



Language Specification

Notice

This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2002-2005 Microsoft Corporation. All rights reserved.

Microsoft, Windows, and Visual C# are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Table of Contents

1. Introduction	1
1.1 Overview of features.....	1
1.1.1 Processes	2
1.1.2 Creating processes.....	2
1.1.2.1 activate methods.....	2
1.1.2.2 async method calls.....	2
1.1.3 Inter-process communication	3
1.1.4 Synchronization.....	3
1.1.5 End-states	3
1.1.6 Blocking and atomicity	3
1.1.7 Non-determinism.....	4
1.1.8 Symbolic execution	4
1.2 Examples.....	4
1.2.1 Dining Philosophers	4
1.2.2 Alternating-bit protocol.....	7
1.2.3 Symbolic execution	12
2. Lexical Structure	14
2.1 Models	14
2.2 Tokens.....	14
2.3 Keywords	14
2.4 Operators and punctuators	14
2.5 Pre-processing.....	14
3. Types.....	15
3.1 Simple Types	15
3.1.1 Predefined types	15
3.1.2 Enumerations.....	15
3.1.3 Symbolic types	15
3.1.4 Ranges	15
3.1.5 Structures.....	16
3.2 Complex Types	16
3.2.1 Arrays	16
3.2.2 Sets	16
3.2.3 Channels.....	16
3.2.4 Classes.....	16
3.2.5 object type	16
3.3 Instantiation and initialization.....	17
4. Enumerations.....	18
4.1 Enum declarations.....	18
4.2 Enum members	18
4.3 Enum values and operations	19
4.4 Non-deterministic selection	19
4.5 Symbolic enums.....	19
Ranges.....	20
4.6 Range declarations	20
4.7 Range variables and operations.....	20

- 4.8 Non-deterministic selection 20
- 5. Arrays 21**
 - 5.1 Array declarations 21
 - 5.2 Array variables and operations 21
 - 5.2.1 Construction 21
 - 5.2.2 Indexing..... 21
 - 5.2.3 Iteration 22
 - 5.2.4 Sizeof..... 22
 - 5.2.5 Non-deterministic selection..... 22
- 6. Sets 23**
 - 6.1 Set declarations 23
 - 6.2 Set variables and operations..... 23
 - 6.2.1 Construction 23
 - 6.2.2 Adding an element 23
 - 6.2.3 Removing an element..... 23
 - 6.2.4 Set addition..... 23
 - 6.2.5 Set subtraction 24
 - 6.2.6 Testing membership 24
 - 6.2.7 Testing cardinality..... 24
 - 6.2.8 Iteration 24
 - 6.2.9 Non-deterministic selection..... 24
- 7. Channels 25**
 - 7.1 Channel declarations..... 25
 - 7.2 Channel variables and operations 25
 - 7.2.1 Construction 25
 - 7.2.2 Sending a message 25
 - 7.2.3 Receiving a message 25
 - 7.2.4 Obtaining the queue length..... 25
- 8. Classes..... 26**
 - 8.1 Class declarations..... 26
 - 8.2 Class members 26
 - 8.2.1 Static and instance members 26
 - 8.3 Fields..... 27
 - 8.3.1 Field initialization 27
 - 8.3.2 Variable initializers 27
 - 8.4 Methods..... 27
 - 8.4.1 Method parameters 28
 - 8.4.1.1 Value parameters 29
 - 8.4.1.2 Output parameters..... 29
 - 8.4.2 Static and instance methods 29
 - 8.4.3 Atomic methods 29
 - 8.4.4 Activated methods..... 29
 - 8.4.5 Method body 30
- 9. Structs..... 31**
 - 9.1 Struct declarations..... 31
 - 9.2 Fields..... 31
- 10. Statements 32**

10.1 Blocks	32
10.1.1 Statement Lists	33
10.2 Labeled statements	33
10.3 Attributed statements	33
10.3.1 Attribute types	34
10.3.2 Context attributes	34
10.3.3 Trace attributes	35
10.4 The empty statement	35
10.5 Declaration statements	35
10.6 Expression statements	36
10.7 The if statement	36
10.8 Iteration statements	37
10.8.1 The while statement	37
10.8.2 The foreach statement	38
10.9 Jump statements	38
10.9.1 The goto statement	38
10.9.2 The return statement	38
10.9.3 The raise statement	39
10.10 The try statement	39
10.11 The async statement	40
10.12 Communication and blocking	40
10.12.1 The send statement	40
10.12.2 The select statement	40
10.12.2.1 The <code>first</code> qualifier	41
10.12.2.2 The <code>end</code> qualifier	41
10.12.2.3 The <code>visible</code> qualifier	41
10.12.2.4 Join lists	41
10.12.2.5 <code>wait</code> patterns	42
10.12.2.6 <code>receive</code> patterns	42
10.12.2.7 <code>event</code> patterns	42
10.12.2.8 Timeouts	42
10.13 Monitoring execution	43
10.13.1 The trace statement	43
10.13.2 The event statement	43
10.14 State-space control	44
10.14.1 The assert statement	44
10.14.2 The assume statement	44
10.14.3 Atomic blocks	45
11. Expressions	46
11.1 Expression classifications	46
11.1.1 Values of expressions	46
11.2 Operators	46
11.2.1 Operator precedence and associativity	47
11.2.2 Numeric promotions	47
11.2.2.1 Unary numeric promotions	47
11.2.2.2 Binary numeric promotions	48
11.3 Member lookup	48
11.4 Function members (methods)	48
11.4.1 Argument lists	48
11.4.2 Method invocation	49

ZING LANGUAGE SPECIFICATION

11.5 Primary expressions	49
11.5.1 Literals.....	50
11.5.2 Simple names	50
11.5.3 Parenthesized expressions	50
11.5.4 Member access	50
11.5.4.1 Identical simple names and type names.....	51
11.5.5 Element access	51
11.5.5.1 Array access.....	51
11.5.6 This access.....	52
11.5.7 The new operator.....	52
11.5.8 The sizeof operator.....	52
11.5.9 The choose operator	52
11.6 Unary operators.....	53
11.6.1 Unary plus operator	53
11.6.2 Unary minus operator.....	53
11.6.3 Logical negation operator.....	53
11.6.4 Bitwise complement operator.....	53
Arithmetic operators	53
11.6.5 Multiplication operator.....	54
11.6.6 Division operator.....	54
11.6.7 Remainder operator	54
11.6.8 Addition operator	54
11.6.9 Subtraction operator	54
11.7 Shift operators	55
11.8 Relational and membership-testing operators.....	55
11.8.1 Integer comparison operators.....	56
11.8.2 Boolean equality operators.....	56
11.8.3 Enumeration comparison operators.....	56
11.8.4 Reference type equality operators	56
11.8.5 The in operator	56
11.9 Logical operators	57
11.10 Conditional logical operators	57
11.11 Invocation expressions.....	57
11.12 Assignment	58
11.13 Expression.....	58
11.14 Constant expressions.....	58
11.15 Boolean expressions.....	59
A. Grammar	60
A.1 Lexical grammar	60
A.1.1 Line terminators	60
A.1.2 White space	60
A.1.3 Comments.....	60
A.1.4 Tokens	61
A.1.5 Unicode character escape sequences	61
A.1.6 Identifiers.....	61
A.1.7 Keywords.....	62
A.1.8 Literals.....	63
A.1.9 Operators and punctuators.....	64
A.1.10 Pre-processing directives.....	64
A.2 Syntactic grammar	66

A.2.1 Basic concepts	66
A.2.2 Types	67
A.2.3 Variables.....	68
A.2.4 Expressions.....	68
A.2.5 Statements	70
A.2.6 Compilation Unit.....	73
A.2.7 Classes	73
A.2.8 Structs.....	75
A.2.9 Arrays	75
A.2.10 Enums.....	75
A.2.11 Ranges	75
A.2.12 Sets	75
A.2.13 Channels	75
B. Runtime Errors	76
C. Example Source Code.....	77
C.1 Dining Philosophers	77
C.2 Alternating-bit protocol.....	79

1. Introduction

Concurrent programs are hard to develop and test. While writing concurrent programs, the programmer has to consider every possible interleaving of events among various processes. In spite of several decades of research and engineering experience, few people write robust concurrent programs. Concurrency related bugs (sometimes called “heisenbugs”) still surface only in stress-tests, and these bugs are very hard to reproduce, debug and fix. With the advent of efforts such as the .Net platform, we are enabling more programmers to write distributed and concurrent programs. Thus, problems associated with concurrency are only going to be more widespread.

A technique called “model checking” has proven to be surprisingly effective in the design and testing of concurrent programs. Model checkers work by systematically exploring all possible states of the concurrent program. Industrial software has such large number of states that it is infeasible for any systematic approach to cover all the reachable states. Our goal is the following: *suppose we manage to represent a “model” from a program, where a model abstractly represents only a small amount of information about the program, then it is feasible to systematically explore the states of the model.*

The Zing project has three components: (1) a modeling language for expressing executable concurrent models of software, (2) a model checking infrastructure for exploring the state space of Zing models, and (3) support infrastructure for generating Zing models automatically from common programming languages like VB, C/C++, C#, and MSIL. This document is a language specification for the Zing language. Details of the model checking infrastructure and support infrastructure for generating Zing models are beyond the scope of this document.

Zing is *not* a programming language. One does not do useful work directly in Zing. In a typical scenario, a Zing model is automatically extracted from some source system. Generally, the Zing model represents an abstraction of the original system’s behavior that is geared toward the detection of a particular error, or class of errors. Thus, Zing is a *target for automatic model extraction* from code. Like traditional modeling languages, Zing includes constructs for concurrency, communication (via shared memory or queues), and non-determinism. But unlike earlier efforts, Zing embraces modern software constructs such as functions, objects, exceptions, and dynamic memory allocation. Zing’s goal is to preserve as much of the control-structure of the source program as possible. Zing’s model checker can exploit the structure of the code, which is preserved in the model, to optimize systematic state space exploration.

1.1 Overview of features

Zing provides several features to support automatic generation of models from programs written in common programming languages.

- **Concurrency model:** Zing supports a basic asynchronous interleaving model of concurrency with both shared memory and message queues.
- **Control constructs:** In addition to sequential flow, branching and iteration, Zing supports function calls and exception handling. Functions can be called asynchronously. An asynchronous call returns to the caller immediately, and the callee runs as a fresh process in parallel with the caller.
- **Data:** Zing supports primitive and reference types. An object model similar to C# is supported, although inheritance is not supported.

Zing also provides features to support abstraction and efficient model checking.

- **Atomicity brackets:** Any sequence of Zing statements (with some restrictions) can be bracketed as atomic. This is essentially a directive to the model checker to not consider interleavings with other

ZING LANGUAGE SPECIFICATION

threads while any given thread executes an atomic sequence. It is the responsibility of the Zing model extractor (the tool that generates Zing from source code) to ensure that the source code satisfies such atomicity specifications, implemented possibly using locks.

- **Sets:** Zing supports a set data structure. Sets are used to represent collections where the ordering of objects is not important (thus reducing the number of potentially distinct states Zing needs to explore).
- **Nondeterministic choice:** Zing supports a nondeterministic construct called choose. This construct can be used to non-deterministically pick an element out of a finite set of integers, enumeration values, or object references. The choose construct has several uses. For example, it can be used to abstract irrelevant data from the code, and soundly model conditionals that are dependent on such irrelevant data.

Below, we give a brief overview of some important features that will enable the reader to get started with Zing quickly.

1.1.1 Processes

Processes are the unit of concurrency in Zing. Processes may be created statically in the initial state of a model, or dynamically as the execution of a model proceeds. Each process has an entry point – the Zing method in which it begins execution. A process normally terminates execution by simply returning from its entry point method. Because Zing supports functions (methods), each process also has a stack of unbounded size.

1.1.2 Creating processes

Processes come into existence through one of the two methods described below.

1.1.2.1 activate methods

A method marked with `activate` results, in the initial state of the model, in a single process whose entry point is the method so decorated. A method marked in this way must meet the following qualifications:

- Its type must be `void`.
- It must have no parameters,
- It must also be marked as `static`.

To be useful, a Zing model must contain one or more `activate` methods.

1.1.2.2 async method calls

Processes may also be created dynamically via asynchronous method calls. A method may be called asynchronously if it meets the following requirements:

- Its type must be `void`.
- It may have only input parameters.

A method meeting these requirements may be called both synchronously and asynchronously in the same model. To invoke a method asynchronously, the method call is simply preceded by the `async` keyword. This results in a new process whose entry point is the given method. Once instantiated, there is no implicit connection or relationship between a process and its creator, although in many cases the processes will exchange data or synchronize their execution using the standard communication facilities of the Zing language.

1.1.3 Inter-process communication

Once a process is created, it executes concurrently with all other runnable processes in the Zing model. Zing will consider all useful¹ interleavings between the processes according to the semantics of the Zing statements executed by each process (as described in the later chapters of this document). In many cases, features are deliberately excluded from the Zing language precisely to make these semantics simpler and easier to understand and implement correctly. Zing processes may communicate with one another through shared memory, message queues (channels) or any combination thereof. Channels are typed FIFO queues for conveying messages of any type. The size of a channel is unbounded, but a number of techniques may be used to restrict the size of a channel during the execution of a model.

1.1.4 Synchronization

Zing provides a single blocking construct – the `select` statement. A `select` statement contains a variable number of “condition / statement” pairs, where the execution of a statement is gated by its associated condition (its “join statement”).

