rightarrow p.s_1[this \leftarrow 0] = 0) \wedge (p.m_1[this \leftarrow 0] < p.n_1[this \leftarrow 0] \Rightarrow p.s_1[this \leftarrow 0] = p.s_0[this \leftarrow 0] + \frac{p.m_0[this \leftarrow 0](p.m_0[this \leftarrow 0]+1)}{2})$$

Substituting this result in (Step-3) and performing a case analysis with $p = this$ and $p \neq this$, we have,

When $p = this$

$$C0.new \leq^I_J C1.new$$
$$\equiv R \Rightarrow \forall this \notin C0 \cup \{nil\} \bullet \exists this \notin C0 \cup \{nil\} \bullet (0 \geq 0) \wedge (0 \leq 0 \leq 0) \wedge (0 = 0) \wedge (0 = 0 \Rightarrow 0 = 0) \wedge (0 < 0 \Rightarrow 0 = 0 + \frac{0 * 1}{2})$$

$$\equiv R \Rightarrow \forall this \notin C0 \cup \{nil\} \bullet \exists this \notin C0 \cup \{nil\} \bullet true$$

$$\equiv true$$

When $p \neq this$

$$C0.new \leq^I_J C1.new$$

$\equiv$ $R \Rightarrow \forall this \notin C0 \cup \{nil\} \bullet \exists this \notin C0 \cup \{nil\} \bullet \forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \dfrac{p.m_0(p.m_0+1)}{2})$

$\equiv$ *true*

Therefore, $C0.new \leq_J^I C1.new$.

Main Method Refinement - setN :

$C0.setN \leq_J^I C1.setN$

$\equiv$ $\{this \in C0\}; \{k \geq 0\}; this.n_0 := k$
$\leq_J^I \{this \in C1\}; \{this \in C0'\}; \{k \geq 0\}; this.n_1 := k$

$(Using\ 5.2,\ 5.3 \text{ and } R = I \wedge J)$
$\equiv$ $R \wedge tr(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k) \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; \{k \geq 0\}; this.n_1 := k, \overline{wp}(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k, J))$ ***(Step-4)***

$tr(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k)$

$(Using\ 4.28)$
$\equiv$ $this \in C0 \wedge tr(\{k \geq 0\}; this.n_0 := k)$

$(Using\ 4.28)$
$\equiv$ $this \in C0 \wedge k \geq 0 \wedge tr(this.n_0 := k)$

$(Using\ tr\ S \equiv wp(S, true))$
$\equiv$ $this \in C0 \wedge k \geq 0 \wedge wp(this.n_0 := k, true)$

$(Using\ 4.46,\ 4.5)$
$\equiv$ $this \in C0 \wedge k \geq 0 \wedge true[n_0 \backslash n_0[this \leftarrow k]]$

$$\equiv \quad this \in C0 \wedge k \geq 0 \wedge true$$

$$\equiv \quad this \in C0 \wedge k \geq 0$$

(Result-1)

Substituting this result in (Step-4), we have

$$C0.setN \leq^I_J C1.setN$$

$$\equiv \quad R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; \{k \geq 0\};$$
$$this.n_1 := k, \overline{wp}(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k, J))$$

(Step-5)

$$\overline{wp}(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k, J)$$

$(Using\ 4.62)$
$$\equiv \quad \overline{wp}(\{this \in C0\}; \{k \geq 0\}, \overline{wp}(this.n_0 := k, J))$$

$(Using\ 4.46,\ 4.60)$
$$\equiv \quad \overline{wp}(\{this \in C0\}; \{k \geq 0\}, J[n_0 \backslash n_0[this \leftarrow k]])$$

$(Using\ 4.62)$
$$\equiv \quad \overline{wp}(\{this \in C0\}, \overline{wp}(\{k \geq 0\}, J[n_0 \backslash n_0[this \leftarrow k]]))$$

$(Using\ 4.58)$
$$\equiv \quad \overline{wp}(\{this \in C0\}, (k \geq 0 \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]]))$$

$(Using\ 4.58)$
$$\equiv \quad (this \in C0 \Rightarrow (k \geq 0 \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]]))$$

$(Using\ a \Rightarrow (b \Rightarrow c) \equiv (a \wedge b) \Rightarrow c)$
$$\equiv \quad (this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]]$$

Substituting this result in (Step-5), we have

$$C0.setN \leq_J^I C1.setN$$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\};\{this \in C0'\};\{k \geq 0\}; this.n_1 := k, ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]]))$$

$(Using\ 4.10)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\};\{this \in C0'\};\{k \geq 0\}, wp(this.n_1 := k, ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]])))$$

$(Using\ 4.46,\ 4.5)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\};\{this \in C0'\};\{k \geq 0\}, ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]][n_1 \backslash n_1[this \leftarrow k]]))$$

$(Using\ 4.10)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\};\{this \in C0'\}, wp(\{k \geq 0\}, ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]][n_1 \backslash n_1[this \leftarrow k]])))$$

$(Using\ 4.3)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\};\{this \in C0'\}, k \geq 0 \wedge ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]][n_1 \backslash n_1[this \leftarrow k]]))$$

$(Using\ 4.10)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\}, wp(\{this \in C0'\}, k \geq 0 \wedge ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]][n_1 \backslash n_1[this \leftarrow k]])))$$

$(Using\ 4.3)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow wp(\{this \in C1\}, this \in C0' \wedge k \geq 0 \wedge ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]][n_1 \backslash n_1[this \leftarrow k]]))$$

$(Using\ 4.3)$

$$\equiv \; R \wedge this \in C0 \wedge k \geq 0 \Rightarrow this \in C1 \wedge this \in C0' \wedge k \geq 0 \wedge ((this \in C0 \wedge k \geq 0) \Rightarrow J[n_0 \backslash n_0[this \leftarrow k]][n_1 \backslash n_1[this \leftarrow k]])$$

$$(\mathit{Since}\ R \wedge A1 \wedge A2 \Rightarrow B1 \wedge B2 \wedge A2 \wedge ((A1 \wedge A2) \Rightarrow J1) \equiv$$
$$R \wedge A1 \wedge A2 \Rightarrow B1 \wedge B2 \wedge J1)$$
$$\equiv \quad R \wedge \mathit{this} \in C0 \wedge k \geq 0 \Rightarrow \mathit{this} \in C1 \wedge \mathit{this} \in C0' \wedge J[n_0 \backslash n_0[\mathit{this} \leftarrow k]][n_1 \backslash n_1[\mathit{this} \leftarrow k]]$$