A join statement consists of one or more join conditions. If multiple join conditions are given, the statement is enabled only when all of the join conditions hold.

Join conditions come in two styles:

- Wait: The `wait` join condition allows a Zing process to block until an arbitrary Boolean expression evaluates to `true`. The `wait` expression often refers to the contents of shared memory or the size of a communication channel.
- Receive: The `receive` join condition is the only way in which a message may be received from a channel. The `receive` join condition specifies a channel, and a variable into which the message should be read. When the join statement is enabled (and selected for execution), all of its receive operations are completed.

1.1.5 End-states

The final state of a model is not necessarily one in which all processes have terminated. Sometimes, the proper final state of a “listener” process is one in which it is blocked awaiting the arrival of work. To distinguish between `select` statements that do, or do not, represent valid end states, Zing provides the `end` keyword which may appear between the `select` keyword and the list of join statements.

1.1.6 Blocking and atomicity

Some restrictions apply to the use of `select` statements within atomic blocks. A `select` statement may appear as the first statement of an atomic block in which case it acts as a “guard” on the execution of the block. When the `select` statement becomes runnable and its process is chosen for execution, it will execute the remainder of the atomic block in a single state transition.

It is also possible to use `select` statements elsewhere within an atomic block, but in this case it is required that at least one of the join statements within the `select` be runnable. Violations of this requirement are reported as errors by Zing. Once an atomic block begins execution (regardless of whether it was guarded by a `select` statement) it must be capable of running to its conclusion without blocking.

¹ Some interleavings are not considered useful in that they cannot result in the discovery of additional errors or behaviors. The theory behind this determination is beyond the scope of this document.

1.1.7 Non-determinism

Zing supports non-deterministic behavior (a useful facility for abstraction) in two ways. The `choose` operator allows for the explicit, non-deterministic selection from a set of values. When applied to an enumeration type, a range type, or the `bool` type, a selection is made from a set of values known statically at compile-time. When applied to an array or set variable, a selection is made dynamically from the contents of the variable at run-time. In both bases, the value of the `choose` operator is that of the selected value, and the model-checker will generate successor states corresponding to each of the possible values.

Another form of non-determinism arises from the `select` statement. If multiple join statements in a `select` are satisfied, Zing will, by default, consider the selection of each runnable join statement as a non-deterministic choice. If this behavior is not desired, the `first` qualifier may be added to the `select` statement in which case preference is always given to the join statement that appears first.

1.1.8 Symbolic execution

The `choose` operator results in an explicit case split during model checking. This, if we have a Zing statement “`x = choose(bool)`” this results in the model checker explicitly considering both the `true` and `false` assignments to `x`, in two separate states. If there are n such assignments, then such case splits result in 2^n states, which can be expensive for large values of n . Further, the `choose` operator works only over finite types. Applying a `choose` over an `int`, for example, is not allowed. An alternative modeling strategy in both these situations is to use symbolic types.

Zing supports three kinds of symbolic types, namely `symbolic int`, `symbolic bool`, and symbolic enumerations. A symbolic variable of any type initializes to an unknown, but fixed value. This enables modeling an arbitrary environment. The model checker does not do an explicit case split to analyze all possible initial values of these variables. Instead, it initializes these variables to symbolic values, and maintains relationships between the values as constraints. A theorem prover is used to prune-out paths where these constraints are inconsistent, thus eliminating false paths in the execution.

1.2 Examples

We illustrate the basic features of Zing through two classic examples. For each example, we outline the problem, and describe a model of the scenario written in Zing. The entire source code for example is included (without interruption) in appendix B. Finally, we show a toy example to illustrate symbolic execution.

1.2.1 Dining Philosophers

The “dining philosophers” problem is a demonstration of issues relating to concurrency and shared resources. The problem is described as follows: a group of N philosophers is seated around a circular table eating spaghetti. Each philosopher alternates between periods of thinking and eating. Between each pair of philosophers is a fork. To eat, a philosopher must obtain the forks on their left and right side, which they return to the table when they stop eating and begin to think.

We model this in Zing with an array of fork objects and an array of philosopher objects, arranged appropriately. First, we consider the fork class (below). A fork contains a reference to the philosopher who holds it or null if the fork is not currently in use. The “Pickup” method waits until the fork is not used and then marks it as being held by the given philosopher. An `atomic` block ensures the caller is not interrupted between the time that the fork becomes idle and the time at which it is marked as being used.

```
class Fork {
    Philosopher holder;
```

```

void Pickup(Philosopher eater) {
    atomic {
        select {
            wait(holder == null) -> holder = eater;
        }
    }
}

void PutDown() {
    holder = null;
}
};

```

The philosophers are modeled by the following class. A philosopher has a reference to the forks on their left and right side. (Note that each fork rests to the left of one philosopher and the right of another.) The “Run” method models the behavior of the philosopher. In this example, our philosophers pick up the fork on their left (waiting until it is available), and then the fork on their right. After eating, they return the forks to the table in the same order. Clearly, there are going to be problems with this simple-minded scheme, and the Zing model will discover this.

```

class Philosopher {
    Fork leftFork;
    Fork rightFork;

    void Run() {
        while (true) {
            // pick up forks
            leftFork.PickUp(this);
            rightFork.PickUp(this);
            // eat for a while
            leftFork.PutDown();
            rightFork.PutDown();
            // think for a while
        }
    }
};

```

To model the arrangement of five philosophers and five forks, we create a new array type for each.

```

array Philosophers[5] Philosopher;
array Forks[5] Fork;

```

ZING LANGUAGE SPECIFICATION

Finally, we need to initialize the arrays and set the philosophers loose to go about their business. We do this in a separate “Init” class. The static method “Run” is marked with the `activate` modifier which will cause it to begin execution in the initial state of the model. We first create an instance of each array, and then initialize the arrays by creating each philosopher and fork object. Then, we associate each philosopher with the forks to their left and right, and let them begin by calling their “Run” method asynchronously (thus creating a new process for each).

Because we’re only interested in the behavior of the philosophers once they are seated and the table is set, we enclose the initialization code in an `atomic` block. This way, we don’t need to consider what happens if the first philosopher is allowed to start before the others have been initialized.

```
class Init {
    activate static void Run() {
        Philosophers p;
        Forks f;
        int i;

        atomic {
            // Allocate the arrays of forks and philosophers
            p = new Philosophers;
            f = new Forks;

            // Allocate the individual fork and philosopher objects
            i = 0;
            while (i < sizeof(Philosophers)) {
                p[i] = new Philosopher;
                f[i] = new Fork;
                i = i + 1;
            }

            // Associate the philosophers with their forks and let them begin
            i = 0;
            while (i < sizeof(Philosophers)) {
                p[i].leftFork = f[i];
                p[i].rightFork = f[(i+1) % sizeof(Philosophers)];

                async p[i].Run();
                i = i + 1;
            }
        }
    }
};
```

When this Zing model is compiled, an assembly is generated that is suitable for consumption by the Zing model-checker. For this model, the model-checker will report that the system can become stuck and will generate an execution trace showing how this condition can be reached.

1.2.2 Alternating-bit protocol

The alternating-bit protocol is a simple data-transfer protocol. It implements the reliable transmission of data in one direction, from a sender to a receiver, using a pair of communication channels, both of which may be unreliable. Each message is appended with a single “protocol” bit. Messages are acknowledged by the receiver by sending an “ack” message containing a matching protocol bit. The name of the protocol comes from the fact that the protocol bit alternates between zero and one after each successful message transfer.

A simple Zing class is used to model each of the message and acknowledgement packets. The “body” field of the `Msg` class represents the information-bearing portion of the message. The Zing model will verify that this data is conveyed to the receiver without corruption, duplication, or message loss, and that the sender’s messages arrive in order. The “bit” field of each class is the “protocol” portion of the message.

```
class Msg {
    bool body;
    bool bit;
};

class Ack {
    bool bit;
};
```

We declare new channel types for the `Msg` and `Ack` objects. A channel is simply an ordered queue containing zero or more elements of the given type. The size of a queue is unbounded, but as we’ll see later, we can arrange for our model to restrict itself to a limited size.

```
chan MsgChan Msg;
chan AckChan Ack;
```

To verify that the message body reaches the receiver correctly, we’ll use a separate channel that’s effectively independent of the protocol implementation. It models what the sender and receiver would see through a perfectly reliable transport. It needs to carry only a single bit for each message, so we can declare a simpler channel for it.

```
chan BoolChan bool;
```

The sender is implemented as a class containing:

- a transmit channel (for messages)
- a receive channel (for acks)

ZING LANGUAGE SPECIFICATION

- a helper method to model the unreliable transmission of messages
- a main body (in the Run method).

During initialization we'll arrange for the outgoing channel of the sender process to be the same as the incoming channel of the receiver process, and vice-versa.

```
class Sender {
    static MsgChan xmit;
    static AckChan recv;
```

The TransmitMsg method takes a data bit and a protocol bit as parameters and sends them unreliably to the Receiver process. To model the possibility of message loss, we want to consider two distinct alternatives. One way to do this is to use the select statement as shown here. Recall that in the last example we used a select statement to block until a given expression became true. If a select statement contains multiple satisfied join statements it will (by default) select one of them in a non-deterministic fashion.² The Zing idiom shown here is the preferred mechanism for considering multiple possible paths of execution. If the first join statement is chosen, the message is delivered to the Receiver. The second join statement models message loss.

Another important Zing feature is illustrated here. The assume statement is used to avoid considering paths of execution that aren't interesting, or that we wish to avoid for some reason. Following an assume statement, we are assured that its predicate holds true. If an execution trace is encountered in which the condition does not hold, then we simply ignore that path. Unlike the assert statement, an assume failure is not treated as an error. In this case, the assume statement asserts that our transmission channel contains fewer than "Main.QueueSize" messages. Without this, Zing would have to consider execution paths in which the Sender transmits a message and then, not seeing an ack, continues to retry the transmission without end. In practice, a real protocol implementation would use a non-zero timeout or other control-flow features to prevent this, but for this simple example, we can use assume to effectively eliminate executions in which the Sender gets far ahead of the Receiver.

```
static void TransmitMsg(bool body, bool bit)
{
    Msg m;

    select {
        wait(true) -> {
            assume(sizeof(xmit) < Main.QueueSize);
            m = new Msg;
            m.body = body;
            m.bit = bit;
            send(xmit, m);
        }
        wait(true) -> /* lost message */ ;
    }
}
```

² It's convenient to think of this as a random selection, but in reality, the Zing model-checker will fully consider each of the alternatives.

```

    }
}

```

The Run method of the sender loops forever constructing new messages and sending them to the receiver³. Within this loop, the creation of new messages for transmission is modeled abstractly using Zing's choose operator. Given a suitable type⁴ as a parameter, the choose operator returns a member of the type's value-set non-deterministically. To verify that the message reaches the receiver reliably (and in order) we also transmit through a reliable channel. Next, the message is transmitted through the normal, unreliable transport and the sender begins to wait for an acknowledgement. The `atomic` block here insures that the message creation and initial transmission is done as a single step.

```

static void Run()
{
    bool currentBit = false;
    Ack a;
    bool body;
    bool gotAck;

    while (true) {
        atomic {
            body = choose(bool);
            send(Main.reliableChan, body);

            TransmittMsg(body, currentBit);

            gotAck = false;
        }
    }
}

```

Next, the sender begins waiting for an acknowledgement of the message from the receiver. The `while` loop will continue until a suitable ack message is received. The `select` statement waits until an acknowledgement is received, but also deals with the possibility that the message (or its acknowledgement) was lost. If we receive an ack, it is only considered valid if its protocol bit matches that of the message that was sent. Otherwise we continue to wait. If a timeout occurs, we simply transmit the message again and continue waiting for an ack. The `first` keyword on the `select` statement causes it to always select the first satisfied join statement. Because the `timeout` join condition is always satisfied, this gives preference to a pending ack message if one is available.⁵

```

while (!gotAck) {

```

³ Note that the infinite loop is not troubling here because the state-space of the model remains finite.

⁴ In addition to "bool", any enumeration or range type may be specified.

⁵ The model would still work correctly if the "first" modifier was omitted, but its state-space would be unnecessarily large. The simple interleaving of execution between the sender and receiver will insure that the timeout case is considered.

ZING LANGUAGE SPECIFICATION

```
atomic {
    select first {
        receive(recv, a) -> gotAck = (a.bit == currentBit);
        timeout -> TransmitMsg(body, currentBit);
    }
}
}
```

Finally, after the message is acknowledged we toggle the current protocol bit before proceeding with the next message.

```
currentBit = !currentBit;
}
}
};
```

The receiver class is similar to the sender. It has a receive channel for messages and a transmit channel for acknowledgements. Its TransmitAck method delivers acknowledgements unreliably to the sender.

```
class Receiver {
    static MsgChan recv;
    static AckChan xmit;

    static void TransmitAck(bool bit)
    {
        Ack a;

        select {
            wait(true) -> {
                a = new Ack;
                a.bit = bit;
                send(xmit, a);
            }
            wait(true) -> /* lost ack */ ;
        }
    }
}
```

The Run method of the receiver loops forever waiting for messages from the sender and then consuming them. The loop begins by blocking until a message is received from the sender. A received message is always acknowledged by sending an ack with the same protocol bit value. If the protocol bit received is the next “expected” bit, then the message is “new” and should be consumed. When a message is consumed, we get its

counterpart from the reliable channel and verify that the message body value matches. Then, the value of the expected protocol bit is toggled. Here again, atomic blocks are used to eliminate potential interleavings due to statements that are purely internal to the process.

```

static void Run()
{
    bool expectedBit = false;
    bool trueBody;
    Msg m;

    // Loop forever consuming messages
    while (true) {
        select { receive(recv, m) -> ; }

        atomic {
            // Always send an ack with the same bit
            TransmitAck(m.bit);

            if (expectedBit == m.bit) {
                // Consume the message here and verify it's body matches
                // what we received through the reliable channel
                select { receive(Main.reliableChan, trueBody) -> ; }
                assert(trueBody == m.body);

                expectedBit = !expectedBit;
            }
        }
    }
};

```

As in the dining philosophers example, a separate class is used to hold global data and initialize the model. The Run method initializes the reliable message channel, and then creates channels for the the unreliable transmission of messages and acknowledgements. The new channels are used to initialize the xmit and recv fields of the sender and receiver as appropriate. Finally, the Run methods of both the sender and receiver are called asynchronously to begin execution of the model.

```

class Main {
    static int QueueSize = 2;
    static BoolChan reliableChan;

```

```
activate static void Run()
{
    atomic {
        reliableChan = new BoolChan;

        Sender.xmit = Receiver.recv = new MsgChan;
        Sender.recv = Receiver.xmit = new AckChan;

        async Sender.Run();
        async Receiver.Run();
    }
}
};
```

1.2.3 Symbolic execution

We illustrate symbolic execution using a simple example. Suppose we have three integers x , y and z that can take arbitrary values assigned by some environment. The Zing model below checks if x is less than or equal to y , and if y is less than or equal to z , and then asserts that x is less than z .

```
class SymbolicTest {
    static symbolic int x;
    static symbolic int y;
    static symbolic int z;

    activate static void Run()
    {
        if (x <= y)
        {
            if (y <= z)
                assert(x < z); // assertion can fail
        }
    }
};
```

If x , y and z are equal, then the conditionals of the `if` statements can hold, and the `assert` statement can fail. The Zing model checker can detect this possibility using symbolic execution, without explicitly reasoning about all the different combinations of x , y and z . If we make one of the conditionals to be a strict inequality, as illustrated in the program below, then the Zing model checker is able to prove that the `assert` statement can never fail.

```
class SymbolicTest {
    static symbolic int x;
    static symbolic int y;
    static symbolic int z;

    activate static void Run()
    {
        if (x <= y)
        {
            if (y < z)
                assert(x < z);    // assertion cannot fail
        }
    }
};
```

2. Lexical Structure

The lexical (and syntactic) structure of Zing is heavily influenced by C#. For a detailed description of the C# lexical structure as well as the grammar notation use here, please refer to chapter 2 of the C# language specification.

In the remainder of this chapter, we highlight differences between the lexical structure of C# and Zing.

2.1 Models

A Zing **model** consists of one or more source files. From a set of source files, the Zing compiler produces a single .Net assembly suitable for use with other Zing tools such as the model-checker.

To be executable, a Zing model must include at least one static method definition qualified with the `activate` keyword.

2.2 Tokens

Zing does not currently support character literals as tokens.

2.3 Keywords

The following keywords are reserved and may not be used as an identifier.

keyword: one of

activate	array	assert	assume	async
atomic	bool	byte	chan	choose
class	decimal	double	else	enum
end	event	false	first	float
foreach	goto	if	in	int
long	new	null	object	out
range	raise	receive	return	sbyte
select	send	set	short	sizeof
static	struct	symbolic	this	timeout
trace	true	try	uint	ulong
ushort	visible	void	wait	while
with				

2.4 Operators and punctuators

The following character sequences are recognized as operators and punctuators by the Zing compiler.

operator-or-punctuator: one of

{	}	[]	()	.	,	:	;
+	-	*	/	%	&		^	!	=
<	>	..	&&		<<	>>	==	!=	->
<=	>=								

2.5 Pre-processing

The pre-processing facilities of Zing are identical to C#.

3. Types

A Zing source file is a collection of type definitions. In Zing, types are either *simple* or *complex*, the primary difference being that complex types are allocated on the heap, and simple types are not. This chapter contains a brief overview of the various types supported by Zing. More detailed information is available in later chapters of this document.

3.1 Simple Types

Simple Zing types include the predefined types, enumerations and ranges, which are effectively subtypes of `int`; and structures, which are collections of simple types or references to complex types.

3.1.1 Predefined types

Zing supports all of the C# predefined types with the exception of `char` and `string`.

3.1.2 Enumerations

An enumeration is a distinct type that declares a set of named constants. These work much like C# except that enumeration members may not be assigned specific integer values.

3.1.3 Symbolic types

Zing supports three kinds of symbolic types, namely `symbolic bool`, `symbolic int`, and symbolic enumerations. We refer to “non-symbolic” types as concrete.

Variables of concrete types initialize to default values (0 in the case of `int`, and `false` in the case of `bool`) and variables of symbolic types initialize to arbitrary values. The model checker uses symbolic execution to consider all possible initial values, and uses a theorem prover to prune out infeasible execution paths.

We refer to the values held by symbolic and concrete types as symbolic values and concrete values respectively. Concrete values may be assigned to either concrete variables or symbolic variables. Symbolic boolean values may be assigned to concrete boolean variables (Zing handles this by introducing a case-split), but symbolic integer values cannot be assigned to concrete integer variables (since this would require an unbounded case-split). Such implicit conversions from concrete values to symbolic values and vice-versa (with above restrictions) cannot occur in the same statement as a method call.

Symbolic enum values cannot be assigned to concrete enum variables, but this will likely be permitted in a future release.

Classes and structs may have members of symbolic types, and methods can have symbolic parameters or return values. Currently, arrays, sets and channels of symbolic types are not supported, and symbolic ints cannot index arrays.

3.1.4 Ranges

A range type is a distinct type that declares a range of integer values.

Caveat: Range types are currently only useful in combination with the `choose` operator, the result of which should be assigned to a variable of an integral type. While it should be possible to declare and use variables of a range type, this is not currently supported.

3.1.5 Structures

A structure type defines a collection of fields of simple types or references to complex types. A structure may be declared as a field of another structure. Structure declarations may appear:

1. As static members of a Zing class. In this case the structure is statically allocated as part of the state's "global" section.
2. As instance members of a Zing class. The structure appears in the heap as part of the class instance.
3. As the element type of a Zing set, array, or channel. The structure appears in the heap as part of its container type.
4. As a parameter or local variable in a method. The structure is allocated on a Zing process stack as part of the method's call frame.

Caveat: Structures should be generally avoided for now. There are known issues related to channels, sets, or arrays of structures – and probably other unknown problems as well. Until these bugs are addressed, classes should be strongly preferred over structures.

3.2 Complex Types

Complex types are allocated from the Zing heap. All complex objects are allocated using the `new` operator.

3.2.1 Arrays

Zing currently supports a subset of the array syntax shown in the BNF of Appendix A. Arrays must currently be of a fixed size, and may be indexed only with an integer type. Support for variable-size arrays and indexing by enumeration and range types will likely be added in a future release.

3.2.2 Sets

A set is a homogeneous, unordered collection of elements. The size of a set is unbounded. One may iterate over the contents of a set (using the `foreach` statement), but array-style indexing is not permitted. Detailed information on the available set operations is provided in chapter 6.

3.2.3 Channels

A channel is a homogeneous, ordered queue of elements. The size of a channel is unbounded. One may send a message to a channel, receive a message from a channel, or obtain the number of messages currently residing in the channel.

3.2.4 Classes

Zing supports a very simple notion of class. Zing classes do not support: inheritance, overloading of any kind, constructors, or access modifiers. Class definitions may not be nested.

Classes may contain static or non-static (i.e. instance) declarations of both fields and methods. Fields may include initializers provided the initialization expression is simple – they may not include method calls or the `choose` operator.

3.2.5 object type

The generic `object` type may be used in place of a strongly-typed declaration. Zing does not support typecasts or a `typeof` operator. Any complex type reference may be assigned to a variable of type `object`. An `object` value may be assigned to a strongly-typed variable, which is effectively a typecast to the target type. If the object instance is not of the target type, a runtime error will be reported. This facility can be used to implement a

kind of “poor man’s polymorphism” but it is up to the Zing author to ensure that runtime errors do not result. This often calls for the use of a wrapper class and enumeration definition.

One may not create an instance of the object type.

3.3 Instantiation and initialization

The predefined types are initialized to their default values (false for bool, zero for int or byte) unless an initialization expression is applicable. The default value of a complex type reference is null.

The fields of a struct type may include an initialization expression which will be applied when the struct is initialized.

All complex objects are instantiated using the new operator. Sets and channels are empty upon creation. An array is filled with the default value of its element.

For classes, the fields of the new instance are initialized with their default values unless an initialization expression is given. In that case, the initialization expression is executed atomically as part of the new operator.

4. Enumerations

As in C#, an enum type is a distinct type that declares a set of named constants.

The example

```
enum Col or
{
    Red,
    Green,
    Yel l ow
};
```

declares an enum type named Col or with members Red, Green, and BI ue.

4.1 Enum declarations

An enum declaration declares a new enum type. It begins with the keyword `enum`, and defines the members of the enum.

enum-declaration:

```
enum identifier enum-body ;
```

enum-body:

```
{ enum-member-declarations }
{ enum-member-declarations , }
```

Enum member declarations are separated by the comma character, and a comma is permitted but not required after the last one. Both of the enum declarations in the following example are valid.

```
enum Col or1
{
    Red,
    Green,
    Yel l ow
};
enum Col or2
{
    Red,
    Green,
    Yel l ow,
};
```

4.2 Enum members

The body of an enum type declaration defines one or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

enum-member-declarations:

```
enum-member-declaration
enum-member-declarations , enum-member-declaration
```

enum-member-declaration:

```
identifier
```

Note that unlike C#, the constant value of the enum members may not be set explicitly.

4.3 Enum values and operations

No conversion is permitted between an enum type and other integral types (or other enums). The following operators may be used on the values of an enum type: ==, !=, <, >, <=, and >=.

4.4 Non-deterministic selection

An enum type name may be used as the operand of the choose operator. The result of this expression is a value of the enum type. In the following example, the Zing model makes a non-deterministic selection from the Col or enum. The model-checker will consider the execution paths that result from each of the possible values of Col or.

```
Col or stopLi ghtState;  
stopLi ghtState = choose(Col or):  
i f (stopLi ghtState == Col or. Green)  
  . . .  
el se i f (stopLi ghtState == Col or. Red)  
  . . .  
el se  
  . . .
```

4.5 Symbolic enums

To declare a variable as a symbolic enum, the symbol i c keyword is used. The example above could be written without an explicit case-split using a symbolic enum.

```
symbol i c Col or stopLi ghtState;  
i f (stopLi ghtState == Col or. Green)  
  . . .  
el se i f (stopLi ghtState == Col or. Red)  
  . . .  
el se  
  . . .
```

Ranges

A range type is a distinct type that declares a range of integer values.

The example

```
range NumItems 0 .. 5;
```

declares a range type named NumItems with the possible values 0, 1, 2, 3, 4, and 5.

4.6 Range declarations

A range declaration creates a new range type. It begins with the keyword `range` followed by the lower and upper limits of the range, separated by “`..`”.

```
range-declaration:  
range identifier constant-expression .. constant-expression ;
```

4.7 Range variables and operations

Currently, Zing does not support instances of range types. They may only be used as the operand for the `choose` operator.

4.8 Non-deterministic selection

A range type name may be used as the operand of the `choose` operator. The result is a value of type `int`, chosen from the possible values of the range type. The model-checker will consider all possible value selections from the range.

5. Arrays

An array type is a distinct type that declares an ordered, indexable container of elements. The elements of an array are all of the same type, and this type is referred to as the element type of the array. The size of an array may be fixed or variable. Arrays are indexed by a zero-based integer expression.

The element type of an array can be any type, including an array type.

The example

```
array TwoBools[2] bool ;
```

declares an array type named `TwoBools` containing two elements of type `bool`.

5.1 Array declarations

An array declaration declares a new array type. An array declaration begins with the keyword `array` followed by the name of the array type, an optional specification of its size, and finally its element type.

array-declaration:

```
array identifier [ constant-expression ] type ;  
array identifier [ ] type ;
```

In the first declaration form, the constant expression must be of type `int`, declares a fixed-size array of elements. The number of elements is specified by the constant expression, which must be of type `int`. The elements of the array are indexed by integer values ranging from zero to (but not including) the value of the given expression.

In the second declaration form, the size is unspecified and will be provided dynamically when the array is constructed.

5.2 Array variables and operations

5.2.1 Construction

Arrays are constructed using the `new` operator. During construction, the elements of the array are initialized to their default value.

Example:

```
TwoBools listA;  
listA = new TwoBools;
```

Dynamic arrays are constructed by providing an additional size expression enclosed in square brackets. The expression must be of type `int`, or of a type which can be implicitly coerced to `int`.

Example:

```
Array ManyBools[] bool ;  
ManyBools flagList;  
flagList = new ManyBools[count];
```

5.2.2 Indexing

The indexing operator `[]` is used to access the elements of an array.

Example:

```
TwoBools listA;  
listA = new TwoBools;  
listA[0] = choose(bool);  
listA[1] = !listA[1];
```

5.2.3 Iteration

It is possible to iterate over the contents of an array using the `foreach` statement. The type of the iteration variable in a `foreach` statement must be the same as the element type of the array.