(Step-6)

$$J[n_0 \backslash n_0[\mathit{this} \leftarrow k]][n_1 \backslash n_1[\mathit{this} \leftarrow k]]$$

$$\equiv \quad \forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2}) [n_0 \backslash n_0[\mathit{this} \leftarrow k]][n_1 \backslash n_1[\mathit{this} \leftarrow k]]$$

$$\equiv \quad \forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1[\mathit{this} \leftarrow k]) \wedge (p.n_1[\mathit{this} \leftarrow k] = p.n_0[\mathit{this} \leftarrow k]) \wedge (p.n_1[\mathit{this} \leftarrow k] = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1[\mathit{this} \leftarrow k] \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2})$$

Substituting this result in (Step-6) and performing a case analysis with $p = \mathit{this}$ and $p \neq \mathit{this}$, we have,

When $p = \mathit{this}$

$$C0.\mathit{setN} \leq^I_J C1.\mathit{setN}$$

$$\equiv \quad R \wedge \mathit{this} \in C0 \wedge k \geq 0 \Rightarrow \mathit{this} \in C1 \wedge \mathit{this} \in C0' \wedge (\mathit{this}.s_1 \geq 0) \wedge (0 \leq \mathit{this}.m_1 \leq k) \wedge (k = k) \wedge (k = 0 \Rightarrow \mathit{this}.s_1 = 0) \wedge (\mathit{this}.m_1 < k \Rightarrow \mathit{this}.s_1 = \mathit{this}.s_0 + \frac{\mathit{this}.m_0(\mathit{this}.m_0+1)}{2})$$

$$\equiv \quad \mathit{true}$$

When $p \neq \mathit{this}$

$$C0.\mathit{setN} \leq^I_J C1.\mathit{setN}$$

$$\equiv \quad R \wedge \mathit{this} \in C0 \wedge k \geq 0 \Rightarrow \mathit{this} \in C1 \wedge \mathit{this} \in C0' \wedge \forall p \in C1 \bullet (p.s_1$$

$$\geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0)$$
$$\wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0 + 1)}{2})$$

$$\equiv \quad true$$

Therefore, $C0.setN \leq^I_J C1.setN$.

Main Method Enabledness - setN :

$$(I \wedge J \wedge en\ C0.setN \wedge tr\ C0.setN) \Rightarrow (en\ C1.setN \vee en\ C1.doSum)$$

(Step-7)

$$en\ C0.setN \equiv en(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k)$$

$$(Using\ \ 4.25)$$
$$\equiv \quad en(\{k \geq 0\}; this.n_0 := k)$$

$$(Using\ \ 4.25)$$
$$\equiv \quad en(this.n_0 := k)$$

$$(Using\ en\ S \equiv \neg wp(S, false))$$
$$\equiv \quad \neg wp(this.n_0 := k, false)$$

$$(Using\ \ 4.46,\ 4.5)$$
$$\equiv \quad \neg(false[n_0 \backslash n_0[this \leftarrow k]])$$

$$\equiv \quad \neg false$$

$$\equiv \quad true$$

(Result-2)

$$tr\ C0.setN \equiv tr(\{this \in C0\}; \{k \geq 0\}; this.n_0 := k)$$

$(From\ Result-1)$

$$\equiv\ this \in C0 \wedge k \geq 0$$

(Result-3)

$$en\ C1.setN \equiv en(\{this \in C1\};\ C0'.setN(this, k))$$

$$\equiv\ en(\{this \in C1\}; \{this \in C0'\}; \{k \geq 0\}; this.n_1 := k)$$

$(Using\ \ 4.25)$

$$\equiv\ en(\{this \in C0'\}; \{k \geq 0\}; this.n_1 := k)$$

$(Using\ \ 4.25)$

$$\equiv\ en(\{k \geq 0\}; this.n_1 := k)$$

$(Using\ \ 4.25)$

$$\equiv\ en(this.n_1 := k)$$

$(Using\ en\ S \equiv \neg wp(S, false))$

$$\equiv\ \neg wp(this.n_1 := k, false)$$

$(Using\ \ 4.46,\ \ 4.5)$

$$\equiv\ \neg(false[n_1 \backslash n_1[this \leftarrow k]])$$

$$\equiv\ \neg false$$

$$\equiv\ true$$

(Result-4)

$$en\ C1.doSum$$

$(Using\ \ 4.53)$

$$\equiv\ en(\sqcap this \in C1 \bullet [this.m_1 > 0]; this.s_1, this.m_1 :=$$

$\dfrac{this.m_1(this.m_1+1)}{2}, 0)$

$(Using\ \ 4.27)$

$\equiv\ \exists this \in C1 \bullet en([this.m_1 > 0]; this.s_1, this.m_1 :=$
$\dfrac{this.m_1(this.m_1+1)}{2}, 0)$

$(Using\ \ 4.26)$

$\equiv\ \exists this \in C1 \bullet this.m_1 > 0 \wedge en(this.s_1, this.m_1 :=$
$\dfrac{this.m_1(this.m_1+1)}{2}, 0)$

$(Using\ en\ S \equiv \neg wp(S, false))$

$\equiv\ \exists this \in C1 \bullet this.m_1 > 0 \wedge \neg wp(this.s_1, this.m_1 :=$
$\dfrac{this.m_1(this.m_1+1)}{2}, 0, false)$

$(Using\ \ 4.46,\ \ 4.5)$

$\equiv\ \exists this \in C1 \bullet this.m_1 > 0 \wedge$
$\neg(false[s_1 \backslash s_1[this \leftarrow \dfrac{this.m_1(this.m_1+1)}{2}]][m_1 \backslash m_1[this \leftarrow 0]])$

$\equiv\ \exists this \in C1 \bullet this.m_1 > 0 \wedge \neg false$

$\equiv\ \exists this \in C1 \bullet this.m_1 > 0$

(Result-5)

Substituting Results 2, 3, 4, and 5 in (Step-7), we have

$(I \wedge J \wedge true \wedge this \in C0 \wedge k \geq 0) \Rightarrow (true \vee \exists this \in C1 \bullet this.m_1$
$> 0)$

$\equiv\ (I \wedge J \wedge this \in C0 \wedge k \geq 0) \Rightarrow true$

$\equiv\ true$

Therefore, the enabledness condition for the method *setN* is satisfied.