Example:

```
foreach (bool b in listA)  
{  
    ...  
}
```

5.2.4 Sizeof

The `sizeof` operator may be applied to an array reference or an array type name to return the number of elements in the array.

Example:

```
assert(sizeof(TwoBools) == 2);  
assert(sizeof(listA) == 2);
```

Note: It is an error to apply the `sizeof` operator to an array type name that refers to a dynamically-sized array.

5.2.5 Non-deterministic selection

A reference to an array instance may be used as the operand for the `choose` operator. The result is a non-deterministic selection from the elements of the array. The resulting expression type is that of the array's element type.

Example:

```
bool someElement;  
someElement = choose(listA);
```

6. Sets

A set type is a distinct type that declares an unordered collection of elements. The number of elements in a set is unbounded and the elements must be of the same type.

The example

```
set SmallInts byte;
```

declares a set type named `SmallInts` whose elements are of type `byte`.

6.1 Set declarations

A set declaration declares a new set type. The set declaration begins with the `set` keyword followed by the name and element type of the set.

```
set-declaration:  
set identifier type ;
```

6.2 Set variables and operations

Note: the set operations described in this section are the only forms currently supported by the Zing compiler. Future versions of the compiler may support more operators, or more flexible use of the existing operators.

6.2.1 Construction

Set instances are created using the `new` operator. Sets are empty following construction.

Example:

```
SmallInts myInts;  
myInts = new SmallInts;
```

6.2.2 Adding an element

To add an element to a set, use the `+` operator in an assignment of the form:

```
s = s + e;
```

or

```
s = e + s;
```

where `s` refers to a set and `e` is an expression whose type matches that of the set's element type.

6.2.3 Removing an element

To remove an element from a set, use the `-` operator in an assignment of the form:

```
s = s - e;
```

where `s` refers to a set and `e` is an expression whose type matches that of the set's element type.

6.2.4 Set addition

The elements of one set may be added to another using the `+` operator:

```
s1 = s1 + s2;
```

where s_1 and s_2 refer to sets of the same type.

6.2.5 Set subtraction

Set subtraction is supported via the $-$ operator:

```
s1 = s1 - s2;
```

where s_1 and s_2 refer to sets of the same type.

6.2.6 Testing membership

The `in` operator may be used to test for the presence of a particular value in a set:

```
if (e1 in s || e2 in s)
{
    ...
}
```

where s refers to a set, and both e_1 and e_2 are expressions whose type is that of the set's element type.

6.2.7 Testing cardinality

The `sizeof` operator returns the number of elements in a set:

```
if (sizeof(s) > 0)
{
    ...
}
```

6.2.8 Iteration

It is possible to iterate over the members of a set using the `foreach` statement. The type of the iteration variable in a `foreach` statement must be the same as the element type of the set.

Example:

```
foreach (int i in myIntSet)
{
    ...
}
```

6.2.9 Non-deterministic selection

A reference to a set instance may be used as the operand for the `choose` operator. The result is a non-deterministic selection from the members of the set. The resulting expression type is that of the set's element type.

Example:

```
int someElement;
someElement = choose(setOfInts);
```

Applying the `choose` operator to an empty set results in a Zing runtime error.

7. Channels

A channel type is a distinct type that declares a FIFO message queue of unbounded size. The messages in a channel must all be of the same Zing type.

The example

```
chan IntChan int;
```

declares a channel type named `IntChan` which contains messages of type `int`.

7.1 Channel declarations

A channel declaration declares a new channel type. The declaration begins with the `chan` keyword followed by the name and element type of the channel.

```
channel-declaration:  
chan identifier type ;
```

7.2 Channel variables and operations

7.2.1 Construction

Channel instances are created using the `new` operator.

Example:

```
IntChan myIntChan;  
myIntChan = new IntChan;
```

7.2.2 Sending a message

The `send` statement (section 10.12.1) is used to send a message to a channel.

7.2.3 Receiving a message

Messages are received from a channel using a `select` statement (section 10.12.2) with a `receive` join condition.

7.2.4 Obtaining the queue length

The `sizeof` operator returns the number of messages in a channel. This is often useful for bounding the size of a channel, either by forcing senders to block or by using an `assume` statement to simply ignore interleavings in which message producers outpace consumers.

Example:

```
atomic {  
    select {  
        wait (sizeof(myIntChan) < 3) -> send(myIntChan, x);  
    }  
}
```

8. Classes

Zing classes are very simple compared to other object oriented languages. Zing classes do not support inheritance, constructors, overloading, nesting, or access qualifiers (on the class or its members). A Zing class is a heap-allocated data structure that may contain data members (fields) and function members (methods).

8.1 Class declarations

A class declaration declares a new Zing class. It consists of the `class` keyword followed by an identifier that names the class and a set of member declarations enclosed in braces.

```
class-declaration:
  class identifier class-body ;
class-body:
  { class-member-declarationsopt }
```

8.2 Class members

Class members consist of field declarations and method declarations appearing within the class body. The order of the member declarations is not significant. Member names must be unique within the class.

```
class-member-declarations:
  class-member-declaration
  class-member-declarations class-member-declaration
class-member-declaration:
  field-declaration
  method-declaration
```

8.2.1 Static and instance members

Members of a class are either *static members* or *instance members*. It is useful to think of static members as belonging to classes, and instance members as belonging to objects (instances of classes).

When a field or method declaration includes the `static` modifier, it declares a static member. Static members have the following characteristics:

- When a static member is referenced in a member access of the form `E.M`, `E` must denote a type that has a member `M`. It is a compile-time error for `E` to denote an instance.
- A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.
- A static function member does not operate on a specific instance, and it is a compile-time error to refer to `this` in such a function member.

When a field or method declaration does not include a static modifier, it declares an instance member. Instance members have the following characteristics:

- When an instance member is referenced in a member access of the form `E.M`, `E` must denote an instance of a type that has a member `M`. It is a compile-time error for `E` to denote a type.

- Every instance of a class contains a separate set of all instance fields of the class.
- An instance function member operates on a given instance of the class, and this instance can be accessed as `this`.

8.3 Fields

A **field** is a member that represents a variable associated with an object or class. A field declaration introduces a field of a given type.

```

field-declaration:
    field-modifiersopt type variable-declarator ;

field-modifiers:
    field-modifier
    field-modifiers field-modifier

field-modifier:
    static

variable-declarator:
    identifier
    identifier = variable-initializer

variable-initializer:
    expression

```

A field declaration begins (optionally) with a static modifier. The *type* of a field declaration specifies the type of the member introduced by the declaration. The type is followed by an identifier naming the field. This may be followed by an “=” token and a *variable-initializer* that gives the initial value of the member.

8.3.1 Field initialization

The initial value of a field is the default value of the field’s type.

8.3.2 Variable initializers

Field declarations may include *variable-initializers*. For static fields, variable initializers correspond to assignment statements that are executed prior to the initial state of the Zing model. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created (atomically within the context of the new operator’s execution). It is a compile-time error for the variable initializer for a field to include a function call or the choose operator.

8.4 Methods

A **method** is a member that implements a computation or action that can be performed by an object or class. Methods are declared using *method-declarations*.

```

method-declaration:
    method-header method-body

method-header:
    method-modifiersopt return-type identifier ( formal-parameter-listopt )

method-modifiers:
    method-modifier
    method-modifiers method-modifier

```

method-modifier:

acti vate
atomi c
stati c

return-type:

type
voi d

method-body:

block

A *method-declaration* may include any valid combination of the `acti vate`, `atomi c`, and `stati c` modifiers. The only restriction that must be observed with respect to method modifiers is that `acti vate` may only be specified on a method that is also marked as `stati c`.

The *return-type* of a method declaration specifies the type of the value computed and returned by the method and is `voi d` if the method does not return a value.

The *identifier* following the *return-type* specifies the name of the method.

The optional *formal-parameter-list* specifies the parameters of the method.

8.4.1 Method parameters

The parameters of a method, if any, are declared by the method's *formal-parameter-list*.

formal-parameter-list:

parameters

parameters:

parameter
parameters , parameter

parameter:

parameter-modifier_{opt} type identifier

parameter-modifier:

out

The formal parameter list consists of one or more comma-separated parameters. Each parameter consists of an optional `out` modifier followed by the type and name of the parameter.

A method declaration creates a separate declaration space for parameters and local variables. Names are introduced into this declaration space by the formal parameter list of the method and by local variable declarations in the *block* of the method. All names in the declaration space of a method must be unique. It is a compile-time error for a parameter or local variable to have the same name as another parameter or local variable.

A method invocation creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple-name* expressions.

There are two kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Output parameters, which are declared with the `out` modifier.

8.4.1.1 Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter — they have no effect on the actual argument given in the method invocation.

8.4.1.2 Output parameters

A parameter declared with an `out` modifier is an output parameter. An output parameter does not create a new storage location. Instead, an output parameter logically represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable-reference* of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

Output parameters are typically used in methods that produce multiple return values.

8.4.2 Static and instance methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this`.

When a method is referenced in a *member-access* of the form `E.M`, if `M` is a static method, `E` must denote a type containing `M`, and if `M` is an instance method, `E` must denote an instance of a type containing `M`.

8.4.3 Atomic methods

When a method declaration includes the `atomic` modifier, all invocations of the method will be performed atomically. The effect is the same as if each invocation of the method was enclosed in an `atomic` block. The statements in the body of an `atomic` method must conform to all of the restrictions that apply within an `atomic` block.

Note that this is **not** the same as enclosing the entire body of the method in an atomic block, as this would not include the method invocation and return within the scope of the atomic execution.

8.4.4 Activated methods

When a method declaration includes the `activate` modifier, the initial state of the Zing model will include a single process whose entry point is the method being declared. The `activate` modifier may only be applied to methods with a *return-type* of `void`, and with no formal parameters. Only one instance of the process can be created with this facility. All Zing models must include at least one `activate` method.

8.4.5 Method body

The *method-body* of a method declaration consists of a *block* that contains the statements to execute when that method is invoked.

When the return type of a method is `void`, `return` statements in that method's body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its caller.

When the return type of a method is not `void`, each `return` statement in that method's body must specify an expression of a type that is implicitly convertible to the return type. The endpoint of the method body of a value-returning method must not be reachable. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.

9. Structs

Structs are similar to classes in that they represent data structures that can contain data members. However, unlike classes (and other complex types), structs are value types and do not require heap allocation. They also may not contain function members. A variable of a struct type directly contains the data of the struct, whereas a variable of a complex type contains a reference to the data.

Warning: structs are not fully implemented in the current release of Zing and their use is not recommended.

9.1 Struct declarations

A struct declaration declares a new struct type.

```
struct-declaration:
    struct identifier struct-body ;

struct-body:
    { struct-member-declarationsopt }

struct-member-declarations:
    struct-member-declaration
    struct-member-declarations struct-member-declaration

struct-member-declaration:
    field-declaration
```

A *struct-declaration* consists of the keyword `struct` and an *identifier* that names the struct, followed by a *struct-body*, followed by a semicolon. The *struct-body* contains zero or more field declarations, which have the same form as field declarations in a class, except that the `static` modifier may not be used.

9.2 Fields

Struct fields are declared and initialized just as class fields, with the exception that the `static` modifier may not be applied to a field of a struct.

10. Statements

Zing supports a variety of statements. Many of these will be familiar to users of languages like C++ or C#, but others are tailored to the unique needs of a modeling language.

```

statement:
    labeled-statement
    declaration-statement
    embedded-statement

embedded-statement:
    attributed-statement
    block
    atomic-block
    empty-statement
    expression-statement
    if-statement
    iteration-statement
    jump-statement
    try-statement
    assert-statement
    assume-statement
    async-call-statement
    send-statement
    select-statement
    trace-statement
    event-statement

```

The *embedded-statement* nonterminal is used for statements that appear within other statements. The use of *embedded-statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. The example

```

void F(bool b) {
    if (b)
        int i = 44;
}

```

results in a compile-time error because an `if` statement requires an *embedded-statement* rather than a *statement* for its if branch. Note, however, that by placing `i`'s declaration in a block, the example is valid.

10.1 Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

```

block:
    { statement-listopt }

```

A *block* consists of an optional *statement-list*, enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements. The scope of a local variable declared in a block is that of the method in which the block appears. *Note:* this is different from the semantics of C++ or C#.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

10.1.1 Statement Lists

A statement list consists of one or more statements written in sequence. Statement lists occur in *blocks*.

```
statement-list:  
    statement  
    statement-list statement
```

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

10.2 Labeled statements

A *labeled-statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

```
labeled-statement:  
    identifier : statement
```

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from goto statements within the scope of the label. This means that goto statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

```
int F(int x) {  
    if (x >= 0) goto x;  
    x = -x;  
    x: return x;  
}
```

is valid and uses the name *x* as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

10.3 Attributed statements

An *attributed-statement* permits a statement to be prefixed by a set of attributes. Statement attributes are used for two general purposes: 1) for correlation of Zing statements with foreign source code, and 2) for the annotation of execution traces with application-specific information. While the infrastructure for attributes is designed to be extensible, only two pre-defined attributes are supported in the current release. User-defined attributes should be supported in the future.

An attributed statement has precisely the same execution semantics as an identical, but un-attributed, statement.

attributed-statement:
attributes statement

attributes:
attribute-sections

attribute-sections:
attribute-section
attribute-sections attribute-section

attribute-section:
 [*attribute-list*]
 [*attribute-list* ,]

attribute-list:
attribute
attribute-list , *attribute*

attribute:
attribute-name attribute-arguments_{opt}

attribute-name:
identifier

attribute-arguments:
 (*attribute-argument-list*)

attribute-argument-list:
attribute-argument
attribute-argument-list , *attribute-argument*

attribute-argument:
expression

10.3.1 Attribute types

Zing attributes come in two varieties – context attributes and trace attributes. Context attributes are generally intended for use in correlating Zing statements with some other source environment. This allows Zing error traces to be rendered directly in the context from which the Zing model was extracted. Trace attributes serve the same purpose as (and largely replace) the Zing trace statement.

When an attributed statement is executed by the Zing runtime, a Zing attribute event is generated for each attribute and is attached to the resulting State object. This occurs for both context and trace attributes.

In addition, if a statement is preceded by a context attribute, then the runtime object model makes available the associated context attribute as a property on the appropriate process information object.

Because of their special use by the Zing runtime, the arguments to a context attribute must be literal expressions.

10.3.2 Context attributes

The only context attribute currently supported is SourceContextAttribute (as with C#, the attribute may be referred to as either SourceContext or SourceContextAttribute). The SourceContext attribute takes three parameters, a string and two integers. These typically denote the file name, starting line number, and ending line number of the source context:

```
[SourceContext("foo.c", 14, 14)]
x = x + 1;
```

10.3.3 Trace attributes

Trace attributes should generally be used in preference to trace statements. Unlike a trace statement, a trace attribute has no impact on the behavior of the Zing model. The only trace attribute supported in the current release is Trace, which accepts a string argument followed by a set of additional, optional arguments. Like the trace statement, the first argument is considered, by convention, to be a .Net-style format string for the remaining arguments although this is not strictly required:

```
[Trace("x.f={0}, g={1}", x.f, g)]
x.f = g;
```

10.4 The empty statement

An empty-statement does nothing, but does constitute a real, albeit trivial, state transition in the Zing model.

```
empty-statement:
;
```

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement.

An empty statement can be used when writing a while statement with a null body:

```
bool ProcessMessage() { . . . }
void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

Also, an empty statement can be used to declare a label just before the closing "}" of a block:

```
void F() {
    . . .
    if (done) goto exit;
    . . .
    exit: ;
}
```

10.5 Declaration statements

A *declaration-statement* declares a local variable. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

```
declaration-statement:
    local-variable-declaration ;
```

```
local-variable-declaration:
    type variable-declarator
```

```
variable-declarator:
    identifier
    identifier = expression
```

The *type* of a *local-variable-declaration* specifies the type of the variables introduced by the declaration. The type is followed by a *variable-declarator*, which introduces a new variable. A *local-variable-declarator* consists

of an *identifier* that names the variable, optionally followed by an "=" token and an *expression* that gives the initial value of the variable.

The value of a local variable is obtained in an expression using a *simple-name*, and the value of a local variable is modified using an assignment. A local variable must be definitely assigned at each location where its value is obtained.

The scope of a local variable declared in a local-variable-declaration is the method in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable with the same name.

A variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
void F() {  
    int x = 1;  
}
```

corresponds exactly to

```
void F() {  
    int x; x = 1;  
}
```

10.6 Expression statements

An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement:
statement-expression ;

statement-expression:
invocation-expression
assignment

Not all expressions are permitted as *statements*. In particular, expressions such as $x + y$ and $x == 1$ that merely compute a value (which will be discarded), are not permitted as *statements*.

Execution of an *expression-statement* evaluates the contained *expression* and then transfers control to the end point of the *expression-statement*.

10.7 The if statement

The `if` statement selects a statement for execution based on the value of a Boolean expression.

if-statement:
`if (boolean-expression) embedded-statement`
`if (boolean-expression) embedded-statement else embedded-statement`

boolean-expression:
expression

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Thus, an `if` statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {  
    if (y) {  
        F();  
    }  
    else {  
        G();  
    }  
}
```

An `if` statement is executed as follows:

- The *boolean-expression* is evaluated.
- If the Boolean expression yields `true`, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the Boolean expression yields `false` and if an `else` part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the Boolean expression yields `false` and if an `else` part is not present, control is transferred to the end point of the `if` statement.

10.8 Iteration statements

Iteration statements repeatedly execute an embedded statement.

```
iteration-statement:  
    while-statement  
    foreach-statement
```

10.8.1 The `while` statement

The `while` statement conditionally executes an embedded statement zero or more times.

```
while-statement:  
    while ( boolean-expression ) embedded-statement
```

A `while` statement is executed as follows:

- The *boolean-expression* is evaluated.
- If the Boolean expression yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded, control is transferred to the beginning of the `while` statement.
- If the Boolean expression yields `false`, control is transferred to the end point of the `while` statement.

Note that Zing does not currently support `break` or `continue` statements. Equivalent behavior can be achieved using `goto` statements.

10.8.2 The foreach statement

The foreach statement enumerates the elements of a set or array, executing an embedded statement for each element of the collection.

foreach-statement:
 foreach (*type identifier in expression*) *embedded-statement*

The *type* and *identifier* of a foreach statement declare the iteration variable of the statement. The iteration variable corresponds to a read-only local variable. During execution of a foreach statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable through assignment or pass the iteration variable as an out parameter.

The type of the *expression* of a foreach statement must be a set or array reference, and the element type of the collection must match that of the type of the iteration variable. If *expression* has the value null, a runtime error is reported.

10.9 Jump statements

Jump statements unconditionally transfer control.

jump-statement:
goto-statement
return-statement
raise-statement

The location to which a jump statement transfers control is called the target of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to exit the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

10.9.1 The goto statement

The goto statement transfers control to a statement that is marked by a label.

goto-statement:
 goto *identifier* ;

The target of a goto statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control *out of* a nested scope, but not *into* a nested scope.

10.9.2 The return statement

The return statement returns control to the caller of the function member in which the return statement appears.

return-statement:
 return *expression_{opt}* ;

A return statement with no expression can be used only in a method with the return type void.

A return statement with an expression can only be used in a method with a non-void return type. An implicit conversion must exist from the type of the expression to the return type of the containing function member.

A return statement is executed as follows:

- If the return statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.
- Control is returned to the caller of the containing function member.

10.9.3 The raise statement

The raise statement raises a named exception.

```
raise-statement:
    raise identifier ;
```

Exceptions in Zing are simply globally-scoped names. Exceptions (like labels) are not declared explicitly.

When an exception is raised, control is transferred to the first with clause in an enclosing try statement that can handle the exception. The process that takes place from the point of the exception being raised to the point of transferring control to a suitable exception handler is known as exception propagation. Propagation of an exception consists of repeatedly evaluating the following steps until a with clause that matches the exception is found. In this description, the raise point is initially the location at which the exception is raised.

- In the current function member, each try statement that encloses the raise point is examined. For each statement S, starting with the innermost try statement and ending with the outermost try statement, the following steps are evaluated:
- If the try block of S encloses the raise point and if S has one or more with clauses, the with clauses are examined in order of appearance to locate a suitable handler for the exception. The first with clause whose name matches the raised exception is considered a match. A general with clause is considered a match for any exception name. If a matching with clause is located, the exception propagation is completed by transferring control to the statement of that with clause.
- If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
- If the exception processing terminates all function member invocations in the current process, indicating that the process has no handler for the exception, then a runtime error is reported.

Note that exceptions as described here are different from runtime errors like ZingDivideByZeroException or ZingNullReferenceException, which cannot be caught or handled in any way.

10.10 The try statement

The try statement provides a mechanism for catching exceptions that occur during execution of a block.

```
try-statement:
    try block with { with-clauses }

with-clauses:
    with-clause
    with-clauses with-clause

with-clause:
    identifier -> embedded-statement
    * -> embedded-statement
```

ZING LANGUAGE SPECIFICATION

If an exception is raised during the execution of the `try` statement's *block*, control may be transferred to one of the `with` clauses as described in section 10.9.3. A `with` clause using the token "*" instead of an exception name is a *general* `with` clause and will match any raised exception.

Runtime errors like `ZingNullReferenceException` are not Zing exceptions and cannot be caught in a `try` statement.

10.11 The `async` statement

The `async` statement creates a new Zing process (thread) through the asynchronous invocation of a given method.

```
async-call-statement:
  async invocation-expression ;
```

The *invocation-expression* provides for the invocation of any static or instance method in a Zing class or object, respectively. The `async` statement requires only that the method have a return type of `void`, and that it includes no output parameters.

After the `async` statement executes, a new (runnable) process exists whose entry point is the specified method. There is no special relationship between the new process and its creator. Any communication or synchronization between the processes must be explicitly programmed by the author of the Zing model.

10.12 Communication and blocking

10.12.1 The `send` statement

The `send` statement is used to enqueue a message on a Zing channel.

```
send-statement:
  send ( expression , expression ) ;
```

The first expression in the `send` statement identifies the channel to which the message should be delivered. The type of the expression must be a reference to a Zing channel. The second expression is the message to be enqueued on the channel. The type of the second expression must match the element type of the channel referenced in the first expression.

Note that if the channel's element type is complex, then sending a message merely enqueues a Zing pointer and the message is passed by reference. If the channel's element type is simple, then messages are passed by value.

10.12.2 The `select` statement

The `select` statement allows a Zing model to select a path of execution from a number of alternatives (*join statements*) based on the status of their *join conditions*. If necessary, the `select` statement will block until one of its join conditions is satisfied. This is the only way in which a Zing process can become blocked.

```
select-statement:
  select select-qualifiersopt { join-statements }
```

```
select-qualifiers:
  select-qualifier
  select-qualifiers select-qualifier
```

```
select-qualifier:
  end
  first
  visible
```

join-statements:
 join-statement
 join-statements join-statement

join-statement:
 join-list -> *embedded-statement*
 timeout -> *embedded-statement*

join-list:
 join-pattern
 join-list && *join-pattern*

join-pattern:
 wait (*boolean-expression*)
 receive (*expression* , *expression*)
 event (*integer-expression* , *boolean-expression*)

The *select* statement consists of the *select* keyword, followed by one or more optional qualifiers and a set of join statements enclosed in braces. Each join statement consists of a *join-list* followed by the “->” token, followed by an *embedded-statement*. The *join-list* specifies the conditions that must be satisfied before executing the *embedded-statement*.

10.12.2.1 The *first* qualifier

When multiple *join-statements* in a *select* statement are satisfied, Zing will, by default, make a non-deterministic selection from the available alternatives and the model-checker will consider the states resulting from each possible selection. If the *first* qualifier appears on a *select* statement, then only the first satisfied join statement will be considered.

10.12.2.2 The *end* qualifier

By default, Zing models reach a successful conclusion when all of their running processes terminate normally by returning from their entry point method. In some cases, though, it is normal for one or more processes to reach an idle state awaiting input of some kind. The *end* qualifier identifies those *select* statements that can be considered normal end states for a process.

10.12.2.3 The *visible* qualifier

When constructing Zing models for use with the Zing refinement checker, it is sometimes necessary to make visible certain aspects of the model’s execution. The *visible* qualifier guarantees that the selection of any join statement of the *select* statement for execution will result in the generation of an external event by the Zing runtime. If the selected join statement contains one or more event patterns, then external events are generated according to the integer and boolean expressions provided. If a join statement is executed that contains no event patterns, then a “tau” event is generated indicating that an “internal” move was made by the model.

10.12.2.4 Join lists

Each join statement in a *select* begins with a join list. A join list may be either a list of one or more join patterns (separated by the “&&” token), or the *timeout* keyword. The *timeout* keyword may not be combined with the other kinds of join patterns described here.

If a list of join patterns is used, then the join statement is runnable when all of the patterns in the list are satisfied.

10.12.2.5 wait patterns

A wait pattern is used to enable or disable the execution of a join statement based on the value of a given Boolean expression. In the simplest case of a single join statement and join pattern, the select statement simply blocks until the Boolean expression becomes true.

The expression must be free of side-effects and the choose operator must not be used.

Method calls are permitted within wait expressions, however they must:

1. Have no side-effects.
2. Have deterministic behavior.
3. Be non-blocking.
4. Have no output parameters.
5. Have a Boolean return value.

Methods invoked from a wait expression need not be atomic, but they will always be executed within an atomic context, so blocking is not permitted.

10.12.2.6 receive patterns

A receive pattern is enabled when a message is available on the referenced channel. The receive pattern specifies both the channel and a variable to which the received message will be assigned. This assignment happens as a side-effect, when (and if) the receive pattern's join statement is selected for execution.

This is the only way in which a message can be received from a channel.

Note that the join patterns of a join list are tested individually to see if they are runnable. Because a receive pattern has an implied side-effect, it is possible to create join statements that appear to be runnable, but in fact are not. In the example below, the select statement will appear to be runnable if channel "ch" contains a single message, even though the join pattern will attempt to consume two messages. If this statement is executed with only a single message in the channel, a runtime error will be reported.

```
select {
    receive(ch, m1) && receive(ch, m2) -> got_messages = true;
}
```

The following example shows how to safely consume multiple messages from a channel in a single select statement.