Main Method Refinement - calcSum :

$$C0.calcSum \leq^I_J C1.calcSum$$

$$\equiv \{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0 \leq^I_J \{this \in C1\}; C0'.calcSum(this)$$

$$\equiv \{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0 \leq^I_J \{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0]; this.s_1, this.m_1 := 0, this.n_1$$

$(Using\ 5.2,\ 5.3 \text{ and } R = I \wedge J)$

$$\equiv R \wedge tr(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0) \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0]; this.s_1, this.m_1 := 0, this.n_1, \overline{wp}(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0, J)) \quad \textbf{(Step-8)}$$

$$tr(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0)$$

$(Using\ 4.28)$

$$\equiv this \in C0 \wedge tr([this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0)$$

$(Using\ 4.29)$

$$\equiv this \in C0 \wedge en(this.s_0, this.m_0 := 0, this.n_0)$$

$(Using\ en\ S \equiv \neg wp(S, false))$

$$\equiv this \in C0 \wedge \neg wp(this.s_0, this.m_0 := 0, this.n_0, false)$$

$(Using\ 4.46,\ 4.5)$

$$\equiv this \in C0 \wedge \neg(false[s_0 \backslash s_0[this \leftarrow 0]][m_0 \backslash m_0[this \leftarrow this.n_0]])$$

$$\equiv this \in C0 \wedge \neg false$$

$$\equiv \; this \in C0$$

(Result-6)

Substituting this result in (Step-8), we have

$$C0.calcSum \leq^{I}_{J} C1.calcSum$$

$$\equiv \; R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0]; this.s_1, this.m_1 := 0, this.n_1, \overline{wp}(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0, J))$$

(Step-9)

$$\overline{wp}(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0, J)$$

$(Using\ 4.62)$

$$\equiv \; \overline{wp}(\{this \in C0\}; [this.n_0 > 0], \overline{wp}(this.s_0, this.m_0 := 0, this.n_0, J))$$

$(Using\ 4.46,\ 4.60)$

$$\equiv \; \overline{wp}(\{this \in C0\}; [this.n_0 > 0], J[s_0 \backslash s_0[this \leftarrow 0]] [m_0 \backslash m_0[this \leftarrow this.n_0]])$$

$(Using\ 4.62)$

$$\equiv \; \overline{wp}(\{this \in C0\}, \overline{wp}([this.n_0 > 0], J[s_0 \backslash s_0[this \leftarrow 0]] [m_0 \backslash m_0[this \leftarrow this.n_0]]))$$

$(Using\ 4.59)$

$$\equiv \; \overline{wp}(\{this \in C0\}, this.n_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow 0]] [m_0 \backslash m_0[this \leftarrow this.n_0]])$$

$(Using\ 4.58)$

$$\equiv \; this \in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow 0]] [m_0 \backslash m_0[this \leftarrow this.n_0]]$$

Substituting this result in (Step-9), we have

$C0.calcSum \leq^{I}_{J} C1.calcSum$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0];$
$this.s_1, this.m_1 := 0, this.n_1, (this \in C0 \Rightarrow this.n_0 > 0 \wedge$
$J[s_0 \backslash s_0[this \leftarrow 0]][m_0 \backslash m_0[this \leftarrow this.n_0]]))$

$(Using\ \ 4.10)$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0],$
$wp(this.s_1, this.m_1 := 0, this.n_1, (this \in C0 \Rightarrow this.n_0 > 0 \wedge$
$J[s_0 \backslash s_0[this \leftarrow 0]][m_0 \backslash m_0[this \leftarrow this.n_0]])))$

$(Using\ \ 4.46,\ \ 4.5)$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0], (this$
$\in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow 0]][m_0 \backslash m_0[this \leftarrow this.n_0]]$
$[s_1 \backslash s_1[this \leftarrow 0]][m_1 \backslash m_1[this \leftarrow this.n_1]]))$

$(Using\ \ 4.10)$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}, wp([this.n_1 > 0],$
$(this \in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow 0]]$
$[m_0 \backslash m_0[this \leftarrow this.n_0]][s_1 \backslash s_1[this \leftarrow 0]][m_1 \backslash m_1[this \leftarrow this.n_1]])))$

$(Using\ \ 4.4)$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}, (this.n_1 > 0 \Rightarrow$
$(this \in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow 0]]$
$[m_0 \backslash m_0[this \leftarrow this.n_0]][s_1 \backslash s_1[this \leftarrow 0]][m_1 \backslash m_1[this \leftarrow this.n_1]])))$

$(Using\ \ 4.10)$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}, wp(\{this \in C0'\}, (this.n_1 > 0 \Rightarrow$
$(this \in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow 0]]$
$[m_0 \backslash m_0[this \leftarrow this.n_0]][s_1 \backslash s_1[this \leftarrow 0]][m_1 \backslash m_1[this \leftarrow this.n_1]]))))$

$(Using\ \ 4.3)$

$$\equiv \; R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}, this \in C0' \wedge (this.n_1 > 0 \Rightarrow (this \in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0\backslash s_0[this \leftarrow 0]] [m_0\backslash m_0[this \leftarrow this.n_0]][s_1\backslash s_1[this \leftarrow 0]][m_1\backslash m_1[this \leftarrow this.n_1]])))$$

(*Using* 4.3)

$$\equiv \; R \wedge this \in C0 \Rightarrow (this \in C1 \wedge this \in C0' \wedge (this.n_1 > 0 \Rightarrow (this \in C0 \Rightarrow this.n_0 > 0 \wedge J[s_0\backslash s_0[this \leftarrow 0]] [m_0\backslash m_0[this \leftarrow this.n_0]][s_1\backslash s_1[this \leftarrow 0]][m_1\backslash m_1[this \leftarrow this.n_1]])))$$