```
select {
    wait(sizeof(ch) >= 2) && receive(ch, m1) && receive(ch, m2) ->
    got_messages = true;
}
```

10.12.2.7 event patterns

An event pattern is used to encode the communication of a Zing model with an external process. An event pattern works much like the corresponding event statement. As a join statement pattern, an event pattern is always considered to be runnable. event patterns may only be used in select statements that include the visible qualifier.

10.12.2.8 Timeouts

The timeout join pattern is effectively a shortcut for using "wait(true)" as the join condition for the final join statement in a select. If the first qualifier is not present, the timeout will be considered as a possibility (perhaps the *only* possibility) whenever the select statement is executed. If the first qualifier is

present, then the timeout is effectively treated as a last resort – to be considered only when no other join statements are runnable.

10.13 Monitoring execution

10.13.1 The trace statement

The Zing runtime environment includes a facility for tracking interesting events in the execution of a Zing model. Most of these events are “hard-wired” into the Zing runtime – events related to process creation and termination, sending and receiving messages, and so on. One kind of event, called a “trace” event, is reserved for annotating an execution trace with information that may be application- or domain-specific. Trace events are generated whenever a trace statement is executed.

```
trace-statement:  
trace ( string-literal ) ;  
trace ( string-literal , argument-list ) ;
```

A trace statement consists of the trace keyword and a parenthesized, variable-length argument list. The first argument in the list is required to be a *string-literal*.

By convention, the first argument is treated as a format string for the remaining arguments (see String.Format in the .NET Framework). When the Zing tools display the event, it will be formatted using this assumption. This is not required, however, and the Zing runtime object model makes it a simple matter to retrieve the arguments of a trace statement whether or not they are referenced and formatted by the *string-literal*.

During state-space exploration, trace statements are effectively ignored to minimize their impact on performance. When an error is detected, the offending path can be quickly executed again with trace statements enabled to make their events available for inspection.

Multiple trace statements may be executed within a single atomic block. The resulting Zing state will be annotated with trace events for all of the trace statements encountered, and the events will appear in the order in which they were generated.

10.13.2 The event statement

In some cases, it can be useful to consider application-specific information during state-space exploration. trace statements cannot be used for this purpose because of their relatively high overhead (which is why they are normally disabled during model-checking). The event statement is similar in spirit to the trace statement, but with less flexibility and overhead.

```
event-statement:  
event ( integer-expression , boolean-expression ) ;
```

The event statement has a fixed argument list consisting of an integer-valued expression followed by a Boolean expression.

When an event statement is executed, the resulting Zing state is annotated with the runtime values of the two expressions. Multiple event statements may be executed within a single atomic block.

The Zing refinement checker employs event statements (and event join patterns in select statements) to make visible the interactions of a Zing model with a simulated “environment” process. Used in this way, the integer expression encodes the number of a simulated “external” channel, and the Boolean expression encodes the message direction. For more information on constructing Zing models for use with the refinement checker, refer to the Zing User Guide.

Another potential application would be to encode information such as time, cost, or probability using events and use this information in a custom model-checker to find paths that exceed some time or cost threshold, or to estimate the likelihood that some execution path will be encountered.

The standard Zing model-checker makes no use of the data from event statements, but this information is available through the Zing runtime, and is displayed in the Zing trace viewer.

10.14 State-space control

10.14.1 The assert statement

The `assert` statement is used to encode application-specific safety properties in a Zing model.

assert-statement:

```
assert ( boolean-expression ) ;
assert ( boolean-expression , string-literal ) ;
```

The `assert` statement asserts that a given expression will be true in any state in which the statement is runnable. If a state is encountered in which the `assert` statement is runnable, but its *boolean-expression* is not true, then an error will be reported. The optional *string-literal* may be used to supply additional application-specific context with the assertion.

If an assertion should hold for all states of a model, this can be encoded by placing the assertion (or assertions) in a separate, activated process:

```
class Global Assertions
{
    activate static void CheckAssertions()
    {
        assert ( ... expression1 ... );
        assert ( ... expression2 ... );
        ...
    }
};
```

10.14.2 The assume statement

The `assume` statement is used to remove unwanted execution scenarios from consideration.

assume-statement:

```
assume ( boolean-expression ) ;
```

By “assuming” a condition holds, a Zing model instructs the runtime to silently ignore any execution path in which the condition is found to be false. Unlike the `assert` statement, this is not considered to be an error condition.

The alternating-bit protocol example in chapter 1 shows one example of this. In that case, we used an `assert` statement to avoid any process interleavings in which the producer process got too far ahead of the consumer.

`assume` statements can also be a useful way of applying a filter to a set of related non-deterministic selections:

```
atomic
{
    bool b1 = choose(bool);
    bool b2 = choose(bool);
    assume (b1 != b2);
}
```

During the execution of the atomic block, two non-deterministic selections and a total of four outcomes will have to be considered. When the atomic block completes, only two states will remain –

the ones in which different values were chosen for b1 and b2. The other two states did not satisfy the assume and were effectively pruned from the state graph. And because an atomic block is used, the unwanted intermediate states are never visible to the rest of the Zing model.

10.14.3 Atomic blocks

An atomic block guarantees that the statements within the block will be executed in a single state transition and will not have their execution interleaved with other runnable processes in the Zing model.

atomic-block:

```
atomic { statement-listopt }
```

If the first statement within the atomic block is a select statement, it serves as a guard on the entire block. In this case, the atomic block is not considered runnable until the select statement becomes runnable. Once this happens, the select statement and the remainder of the atomic block are executed in a single atomic step.

If the first statement in the block is not a select statement, then the atomic block is always runnable and will run to completion as soon as its process is selected for execution.

If a select statement appears elsewhere in an atomic block (i.e. not as its first statement), then it must always be the case that at least one join statement in the select is runnable. If this requirement is not satisfied, then a runtime error is reported.

atomic blocks are often required to ensure that a Zing model exhibits the desired operational semantics. A wide variety of synchronization primitives, for example, can be modeled through combinations of select statements and atomic blocks. In addition to achieving correct behavior, the liberal use of atomic blocks is encouraged as a way of reducing the state-space of a model, provided they do not overly constrain its behavior.

11. Expressions

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

11.1 Expression classifications

An expression is classified as one of the following:

A value. Every value has an associated type.

A variable. Every variable has an associated type, namely the declared type of the variable.

A type. An expression with this classification can only appear as the left hand side of a *member-access* or as an operand for the choose operator. In any other context, an expression classified as a type causes a compile-time error.

A method resulting from a member lookup. A method may have an associated instance expression. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by *this*. A method is only permitted in an *invocation-expression*. In any other context, an expression classified as a method causes a compile-time error.

Nothing. This occurs when the expression is an invocation of a method with a return type of `void`. An expression classified as nothing is only valid in the context of a *statement-expression*.

The final result of an expression is never a type or method. Rather, as noted above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

11.1.1 Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a *value*. In such cases, if the actual expression denotes a type, method, or nothing, a compile-time error occurs. However, if the expression denotes a variable, the value of the variable is implicitly substituted: The value of a variable is simply the value currently stored in the storage location identified by the variable.

11.2 Operators

Expressions are constructed from *operands* and *operators*. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

There are two kinds of operators:

Unary operators. The unary operators take one operand and use prefix notation (such as `-x`).

Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).

The order of evaluation of operators in an expression is determined by the *precedence* and *associativity* of the operators (§11.2.1).

Operands in an expression are evaluated from left to right. This is separate from and unrelated to operator precedence.

11.2.1 Operator precedence and associativity

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, the expression $x + y * z$ is evaluated as $x + (y * z)$ because the $*$ operator has higher precedence than the binary $+$ operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an *additive-expression* consists of a sequence of *multiplicative-expressions* separated by $+$ or $-$ operators, thus giving the $+$ and $-$ operators lower precedence than the $*$, $/$, and $\%$ operators.

The following table summarizes all operators in order of precedence from highest to lowest:

Section	Category	Operators
11.5	Primary	x . y $f(x)$ $a[x]$ <code>new</code>
11.5.8	Unary	$+$ $-$ $!$ \sim <code>choose</code> <code>sizeof</code>
0	Multiplicative	$*$ $/$ $\%$
0	Additive	$+$ $-$
11.7	Shift	\ll \gg
11.8	Relational and member testing	$<$ $>$ $<=$ $>=$ <code>in</code>
11.8	Equality	$==$ $!=$
11.9	Logical AND	$\&$
11.9	Logical XOR	\wedge
11.9	Logical OR	\mid
11.10	Conditional AND	$\&\&$
11.10	Conditional OR	$\mid\mid$
11.11	Assignment	$=$

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

Except for assignment, all binary operators are *left-associative*, meaning that operations are performed from left to right. For example, $x + y + z$ is evaluated as $(x + y) + z$.

Assignment is *right-associative*, meaning that operations are performed from right to left. For example, $x = y = z$ is evaluated as $x = (y = z)$.

Precedence and associativity can be controlled using parentheses. For example, $x + y * z$ first multiplies y by z and then adds the result to x , but $(x + y) * z$ first adds x and y and then multiplies the result by z .

11.2.2 Numeric promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators.

11.2.2.1 Unary numeric promotions

Unary numeric promotion occurs for the operands of the predefined $+$, $-$, and \sim unary operators. Unary numeric promotion simply consists of converting operands of type `byte` to type `int`.

11.2.2.2 Binary numeric promotions

Binary numeric promotion occurs for the operands of the predefined +, -, *, /, %, &, |, ^, ==, !=, >, <, >=, and <= binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rule:

If either operand is of type `int`, the other operand is converted to type `int`.

11.3 Member lookup

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup may occur as part of evaluating a *simple-name* or a *member-access* in an expression. Because Zing does not support inheritance or overloading, the member lookup process is quite simple.

A member lookup of a name `N` in a type `T` is processed as follows:

If a member named `N` exists in `T`, then this member is the result of the lookup. If no member named `N` exists, then the lookup produces no match.

11.4 Function members (methods)

Function members are members that contain executable statements. In Zing, the only kind of function member is a method.

The argument list of a method invocation provides actual values or variable references for the parameters of the method.

Once a method has been identified at compile-time, the actual run-time process of invoking the function member is described in §11.4.2.

The following table summarizes the processing that takes place in method invocations. In the table, `e`, `x`, and `y` indicate expressions classified as variables or values, `T` indicates an expression classified as a type, and `F` is the simple name of a method.

Example	Description
<code>F(x, y)</code>	Member lookup is applied to locate method <code>F</code> in the containing class. The method is invoked with the argument list <code>(x, y)</code> . If the method is not <code>static</code> , the instance expression is <code>this</code> .
<code>T.F(x, y)</code>	Member lookup is applied to locate method <code>F</code> in the class <code>T</code> . A compile-time error occurs if the method is not <code>static</code> . The method is invoked with the argument list <code>(x, y)</code> .