(*Since* $a \Rightarrow b \Rightarrow c \equiv (a \wedge b) \Rightarrow c$)

$$\equiv \; R \wedge this \in C0 \Rightarrow (this \in C1 \wedge this \in C0' \wedge (this.n_1 > 0 \wedge this \in C0) \Rightarrow this.n_0 > 0 \wedge J[s_0\backslash s_0[this \leftarrow 0]][m_0\backslash m_0[this \leftarrow this.n_0]] [s_1\backslash s_1[this \leftarrow 0]][m_1\backslash m_1[this \leftarrow this.n_1]])$$

(Step-10)

$$J[s_0\backslash s_0[this \leftarrow 0]][m_0\backslash m_0[this \leftarrow this.n_0]][s_1\backslash s_1[this \leftarrow 0]] [m_1\backslash m_1[this \leftarrow this.n_1]]$$

$$\equiv \; \forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2}) [s_0\backslash s_0[this \leftarrow 0]][m_0\backslash m_0[this \leftarrow this.n_0]][s_1\backslash s_1[this \leftarrow 0]] [m_1\backslash m_1[this \leftarrow this.n_1]]$$

$$\equiv \; \forall p \in C1 \bullet (p.s_1[this \leftarrow 0] \geq 0) \wedge (0 \leq p.m_1[this \leftarrow this.n_1] \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1[this \leftarrow 0] = 0) \wedge (p.m_1[this \leftarrow this.n_1] < p.n_1 \Rightarrow p.s_1[this \leftarrow 0] = p.s_0[this \leftarrow 0] + \frac{p.m_0[this \leftarrow this.n_0](p.m_0[this \leftarrow this.n_0]+1)}{2})$$

Substituting this result in (Step-10) and performing a case analysis with $p = this$ and $p \neq this$, we have,

When $p = this$

$$C0.calcSum \leq^I_J C1.calcSum$$

$$\equiv \quad R \wedge this \in C0 \Rightarrow (this \in C1 \wedge this \in C0' \wedge (this.n_1 > 0 \wedge this \in C0) \Rightarrow this.n_0 > 0 \wedge (0 \leq this.n_1) \wedge (this.n_1 = this.n_0) \wedge (this.n_1 = 0 \Rightarrow 0 = 0))$$

$$\equiv \quad true$$

When $p \neq this$

$$C0.calcSum \leq^I_J C1.calcSum$$

$$\equiv \quad R \wedge this \in C0 \Rightarrow (this \in C1 \wedge this \in C0' \wedge (this.n_1 > 0 \wedge this \in C0) \Rightarrow this.n_0 > 0 \wedge (\forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0 + 1)}{2}))$$

$$\equiv \quad true$$

Therefore, $C0.calcSum \leq^I_J C1.calcSum$.

Main Method Enabledness - calcSum :

$$(I \wedge J \wedge en\ C0.calcSum \wedge tr\ C0.calcSum) \Rightarrow (en\ C1.calcSum \vee en\ C1.doSum)$$

(Step-11)

$$en\ C0.calcSum \equiv en(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0)$$

$(Using\ 4.25)$

$$\equiv \quad en([this.n_0 > 0]; this.s_0, this.m_0 := 0, this.n_0)$$

$(Using\ 4.26)$

$$\equiv \quad this.n_0 > 0 \wedge en(this.s_0, this.m_0 := 0, this.n_0)$$

$(Using\ en\ S \equiv \neg wp(S, false))$

$\equiv \quad this.n_0 > 0 \wedge \neg wp(this.s_0, this.m_0 := 0, this.n_0, false)$

$(Using\ \ 4.46,\ 4.5)$

$\equiv \quad this.n_0 > 0 \wedge \neg(false[s_0 \backslash s_0[this \leftarrow 0]][m_0 \backslash m_0[this \leftarrow this.n_0]])$

$\equiv \quad this.n_0 > 0 \wedge \neg false$

$\equiv \quad this.n_0 > 0$

(Result-7)

$tr\ C0.calcSum \equiv tr(\{this \in C0\}; [this.n_0 > 0]; this.s_0, this.m_0 := 0,$
$this.n_0)$

$(From\ Result - 6)$

$\equiv \quad this \in C0$

(Result-8)

$en\ C1.calcSum \equiv en(\{this \in C1\}; C0'.calcSum(this))$

$\equiv \quad en(\{this \in C1\}; \{this \in C0'\}; [this.n_1 > 0]; this.s_1, this.m_1 := 0,$
$this.n_1)$

$(Using\ \ 4.25)$

$\equiv \quad en(\{this \in C0'\}; [this.n_1 > 0]; this.s_1, this.m_1 := 0, this.n_1)$

$(Using\ \ 4.25)$

$\equiv \quad en([this.n_1 > 0]; this.s_1, this.m_1 := 0, this.n_1)$

$(Using\ \ 4.26)$

$\equiv \quad this.n_1 > 0 \wedge en(this.s_1, this.m_1 := 0, this.n_1)$

$(Using\ en\ S \equiv \neg wp(S, false))$

$$\equiv \quad this.n_1 > 0 \wedge \neg wp(this.s_1, this.m_1 := 0, this.n_1, false)$$

$$(Using\ 4.46,\ 4.5)$$
$$\equiv \quad this.n_1 > 0 \wedge \neg(false[s_1 \backslash s_1[this \leftarrow 0]][m_1 \backslash m_1[this \leftarrow this.n_1]])$$

$$\equiv \quad this.n_1 > 0 \wedge \neg false$$

$$\equiv \quad this.n_1 > 0$$

(Result-9)

Substituting Results 5, 7, 8 and 9 in (Step-11), we have the method enabledness condition as

$$(I \wedge J \wedge this.n_0 > 0 \wedge this \in C0) \Rightarrow (this.n_1 > 0 \vee \exists this \in C1 \bullet this.m_1 > 0)$$
$$\equiv \quad true$$

Therefore, the enabledness condition for method *calcSum* is satisfied.

Main Method Refinement - getSum :

$$C0.getSum \leq_J^I C1.getSum$$

$$\equiv \quad \{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0 \leq_J^I \{this \in C1\}; C0'.getSum(this, result_1)$$

$$\equiv \quad \{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0 \leq_J^I \{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0]; result_1 := this.s_1$$

$$(Using\ 5.2,\ 5.3\ \text{and}\ R = I \wedge J)$$
$$\equiv \quad R \wedge tr(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0) \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0]; result_1 := this.s_1, \overline{wp}(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0, J))$$

(Step-12)

$$tr(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0)$$

$$\begin{aligned}
&(Using\ 4.28)\\
\equiv\ & this \in C0 \wedge tr([this.m_0 = 0]; result_0 := this.s_0)\\[1em]
&(Using\ 4.29)\\
\equiv\ & this \in C0 \wedge en(result_0 := this.s_0)\\[1em]
&(Using\ en\ S \equiv \neg wp(S, false))\\
\equiv\ & this \in C0 \wedge \neg wp(result_0 := this.s_0, false)\\[1em]
&(Using\ 4.5)\\
\equiv\ & this \in C0 \wedge \neg(false[result_0 \backslash this.s_0])\\[1em]
\equiv\ & this \in C0 \wedge \neg false\\[1em]
\equiv\ & this \in C0
\end{aligned}$$

(Result-10)

Substituting this result in (Step-12), we have

$$C0.getSum \leq^I_J C1.getSum$$

$$\begin{aligned}
\equiv\ & R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0];\\
& result_1 := this.s_1, \overline{wp}(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0,\\
& J)) \qquad \textbf{\textit{(Step-13)}}
\end{aligned}$$

$$\overline{wp}(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0, J)$$

$$\begin{aligned}
&(Using\ 4.62)\\
\equiv\ & \overline{wp}(\{this \in C0\}; [this.m_0 = 0], \overline{wp}(result_0 := this.s_0, J))
\end{aligned}$$

$$(Using\ 4.60)$$
$$\equiv\ \overline{wp}(\{this \in C0\}; [this.m_0 = 0], J[result_0 \backslash this.s_0])$$

$$(Using\ 4.62)$$
$$\equiv\ \overline{wp}(\{this \in C0\}, \overline{wp}([this.m_0 = 0], J[result_0 \backslash this.s_0]))$$

$$(Using\ 4.59)$$
$$\equiv\ \overline{wp}(\{this \in C0\}, this.m_0 = 0 \wedge J[result_0 \backslash this.s_0])$$

$$(Using\ 4.58)$$
$$\equiv\ this \in C0 \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0]$$

Substituting this result in (Step-13), we have

$$C0.getSum \leq_J^I C1.getSum$$

$$\equiv\ R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0]; result_1 := this.s_1, (this \in C0 \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0]))$$

$$(Using\ 4.10)$$
$$\equiv\ R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0], wp(result_1 := this.s_1, (this \in C0 \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0])))$$

$$(Using\ 4.5)$$
$$\equiv\ R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0], (this \in C0 \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1]))$$

$$(Using\ 4.10)$$
$$\equiv\ R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}, wp([this.m_1 = 0], (this \in C0 \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1])))$$

$$(Using\ 4.4)$$

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}, (this.m_1 = 0 \Rightarrow (this \in C0 \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1])))$

(*Since* $a \Rightarrow b \Rightarrow c \equiv (a \wedge b) \Rightarrow c$)

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}; \{this \in C0'\}, ((this.m_1 = 0 \wedge this \in C0) \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1]))$

(*Using* 4.10)

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}, wp(\{this \in C0'\}, ((this.m_1 = 0 \wedge this \in C0) \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1])))$

(*Using* 4.3)

$\equiv$ $R \wedge this \in C0 \Rightarrow wp(\{this \in C1\}, (this \in C0') \wedge ((this.m_1 = 0 \wedge this \in C0) \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1]))$

(*Using* 4.3)

$\equiv$ $R \wedge this \in C0 \Rightarrow this \in C1 \wedge this \in C0' \wedge ((this.m_1 = 0 \wedge this \in C0) \Rightarrow this.m_0 = 0 \wedge J[result_0 \backslash this.s_0][result_1 \backslash this.s_1])$

$\equiv$ $R \wedge this \in C0 \Rightarrow this \in C1 \wedge this \in C0' \wedge ((this.m_1 = 0 \wedge this \in C0) \Rightarrow this.m_0 = 0 \wedge J)$

$\equiv$ *true*

Therefore, $C0.getSum \leq^I_J C1.getSum$.

Main Method Enabledness - getSum :

$(I \wedge J \wedge en\ C0.getSum \wedge tr\ C0.getSum) \Rightarrow (en\ C1.getSum \vee en\ C1.doSum)$

(Step-14)

$en\ C0.getSum \equiv en(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0)$

$$(\textit{Using } 4.25)$$
$$\equiv\ en([this.m_0 = 0]; result_0 := this.s_0)$$

$$(\textit{Using } 4.26)$$
$$\equiv\ this.m_0 = 0 \wedge en(result_0 := this.s_0)$$

$$(\textit{Using } en\ S \equiv \neg wp(S, false))$$
$$\equiv\ this.m_0 = 0 \wedge \neg wp(result_0 := this.s_0, false)$$

$$(\textit{Using } 4.5)$$
$$\equiv\ this.m_0 = 0 \wedge \neg(false[result_0 \backslash this.s_0])$$

$$\equiv\ this.m_0 = 0 \wedge \neg false$$

$$\equiv\ this.m_0 = 0$$

(Result-11)

$$tr\ C0.getSum \equiv tr(\{this \in C0\}; [this.m_0 = 0]; result_0 := this.s_0)$$

$$(\textit{From Result} - 10)$$
$$\equiv\ this \in C0$$

(Result-12)

$$en\ C1.getSum \equiv en(\{this \in C1\}; C0'.getSum(this, result_1)$$

$$\equiv\ en(\{this \in C1\}; \{this \in C0'\}; [this.m_1 = 0]; result_1 := this.s_1)$$

$$(\textit{Using } 4.25)$$
$$\equiv\ en(\{this \in C0'\}; [this.m_1 = 0]; result_1 := this.s_1)$$