<code>e.F(x, y)</code>	Member lookup is applied to locate method <code>F</code> in the class given by the type of <code>e</code> . A compile-time error occurs if the method is <code>static</code> . The method is invoked with the instance expression <code>e</code> and the argument list <code>(x, y)</code> .

11.4.1 Argument lists

Every method invocation includes an argument list which provides actual values or variable references for the parameters of the function member. The arguments are specified as an *argument-list*, as described below.

The arguments of a method invocation are specified as an *argument-list*:

argument-list:
 argument
 argument-list , *argument*

argument:
 expression
 out *variable-reference*

An *argument-list* consists of one or more *arguments*, separated by commas. Each argument can take one of the following forms:

An *expression*, indicating that the argument is passed as a value parameter (§8.4.1.1).

The keyword `out` followed by a *variable-reference* (**Error! Reference source not found.**), indicating that the argument is passed as an output parameter (§8.4.1.2).

During the run-time processing of a method invocation (§11.4.2), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

For a value parameter, the argument expression is evaluated and an implicit conversion to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.

For an output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation.

11.4.2 Method invocation

This section describes the process that takes place at run-time to invoke a particular method. It is assumed that a compile-time process has already determined the particular method to invoke.

For purposes of describing the invocation process, methods are divided into two categories:

Static methods.

Instance methods. Instance methods are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as `this` (§11.5.6).

The run-time processing of a method invocation consists of the following steps, where *M* is the method and, if *M* is an instance method, *E* is the instance expression:

If *M* is a static method:

- The argument list is evaluated as described in §11.4.1.
- *M* is invoked.

If *M* is an instance method:

- *E* is evaluated.
- The argument list is evaluated as described in §11.4.1.
- The value of *E* is checked to be valid. If the value of *E* is `null`, a `NullPointerException` error is reported and execution is halted.
- The function member implementation is invoked. The object referenced by *E* becomes the object referenced by `this`.

11.5 Primary expressions

Primary expressions include the simplest forms of expressions.

primary-expression:
literal
simple-name
parenthesized-expression
member-access
element-access
this-access
object-creation-expression
sizeof-expression
choose-expression

11.5.1 Literals

A *primary-expression* that consists of a *literal* is classified as a value.

11.5.2 Simple names

A *simple-name* consists of a single identifier.

simple-name:
identifier

A *simple-name* is evaluated and classified as follows:

If the *simple-name* appears within a *block* and if the *block* (or an enclosing block) contains a local variable or parameter with the given name, then the *simple-name* refers to that local variable or parameter and is classified as a variable.

Otherwise, for type T of the immediately enclosing class declaration, if a member lookup of the *simple-name* in T produces a match:

- If lookup identifies a method, the result is a method with an associated instance expression of this form.
- If the lookup identifies an instance member, and if the reference occurs within the *block* of an instance method, the result is the same as a member access (§11.5.4) of the form this.E, where E is the *simple-name*.
- Otherwise, the result is the same as a member access (§11.5.4) of the form T.E, where E is the *simple-name*. In this case, it is a compile-time error for the *simple-name* to refer to an instance member.

Otherwise, the name given by the *simple-name* is undefined and a compile-time error occurs.

11.5.3 Parenthesized expressions

A *parenthesized-expression* consists of an *expression* enclosed in parentheses.

parenthesized-expression:
(*expression*)

A *parenthesized-expression* is evaluated by evaluating the *expression* within the parentheses. If the *expression* within the parentheses denotes a type or method, a compile-time error occurs. Otherwise, the result of the *parenthesized-expression* is the result of the evaluation of the contained *expression*.

11.5.4 Member access

A *member-access* consists of a *primary-expression*, followed by a “.” token, followed by an *identifier*.

member-access:
primary-expression . *identifier*

A *member-access* of the form *E*. *I*, where *E* is a *primary-expression* and *I* is an *identifier*, is evaluated and classified as follows:

If *E* is a *primary-expression* classified as a type, and a member lookup (§11.3) of *I* in *E* produces a match, then *E*. *I* is evaluated and classified as follows:

- If *I* identifies a method, then the result is a method with no associated instance expression.
- If *I* identifies a `static` field, the result is a variable, namely the static field *I* in *E*.
- If *I* identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
- Otherwise, *E*. *I* is an invalid member reference, and a compile-time error occurs.

If *E* is a variable, or value, the type of which is *T*, and a member lookup (§11.3) of *I* in *T* produces a match, then *E*. *I* is evaluated and classified as follows:

- If *I* identifies a method, then the result is a method with an associated instance expression of *E*.
- If *T* is a *class-type* and *I* identifies an instance field of that *class-type*:
 - If the value of *E* is `null`, then a `NullPointerException` is reported.
 - Otherwise, the result is a variable, namely the field *I* in the object referenced by *E*.

Otherwise, *E*. *I* is an invalid member reference, and a compile-time error occurs.

11.5.4.1 Identical simple names and type names

In a member access of the form *E*. *I*, if *E* is a single identifier, and if the meaning of *E* as a *simple-name* (§11.5.2) is a field, local variable, or parameter with the same type as the meaning of *E* as a *type-name*, then both possible meanings of *E* are permitted. The two possible meanings of *E*. *I* are never ambiguous, since *I* must necessarily be a member of the type *E* in both cases. In other words, the rule simply permits access to the static members of *E* where a compile-time error would otherwise have occurred.

11.5.5 Element access

An *element-access* consists of a *primary-expression*, followed by a “[“ token, followed by an *expression*, followed by a “]” token.

element-access:

primary-expression [*expression*]

If the *primary-expression* of an *element-access* is a value of an *array-type*, the *element-access* is an array access. If the *primary-expression* is a type or a value of a non-array type, then a compile-time error occurs.

11.5.5.1 Array access

For an array access, the *primary-expression* of the *element-access* must be a value of an *array-type*. The index expression must be of type `int` or `byte`.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value of the index expression.

The run-time processing of an array access of the form *P*[*A*], where *P* is a *primary-expression* of an *array-type* and *A* is an *expression*, consists of the following steps:

P is evaluated.

The index expression is evaluated. Following evaluation of the index expression, an implicit conversion to `int` is performed.

The value of `P` is checked to be valid. If the value of `P` is `null`, a `ZingNullReferenceException` is reported and no further steps are executed.

The value of the index expression is checked against the actual bounds of the array instance referenced by `P`. If the value is out of range, a `ZingIndexOutOfRangeException` error is reported and no further steps are executed.

The location of the array element given by the index expression is computed, and this location becomes the result of the array access.

11.5.6 This access

A *this-access* consists of the reserved word `this`.

this-access:
`this`

A *this-access* is permitted only in the *block* of an instance method. When `this` is used in a *primary-expression* within an instance method, it is classified as a value. The type of the value is the class within which the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.

Use of `this` in a *primary-expression* in a context other than the one listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method or in a *variable-initializer* of a field declaration.

11.5.7 The new operator

The new operator is used to create new instances of complex types (classes, arrays, sets, and channels).

object-creation-expression:
`new type`

The *type* of an *object-creation-expression* must be a class, array, set, or channel type. Otherwise, a compile-time error occurs. The result of the *object-creation-expression* is a value of the given type. The instance is initialized according to the rules given for the type (class, array, set, or channel).

11.5.8 The sizeof operator

The `sizeof` operator is used to obtain the size of some collection types in Zing.

sizeof-expression:
`sizeof (type)`
`sizeof (primary-expression)`

The operand must either denote an array type, or be a value referring to an instance of an array (§5.2.4), set (§6.2.7), or channel (§7.2.4) type. The `int` result of the `sizeof` operator is the number of elements, members, or messages in the array, set, or channel, respectively.

11.5.9 The choose operator

The choose operator is used to make a non-deterministic choice from a set of alternative values.

choose-expression:
`choose (type)`
`choose (primary-expression)`

The operand determines the set of values from which the selection will be made. When the Zing model is executed, the number of available alternatives is accessible through the object model, and subsequent execution can be directed to accept a particular alternative. The model-checker will consider all possible selections.

The value of the choose operator is that of the externally-chosen alternative.

The operand must be one of the following:

- The type name of a range type (§4.8), an enumeration (§4.4), or the predefined `bool` type. In this case, the number of alternatives and their values are known at compile-time.
- A value referring to an array (§0) or set type (§6.2.9). In this case, the number of alternatives and their values are determined at runtime.

Note: The choose operator may only be used as the right operand of a simple assignment statement.

11.6 Unary operators

The `+`, `-`, `!`, and `~` are called the unary operators.

unary-expression:
primary-expression
+ unary-expression
- unary-expression
! unary-expression
~ unary-expression

11.6.1 Unary plus operator

For this operator, the result is simply the value of the operand. The operand must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

11.6.2 Unary minus operator

For an operation of the form `-x`, the result is computed by subtracting `x` from zero. If the value of `x` is the smallest representable value of the operand type (-2^{31} for `int`), then the mathematical negation of `x` is not representable within the operand type. If this occurs, the result is the value of the operand and the overflow is not reported.

The operand must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

11.6.3 Logical negation operator

For an operation of the form `!x`, the operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`. The operand must be of type `bool`, otherwise a compile-time error occurs.

11.6.4 Bitwise complement operator

For an operation of the form `~x`, the result of the operation is the bitwise complement of `x`. The operand must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

Arithmetic operators

The `*`, `/`, `%`, `+`, and `-` operators are called the arithmetic operators.

multiplicative-expression:

unary-expression

multiplicative-expression * *unary-expression*

multiplicative-expression / *unary-expression*

multiplicative-expression % *unary-expression*

additive-expression:

multiplicative-expression

additive-expression + *multiplicative-expression*

additive-expression - *multiplicative-expression*

11.6.5 Multiplication operator

For an operation of the form $x * y$, the result is the product of x and y . The operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

Overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

11.6.6 Division operator

For an operation of the form x / y , the result is the quotient of x and y . The operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

If the value of the right operand is zero, a `Zi ngDi vi deByZeroExcept i on` error is reported.

The division rounds the result towards zero, and the absolute value of the result is the largest possible integer that is less than the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have opposite signs.

If the left operand is the smallest representable `int` value and the right operand is -1 , an overflow occurs. A `Zi ngOverfl owExcept i on` error is always reported in this situation.

11.6.7 Remainder operator

For an operation of the form $x \% y$, the result is the remainder of the division between x and y . The operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

If the value of the right operand is zero, a `Zi ngDi vi deByZeroExcept i on` error is reported.

11.6.8 Addition operator

For an operation of the form $x + y$, the result is the sum of x and y . The operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

Overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

The addition operator may also be applied to set types as described in §6.2.

11.6.9 Subtraction operator

For an operation of the form $x - y$, the result is the subtraction of y from x . The operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

Overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

The subtraction operator may also be applied to set types as described in §6.2.

11.7 Shift operators

The << and >> operators are used to perform bit shifting operations.