$$(\textit{Using } 4.25)$$
$$\equiv\ en([this.m_1 = 0]; result_1 := this.s_1)$$

$$\begin{aligned}
& (\mathit{Using}\ \ 4.26) \\
\equiv\ & \mathit{this}.m_1 = 0 \wedge \mathit{en}(\mathit{result}_1 := \mathit{this}.s_1) \\[1em]
& (\mathit{Using\ en}\ S \equiv \neg \mathit{wp}(S, \mathit{false})) \\
\equiv\ & \mathit{this}.m_1 = 0 \wedge \neg \mathit{wp}(\mathit{result}_1 := \mathit{this}.s_1, \mathit{false}) \\[1em]
& (\mathit{Using}\ \ 4.5) \\
\equiv\ & \mathit{this}.m_1 = 0 \wedge \neg(\mathit{false}[\mathit{result}_1 \backslash \mathit{this}.s_1]) \\[1em]
\equiv\ & \mathit{this}.m_1 = 0 \wedge \neg \mathit{false} \\[1em]
\equiv\ & \mathit{this}.m_1 = 0
\end{aligned}$$

(Result-13)

Substituting Results 5, 11, 12 and 13 in (Step-14), we have the method enabledness condition as

$$\begin{aligned}
& (I \wedge J \wedge \mathit{this}.m_0 = 0 \wedge \mathit{this} \in C0) \Rightarrow (\mathit{this}.m_1 = 0 \vee \exists \mathit{this} \in C1 \bullet \\
& \mathit{this}.m_1 > 0) \\[1em]
\equiv\ & \mathit{true}
\end{aligned}$$

Therefore, the enabledness condition for method *getSum* is satisfied.

Main Action Refinement - doSum :

$$C0.\mathit{doSum} \leq^I_J C1.\mathit{doSum}$$

$$\begin{aligned}
& (\mathit{Using}\ \ 4.53) \\
\equiv\ & \sqcap \mathit{this} \in C0 \bullet [\mathit{this}.m_0 > 0]; \mathit{this}.s_0, \mathit{this}.m_0 := \mathit{this}.s_0 + \mathit{this}.m_0, \\
& \mathit{this}.m_0 - 1 \leq^I_J \sqcap \mathit{this} \in C1 \bullet [\mathit{this}.m_1 > 0]; \mathit{this}.s_1, \mathit{this}.m_1 := \\
& \frac{\mathit{this}.m_1(\mathit{this}.m_1 + 1)}{2}, 0 \\[1em]
& (\mathit{Using}\ 5.2,\ \ 5.3\ \text{and}\ R = I \wedge J)
\end{aligned}$$

$$\equiv \quad R \wedge tr(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1) \Rightarrow wp(\sqcap this \in C1 \bullet [this.m_1 > 0]; this.s_1, this.m_1 := \frac{this.m_1(this.m_1 + 1)}{2}, 0, \overline{wp}(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, J))$$

(Step-15)

$tr(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1)$

$(Using\ tr\ S \equiv wp(S, true))$

$\equiv \quad wp(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, true)$

$(Using\ 4.8)$

$\equiv \quad \forall this \in C0 \bullet wp([this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, true)$

$(Since\ tr\ S \equiv wp(S, true))$

$\equiv \quad \forall this \in C0 \bullet tr([this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1)$

$(Using\ 4.29)$

$\equiv \quad \forall this \in C0 \bullet en(this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1)$

$(Using\ en\ S \equiv \neg wp(S, false))$

$\equiv \quad \forall this \in C0 \bullet \neg wp(this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, false)$

$(Using\ 4.46,\ 4.5)$

$\equiv \quad \forall this \in C0 \bullet \neg(false[s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]])$

$$\equiv \ \forall this \in C0 \bullet \neg false$$

$$\equiv \ true$$

(Result-14)

Substituting this result in (Step-15), we have

$$C0.doSum \leq^I_J C1.doSum$$

$$\equiv \ R \wedge true \Rightarrow wp(\sqcap this \in C1 \bullet [this.m_1 > 0]; this.s_1, this.m_1 := \frac{this.m_1(this.m_1+1)}{2}, 0, \overline{wp}(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, J))$$

(Step-16)

$$\overline{wp}(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, J)$$

$(Using\ 4.64)$

$$\equiv \ \exists this \in C0 \bullet \overline{wp}([this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, J)$$

$(Using\ 4.62)$

$$\equiv \ \exists this \in C0 \bullet \overline{wp}([this.m_0 > 0], \overline{wp}(this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, J))$$

$(Using\ 4.46,\ 4.60)$

$$\equiv \ \exists this \in C0 \bullet \overline{wp}([this.m_0 > 0], J[s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]] [m_0 \backslash m_0[this \leftarrow this.m_0 - 1]])$$

$(Using\ 4.59)$

$$\equiv \ \exists this \in C0 \bullet this.m_0 > 0 \wedge J[s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]] [m_0 \backslash m_0[this \leftarrow this.m_0 - 1]]$$

Substituting this result in (Step-16), we have

$C0.doSum \leq^I_J C1.doSum$

$$\equiv \; R \Rightarrow wp(\sqcap this \in C1 \bullet [this.m_1 > 0]; this.s_1, this.m_1 := \frac{this.m_1(this.m_1+1)}{2}, 0, (\exists this \in C0 \bullet this.m_0 > 0 \wedge J \; [s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]]))$$

$(Using\;\; 4.8)$

$$\equiv \; R \Rightarrow \forall this \in C1 \bullet wp([this.m_1 > 0]; this.s_1, this.m_1 := \frac{this.m_1(this.m_1+1)}{2}, 0, (\exists this \in C0 \bullet this.m_0 > 0 \wedge J \; [s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]]))$$

$(Using\;\; 4.10)$

$$\equiv \; R \Rightarrow \forall this \in C1 \bullet wp([this.m_1 > 0], wp(this.s_1, this.m_1 := \frac{this.m_1(this.m_1+1)}{2}, 0, (\exists this \in C0 \bullet this.m_0 > 0 \wedge J \; [s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]])))$$

$(Using\;\; 4.46,\;\; 4.5)$

$$\equiv \; R \Rightarrow \forall this \in C1 \bullet wp([this.m_1 > 0], (\exists this \in C0 \bullet this.m_0 > 0 \wedge J \; [s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]] \; [s_1 \backslash s_1[this \leftarrow \frac{this.m_1(this.m_1+1)}{2}]][m_1 \backslash m_1[this \leftarrow 0]]))$$

$(Using\;\; 4.4)$

$$\equiv \; R \Rightarrow \forall this \in C1 \bullet this.m_1 > 0 \Rightarrow (\exists this \in C0 \bullet this.m_0 > 0 \wedge J \; [s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]] \; [s_1 \backslash s_1[this \leftarrow \frac{this.m_1(this.m_1+1)}{2}]][m_1 \backslash m_1[this \leftarrow 0]])$$

$(Expanding\;\; J)$

$$\equiv \; R \Rightarrow \forall this \in C1 \bullet this.m_1 > 0 \Rightarrow (\exists this \in C0 \bullet this.m_0 > 0 \wedge (\forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0$$

$$\Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2}))$$
$$[s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]]$$
$$[s_1 \backslash s_1[this \leftarrow \frac{this.m_1(this.m_1+1)}{2}]][m_1 \backslash m_1[this \leftarrow 0]])$$

$$\equiv R \Rightarrow \forall this \in C1 \bullet this.m_1 > 0 \Rightarrow (\exists this \in C0 \bullet this.m_0 > 0 \wedge (\forall p \in C1 \bullet (p.s_1[this \leftarrow \frac{this.m_1(this.m_1+1)}{2}] \geq 0) \wedge (0 \leq p.m_1[this \leftarrow 0] \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1[this \leftarrow \frac{this.m_1(this.m_1+1)}{2}] = 0) \wedge (p.m_1[this \leftarrow 0] < p.n_1 \Rightarrow p.s_1[this \leftarrow \frac{this.m_1(this.m_1+1)}{2}] = p.s_0[this \leftarrow this.s_0 + this.m_0] + \frac{p.m_0[this \leftarrow this.m_0 - 1](p.m_0[this \leftarrow this.m_0 - 1]+1)}{2})))$$

(Step-17)

Performing a case analysis at (Step-17) with $p = this$ and $p \neq this$, we have,

When $p = this$

$$C0.doSum \leq^I_J C1.doSum$$

$$\equiv R \Rightarrow \forall this \in C1 \bullet this.m_1 > 0 \Rightarrow (\exists this \in C0 \bullet this.m_0 > 0 \wedge ((\frac{this.m_1(this.m_1+1)}{2} \geq 0) \wedge (0 \leq 0 \leq this.n_1) \wedge (this.n_1 = this.n_0) \wedge (this.n_1 = 0 \Rightarrow \frac{this.m_1(this.m_1+1)}{2} = 0) \wedge (0 < this.n_1 \Rightarrow \frac{this.m_1(this.m_1+1)}{2} = this.s_0 + this.m_0 + \frac{(this.m_0 - 1)this.m_0}{2})))$$

$$\equiv true$$

When $p \neq this$

$$C0.doSum \leq^I_J C1.doSum$$

$$\equiv R \Rightarrow \forall this \in C1 \bullet this.m_1 > 0 \Rightarrow (\exists this \in C0 \bullet this.m_0 > 0 \wedge (\forall p \in C1 \bullet (p.s_1 \geq 0) \wedge (0 \leq p.m_1 \leq p.n_1) \wedge (p.n_1 = p.n_0) \wedge (p.n_1 = 0 \Rightarrow p.s_1 = 0) \wedge (p.m_1 < p.n_1 \Rightarrow p.s_1 = p.s_0 + \frac{p.m_0(p.m_0+1)}{2})))$$

$\equiv$ *true*

Therefore, $C0.doSum \leq^I_J C1.doSum$.

Main Action Enabledness - doSum :

$I \wedge J \wedge en\ C0.doSum \wedge tr\ C0.doSum \Rightarrow en\ C1.doSum$

(Step-18)

$en\ C0.doSum \equiv en(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1)$

$(Using\ \ 4.27)$

$\equiv\ \exists this \in C0 \bullet en([this.m_0 > 0]; this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1)$

$(Using\ \ 4.26)$

$\equiv\ \exists this \in C0 \bullet this.m_0 > 0 \wedge en(this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1)$

$(Using\ en\ S \equiv \neg wp(S, false))$

$\equiv\ \exists this \in C0 \bullet this.m_0 > 0 \wedge \neg wp(this.s_0, this.m_0 := this.s_0 + this.m_0, this.m_0 - 1, false)$

$(Using\ \ 4.46,\ 4.5)$

$\equiv\ \exists this \in C0 \bullet this.m_0 > 0 \wedge \neg(false[s_0 \backslash s_0[this \leftarrow this.s_0 + this.m_0]][m_0 \backslash m_0[this \leftarrow this.m_0 - 1]])$

$\equiv\ \exists this \in C0 \bullet this.m_0 > 0 \wedge \neg false$

$\equiv\ \exists this \in C0 \bullet this.m_0 > 0$

(Result-15)

$tr\ C0.doSum \equiv tr(\sqcap this \in C0 \bullet [this.m_0 > 0]; this.s_0, this.m_0 :=$

$$this.s_0 + this.m_0, this.m_0 - 1)$$

$$(Using\ (Result - 14))$$
$$\equiv\ true$$

(Result-16)

Substituting Results 15, 16 and 5 in (Step-18), we have the action enabledness condition as

$$I \wedge J \wedge \exists this \in C0 \bullet this.m_0 > 0 \wedge true \Rightarrow \exists this \in C1 \bullet this.m_1 > 0$$

$$\equiv\ I \wedge J \wedge \exists this \in C0 \bullet this.m_0 > 0 \Rightarrow \exists this \in C1 \bullet this.m_1 > 0$$

$$\equiv\ true$$

Therefore, the enabledness condition for action *doSum* is satisfied.

A.3 Discussion

When class C1 inherits from class C0, their invariants are preserved by initialization. Object creation in C0 is refined by object creation in C1. The methods setN, calcSum and getSum of C0 are refined by the corresponding methods in C1. The method enabledness condition for each of these methods is also satisfied. The action doSum of C0 is refined by the overridden action doSum of C1. The enabledness condition of the action doSum is also satisfied. Therefore,

$$C0 \leq^I_J C1$$

Bibliography

[1] R.-J. R. Back, "Refinement Calculus, Lattices and Higher Order Logic," in *Program Design Calculi* (M. Broy, ed.), vol. 118 of *Springer NATO ASI Series, Series F : Computer and System Sciences*, (New York, NY, USA), pp. 53–71, Springer-Verlag, 1993.

[2] R.-J. R. Back, "Refinement of Parallel and Reactive Programs," in *Program Design Calculi* (M. Broy, ed.), vol. 118 of *Springer NATO ASI Series, Series F : Computer and System Sciences*, (New York, NY, USA), pp. 73–92, Springer-Verlag, 1993.

[3] R.-J. R. Back, M. Büchi, and E. Sekerinski, "Action-Based Concurrency and Synchronization for Objects," in *Proceedings of the 4th AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software*, Lecture Notes in Computer Science, pp. 248–262, Springer-Verlag, May 1997.

[4] R.-J. R. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," in *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pp. 131–142, ACM, 1983.

[5] R.-J. R. Back and R. Kurki-Suonio, "Distributed co-operation with action systems," *ACM Transactions on Programming Language and Systems*, vol. 10, no. 4, pp. 513–554, 1988.

[6] R.-J. R. Back, A. Mikhajlova, and J. V. Wright, "Class Refinement as Semantics of Correct Object Substitutability," *Formal Aspects of Computing*, vol. 12, pp. 18–40, 2000.

[7] R.-J. R. Back and K. Sere, "Superposition Refinement of Reactive Systems," *Formal Aspects of Computing*, vol. 8, no. 3, pp. 324–346, 1993.

[8] R.-J. R. Back and K. Sere, "Action Systems with Synchronous Communication," in *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi,* (Amsterdam, The Netherlands, The Netherlands), pp. 107–126, North-Holland Publishing Co., 1994.

[9] M. M. Bonsangue, J. N. Kok, and K. Sere, "An Approach to Object-Orientation in Action Systems"," in *Mathematics of Program Construction* (J. Jeuring, ed.), Lecture Notes in Computer Science 1422, (Marstrand, Sweden, June 1998), pp. 68–95, Springer-Verlag, 1998.

[10] M. M. Bonsangue, J. N. Kok, and K. Sere, "Developing Object-based Distributed Systems," in *3rd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)* (P. Ciancarini, A. Fantechi, and R. Gorrieri, eds.), pp. 19–34, Kluwer, 1999.

[11] M. Büchi and E. Sekerinski, "A Foundation for Refining Concurrent Objects," *Fundamenta Informaticae*, vol. 44, no. 1, pp. 25–61, 2000.

[12] T. A. Budd, *An Introduction to Object-Oriented Programming.* New York, NY: Addison Wesley, 2002.

[13] W. Chen and J. T. Udding, "Towards a Calculus of Data Refinement," in *Mathematics of Program Construction, 375th Anniversary of the Groningen University* (J. L. A. v. d. Snepscheut, ed.), Lecture Notes in Computer Science 375, (Groningen, The Netherlands), pp. 197–218, Springer-Verlag, 1989.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* The MIT Press and McGraw Hill, 2001.

[15] S. Ferenczi, "Guarded methods vs. inheritance anomaly: inheritance anomaly solved by nested guarded method calls," *SIGPLAN Not.*, vol. 30, no. 2, pp. 49–58, 1995.

[16] C. Fournet, C. Laneve, L. Maranget, and D. Rémy, "Inheritance in the join calculus," *J. Log. Algebr. Program.*, vol. 57, no. 1-2, pp. 23–69, 2003.

[17] D. Gries, *The Science of Programming.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1987.

[18] J. Liang, *Inheritance, Specification and documentation support for an Object-Oriented language.* Master's thesis, McMaster University, 2004.

[19] B. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, November 1994.

[20] K.-P. Löhr and M. Haustein, "The Jac System: Minimizing the Differences between Concurrent and Sequential Java Code," *Journal of Object Technology*, vol. 5, no. 7, pp. 43–56, 2006.

[21] C. V. Lopes and K. J. Lieberherr, "Abstracting process-to-function relations in concurrent object-oriented applications," in *Object-Oriented Programming*, vol. 821 of *Lecture Notes in Computer Science*, pp. 81–99, Springer Berlin / Heidelberg, 1994.

[22] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages," *Research directions in concurrent object-oriented programming*, pp. 107–150, 1993.

[23] G. Milicia and V. Sassone, "The inheritance anomaly: ten years after," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1267–1274, ACM, 2004.

[24] G. Milicia and V. Sassone, "Jeeg: temporal constraints for the synchronization of concurrent objects: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 5-6, pp. 539–572, 2005.

[25] J. Misra, "A Simple, Object-Based View of Multiprogramming," *Form. Methods Syst. Des.*, vol. 20, no. 1, pp. 23–45, 2002.

[26] C. Morgan, *Programming from Specifications.* Prentice Hall International Series in Computer Science, Prentice Hall, 1994.

[27] E. Sekerinski, "Concurrent Object-Oriented Programs: From Specification to Code," in *Formal Methods for Components and Objects, First International Symposium, FMCO 02* (F. S. d. Boer, M. Bonsangue, S. Graf, and W.-P. d. Roever, eds.), Lecture Notes in Computer Science 2852, (Leiden, The Netherlands), pp. 403–423, Springer-Verlag, 2003.

[28] E. Sekerinski, "Verification and refinement with fine-grained action-based concurrent objects," *Theoretical Computer Science*, vol. 331, no. 2-3, pp. 429–455, February 2005.

[29] K. Sere and M. Walden, "Data Refinement of Remote Procedures," tech. rep., 1997.

[30] A. Shahen, "An Aspect-Oriented Approach for Solving the Inheritance Anomaly Problem," in *Proceedings of the World Congress on Engineering and Computer Science 2007, WCECS '07, October 24 - 26, 2007, San Francisco, USA* (S. I. Ao, C. Douglas, W. S. Grundfest, L. Schruben, and X. Wu, eds.), Lecture Notes in Engineering and Computer Science, pp. 285–293, International Association of Engineers, Newswood Limited, 2007.

[31] N. Wirth, "Program development by stepwise refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, April 1971.