```
shift-expression:  
additive-expression  
shift-expression << additive-expression  
shift-expression >> additive-expression
```

For an operation of the form $x \ll \text{count}$ or $x \gg \text{count}$, the operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs. The result is of type `int`.

The predefined shift operators are listed below.

Shift left:

The << operator shifts x left by a number of bits computed as described below.

The high-order bits outside the range of the result type of x are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

Shift right:

The >> operator shifts x right by a number of bits computed as described below.

The low-order bits of x are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if x is non-negative and set to one if x is negative.

The shift count is given by the low-order five bits of `count`. In other words, the shift count is computed from $\text{count} \& 0x1F$. If the resulting shift count is zero, the shift operators simply return the value of x .

Shift operations never cause overflows.

11.8 Relational and membership-testing operators

The `==`, `!=`, `<`, `>`, `<=`, `>=`, `in` and `as` operators are called the relational and membership-testing operators.

```
relational-expression:  
shift-expression  
relational-expression < shift-expression  
relational-expression > shift-expression  
relational-expression <= shift-expression  
relational-expression >= shift-expression  
relational-expression in primary-expression
```

```
equality-expression:  
relational-expression  
equality-expression == relational-expression  
equality-expression != relational-expression
```

The `in` operator is described in §11.8.5.

The `==`, `!=`, `<`, `>`, `<=` and `>=` operators are **comparison operators**. For an operation of the form $x \text{ op } y$, where *op* is a comparison operator, the operands are converted to the supported parameter types of the operator, and the type of the result is `bool`.

Operation	Result
$x == y$	true if x is equal to y, false otherwise
$x != y$	true if x is not equal to y, false otherwise
$x < y$	true if x is less than y, false otherwise
$x > y$	true if x is greater than y, false otherwise
$x <= y$	true if x is less than or equal to y, false otherwise
$x >= y$	true if x is greater than or equal to y, false otherwise
$x \text{ in } y$	true if element x is a member of set y, false otherwise

11.8.1 Integer comparison operators

Each of integer comparison operators compares the numeric values of the two integer operands and returns a bool value that indicates whether the particular relation is true or false.

11.8.2 Boolean equality operators

The result of $x == y$ is true if both x and y are true or if both x and y are false. Otherwise, the result is false.

The result of $x != y$ is false if both x and y are true or if both x and y are false. Otherwise, the result is true.

11.8.3 Enumeration comparison operators

The result of evaluating $x \text{ op } y$, where x and y are expressions of an enumeration type E, and op is one of the comparison operators is of type bool. Comparisons are made based on the lexical ordering of the enumeration members within the enumeration type declaration. A member that appears before another is “less than” the second member.

11.8.4 Reference type equality operators

The equality operators ($==$ and $!=$) return the bool result of comparing the two references for equality or non-equality. They apply to all complex types.

The predefined reference type equality operators require the operands to be *reference-type* values or the value null; furthermore, they require that a standard implicit conversion exists from the type of either operand to the type of the other operand. Unless both of these conditions are true, a compile-time error occurs.

Given these rules, it is a compile-time error to use the predefined reference type equality operators to compare two references that are known to be different at compile-time. For example, if the compile-time types of the operands are two class types A and B, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a compile-time error.

11.8.5 The in operator

The in operator (§6.2.6) tests for the existence of an element in a set. An expression of the form $x \text{ in } y$ returns true if x is a member of the set y, false otherwise.

The right operand must be an instance of a set type and the type of the left operand must be the same as the set’s element type. For sets containing complex types, equality is tested according to the equality rules for reference

types (§11.8.4). For simple types, the values are compared according to the applicable rules for the type (recursively applied to members for struct types).

11.9 Logical operators

The `&`, `^`, and `|` operators are called the logical operators.

and-expression:

equality-expression

and-expression & *equality-expression*

exclusive-or-expression:

and-expression

exclusive-or-expression ^ *and-expression*

inclusive-or-expression:

exclusive-or-expression

inclusive-or-expression | *exclusive-or-expression*

The `&` operator computes the bitwise logical AND of the two operands, the `|` operator computes the bitwise logical OR of the two operands, and the `^` operator computes the bitwise logical exclusive OR of the two operands. No overflows are possible from these operations. The operands must be of type `int` or implicitly convertible to `int`, otherwise a compile-time error occurs.

11.10 Conditional logical operators

The `&&` and `||` operators are called the conditional logical operators. They are also called the “short-circuiting” logical operators.

conditional-and-expression:

inclusive-or-expression

conditional-and-expression && *inclusive-or-expression*

conditional-or-expression:

conditional-and-expression

conditional-or-expression || *conditional-and-expression*

The operands of the `&&` and `||` operators must be of type `bool`.

For the operation `x && y`, `x` is first evaluated. Then, if `x` is `true`, `y` is evaluated, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.

For the operation `x || y`, `x` is first evaluated. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and this becomes the result of the operation.

11.11 Invocation expressions

An *invocation-expression* is used to invoke a method.

invocation-expression:

primary-expression (*argument-list_{opt}*)

The *primary-expression* of an *invocation-expression* must be a method. If the *primary-expression* is not a method, a compile-time error occurs.

The optional *argument-list* provides values or variable references for the parameters of the method.

The result of evaluating an *invocation-expression* is classified as follows:

ZING LANGUAGE SPECIFICATION

If the *invocation-expression* invokes a method that returns void, the result is nothing. An expression that is classified as nothing cannot be an operand of any operator, and is permitted only in the context of a *statement-expression* (§10.6).

Otherwise, the result is a value of the type returned by the method.

The compile-time processing of a method invocation of the form $M(A)$, where M is a method and A is an optional *argument-list*, consists of the following steps:

The candidate method for the method invocation is determined by member lookup.

If the member lookup returns no result, then no applicable method exists, and a compile-time error occurs.

Given a candidate method, the invocation of the method is validated: If the candidate method is a static method, the method must have resulted from a *simple-name* or a *member-access* through a type. If the candidate method is an instance method, the method must have resulted from a *simple-name*, or a *member-access* through a variable. If neither of these requirements is true, a compile-time error occurs.

Once a method has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in §11.4.2.

11.12 Assignment

The assignment operator assigns a new value to a variable.

assignment:

unary-expression = *expression*

unary-expression = *invocation-expression*

The left operand of an assignment must be an expression classified as a variable.

In an assignment, the right operand must be an expression of a type that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable given by the left operand.

The assignment operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form $a = b = c$ is evaluated as $a = (b = c)$.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

The run-time processing of an assignment of the form $x = y$ consists of the following steps:

- x is evaluated to produce the variable.
- y is evaluated and, if required, converted to the type of x through an implicit conversion.
- The value resulting from the evaluation and conversion of y is stored into the location given by the evaluation of x .

11.13 Expression

An *expression* is either a *conditional-expression* or an *assignment*.

expression:

conditional-expression

assignment

11.14 Constant expressions

A *constant-expression* is an expression that can be fully evaluated at compile-time.

constant-expression:
expression

The type of a constant expression can be one of the following: `byte`, `int`, `bool`, any enumeration type, or the null type. The following constructs are permitted in constant expressions:

Literals (including the `null` literal).

References to members of enumeration types.

Parenthesized sub-expressions, which are themselves constant expressions.

The predefined `+`, `-`, `!`, and `~` unary operators.

The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.

Whenever an expression is of one of the types listed above and contains only the constructs listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have caused a runtime error, compile-time evaluation causes a compile-time error to occur.

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Array size specifications
- Range type declarations

An implicit constant expression conversion permits a constant expression of type `int` to be converted to `byte`, provided the value of the constant expression is within the range of the `byte` type.

11.15 Boolean expressions

A *boolean-expression* is an expression that yields a result of type `bool`.

boolean-expression:
expression

The controlling conditional expression of an *if-statement* (§10.7) or *while-statement* (§10.8.1) is a *boolean-expression*.

A *boolean-expression* is required to be of type `bool`. Otherwise, a compile-time error occurs.

A. Grammar

This appendix contains summaries of the lexical and syntactic grammars found in the main document. Grammar productions appear here in the same order that they appear in the main document.

A.1 Lexical grammar

input:
*input-section*_{opt}

input-section:
input-section-part
input-section input-section-part

input-section-part:
*input-elements*_{opt} *new-line*
pp-directive

input-elements:
input-element
input-elements input-element

input-element:
whitespace
comment
token

A.1.1 Line terminators

new-line:
 Carriage return character (U+000D)
 Line feed character (U+000A)
 Carriage return character (U+000D) followed by line feed character (U+000A)
 Line separator character (U+2028)
 Paragraph separator character (U+2029)

A.1.2 White space

whitespace:
 Any character with Unicode class Zs
 Horizontal tab character (U+0009)
 Vertical tab character (U+000B)
 Form feed character (U+000C)

A.1.3 Comments

comment:
single-line-comment
delimited-comment

single-line-comment:

// input-characters_{opt}

input-characters:

input-character

input-characters input-character

input-character:

Any Unicode character except a *new-line-character*

new-line-character:

Carriage return character (U+000D)

Line feed character (U+000A)

Line separator character (U+2028)

Paragraph separator character (U+2029)

delimited-comment:

/ delimited-comment-characters_{opt} */*

delimited-comment-characters:

delimited-comment-character

delimited-comment-characters delimited-comment-character

delimited-comment-character:

not-asterisk

** not-slash*

not-asterisk:

Any Unicode character except ***

not-slash:

Any Unicode character except */*

A.1.4 Tokens

token:

identifier

keyword

integer-literal

real-literal

string-literal

operator-or-punctuator

A.1.5 Unicode character escape sequences

unicode-escape-sequence:

\u hex-digit hex-digit hex-digit hex-digit

\U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit

A.1.6 Identifiers

identifier:

available-identifier

@ identifier-or-keyword

available-identifier:

An *identifier-or-keyword* that is not a *keyword*

identifier-or-keyword:

identifier-start-character identifier-part-characters_{opt}

identifier-start-character:

letter-character
 _ (the underscore character)

identifier-part-characters:

identifier-part-character
identifier-part-characters identifier-part-character

identifier-part-character:

letter-character
decimal-digit-character
connecting-character
combining-character
formatting-character

letter-character:

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
 A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

combining-character:

A Unicode character of classes Mn or Mc
 A *unicode-escape-sequence* representing a character of classes Mn or Mc

decimal-digit-character:

A Unicode character of the class Nd
 A *unicode-escape-sequence* representing a character of the class Nd

connecting-character:

A Unicode character of the class Pc
 A *unicode-escape-sequence* representing a character of the class Pc

formatting-character:

A Unicode character of the class Cf
 A *unicode-escape-sequence* representing a character of the class Cf

A.1.7 Keywords

keyword: one of

activate	array	assert	assume	async
atomic	bool	byte	chan	choose
class	decimal	double	else	enum
end	event	false	first	float
foreach	goto	if	in	int
long	new	null	object	out
range	raise	receive	return	sbyte
select	send	set	short	sizeof
static	struct	symbolic	this	timeout
trace	true	try	uint	ulong
ushort	visible	void	wait	while
with				

A.1.8 Literals

literal:

boolean-literal
integer-literal
real-literal
string-literal
null-literal

boolean-literal:

true
false

integer-literal:

decimal-integer-literal
hexadecimal-integer-literal

decimal-integer-literal:

decimal-digits *integer-type-suffix*_{opt}

decimal-digits:

decimal-digit
decimal-digits *decimal-digit*

decimal-digit: one of

0 1 2 3 4 5 6 7 8 9

integer-type-suffix: one of

U u L l UL Ul uL ul LU Lu lU lu

hexadecimal-integer-literal:

0x *hex-digits* *integer-type-suffix*_{opt}
0X *hex-digits* *integer-type-suffix*_{opt}

hex-digits:

hex-digit
hex-digits *hex-digit*

hex-digit: one of

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

real-literal:

decimal-digits . *decimal-digits* *exponent-part*_{opt} *real-type-suffix*_{opt}
. *decimal-digits* *exponent-part*_{opt} *real-type-suffix*_{opt}
decimal-digits *exponent-part* *real-type-suffix*_{opt}
decimal-digits *real-type-suffix*

exponent-part:

e *sign*_{opt} *decimal-digits*
E *sign*_{opt} *decimal-digits*

sign: one of

+ -

real-type-suffix: one of

F f D d M m

string-literal:

regular-string-literal
verbatim-string-literal

regular-string-literal:

" *regular-string-literal-characters*_{opt} "

regular-string-literal-characters:

regular-string-literal-character
regular-string-literal-characters *regular-string-literal-character*

regular-string-literal-character:

single-regular-string-literal-character
simple-escape-sequence
hexadecimal-escape-sequence
unicode-escape-sequence

single-regular-string-literal-character:

Any character except " (U+0022), \ (U+005C), and *new-line-character*

verbatim-string-literal:

@" *verbatim-string-literal-characters*_{opt} "

verbatim-string-literal-characters:

verbatim-string-literal-character
verbatim-string-literal-characters *verbatim-string-literal-character*

verbatim-string-literal-character:

single-verbatim-string-literal-character
quote-escape-sequence

single-verbatim-string-literal-character:

any character except "

quote-escape-sequence:

""

null-literal:

nul l

A.1.9 Operators and punctuators

operator-or-punctuator: one of

{	}	[]	()	.	,	:	;
+	-	*	/	%	&		^	!	=
<	>	..	&&		<<	>>	==	!=	->
<=	>=								

A.1.10 Pre-processing directives

pp-directive:

pp-declaration
pp-conditional
pp-line
pp-diagnostic
pp-region

pp-new-line:
whitespace_{opt} single-line-comment_{opt} new-line

conditional-symbol:
 Any *identifier-or-keyword* except `true` or `false`

pp-expression:
whitespace_{opt} pp-or-expression whitespace_{opt}

pp-or-expression:
pp-and-expression
pp-or-expression whitespace_{opt} || whitespace_{opt} pp-and-expression

pp-and-expression:
pp-equality-expression
pp-and-expression whitespace_{opt} && whitespace_{opt} pp-equality-expression

pp-equality-expression:
pp-unary-expression
pp-equality-expression whitespace_{opt} == whitespace_{opt} pp-unary-expression
pp-equality-expression whitespace_{opt} != whitespace_{opt} pp-unary-expression

pp-unary-expression:
pp-primary-expression
! whitespace_{opt} pp-unary-expression

pp-primary-expression:
`true`
`false`
conditional-symbol
`(pp-expression)`

pp-declaration:
whitespace_{opt} # whitespace_{opt} define whitespace conditional-symbol pp-new-line
whitespace_{opt} # whitespace_{opt} undef whitespace conditional-symbol pp-new-line

pp-conditional:
pp-if-section pp-elif-sections_{opt} pp-else-section_{opt} pp-endif

pp-if-section:
whitespace_{opt} # whitespace_{opt} if whitespace pp-expression pp-new-line conditional-section_{opt}

pp-elif-sections:
pp-elif-section
pp-elif-sections pp-elif-section

pp-elif-section:
whitespace_{opt} # whitespace_{opt} elif whitespace pp-expression pp-new-line conditional-section_{opt}

pp-else-section:
whitespace_{opt} # whitespace_{opt} else pp-new-line conditional-section_{opt}

pp-endif-line:
whitespace_{opt} # whitespace_{opt} endif pp-new-line

conditional-section:
 input-section
 skipped-section

skipped-section:
 skipped-section-part
 skipped-section *skipped-section-part*

skipped-section-part:
 *skipped-characters*_{opt} *new-line*
 pp-directive

skipped-characters:
 *whitespace*_{opt} *not-number-sign* *input-characters*_{opt}

not-number-sign:
 Any *input-character* except #

pp-line:
 *whitespace*_{opt} # *whitespace*_{opt} *line* *whitespace* *line-indicator* *pp-new-line*

line-indicator:
 integer-literal *whitespace*_{opt} *file-name*_{opt}
 default

file-name:
 " *file-name-characters* "

file-name-characters:
 file-name-character
 file-name-characters *file-name-character*

file-name-character:
 Any *input-character* except "

pp-diagnostic:
 *whitespace*_{opt} # *whitespace*_{opt} *error* *pp-message*
 *whitespace*_{opt} # *whitespace*_{opt} *warning* *pp-message*

pp-message:
 new-line
 whitespace *input-characters*_{opt} *new-line*

pp-region:
 pp-start-region *conditional-section*_{opt} *pp-end-region*

pp-start-region:
 *whitespace*_{opt} # *whitespace*_{opt} *region* *pp-message*

pp-end-region:
 *whitespace*_{opt} # *whitespace*_{opt} *endregion* *pp-message*

A.2 Syntactic grammar

A.2.1 Basic concepts

type-name:
 identifier

A.2.2 Types

type:

- value-type*
- reference-type*
- symbolic-type*

value-type:

- struct-type*
- enum-type*
- range-type*

struct-type:

- type-name*
- simple-type*

simple-type:

- numeric-type*
- bool*

numeric-type:

- integral-type*
- floating-point-type*
- decimal*

integral-type:

- sbyte*
- byte*
- short*
- ushort*
- int*
- uint*
- long*
- ulong*

floating-point-type:

- float*
- double*

enum-type:

- type-name*

range-type:

- type-name*

reference-type:

- class-type*
- array-type*
- set-type*
- channel-type*

symbolic-type:

- symbolic bool*
- symbolic int*
- symbolic enum-type*

class-type:
 type-name
 obj ect

array-type:
 type-name

set-type:
 type-name

channel-type:
 type-name

A.2.3 Variables

variable-reference:
 expression

A.2.4 Expressions

argument-list:
 argument
 argument-list , *argument*

argument:
 expression
 out *variable-reference*

primary-expression:
 literal
 simple-name
 parenthesized-expression
 member-access
 element-access
 this-access
 object-creation-expression
 sizeof-expression
 choose-expression

simple-name:
 identifier

parenthesized-expression:
 (*expression*)

member-access:
 primary-expression . *identifier*

predefined-type: one of
 bool byte deci mal doubl e fl oat i nt l ong
 obj ect sbyte short ui nt ul ong ushort

element-access:
 primary-expression [*integer-expression*]

this-access:
 thi s

object-creation-expression:
*new type size-specifier*_{opt}

size-specifier:
 [*integer-expression*]

sizeof-expression:
sizeof (type)
sizeof (primary-expression)

choose-expression:
choose (type)
choose (primary-expression)

unary-expression:
primary-expression
 + *unary-expression*
 - *unary-expression*
 ! *unary-expression*

multiplicative-expression:
unary-expression
multiplicative-expression * *unary-expression*
multiplicative-expression / *unary-expression*
multiplicative-expression % *unary-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*
relational-expression in *primary-expression*

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

and-expression:
equality-expression
and-expression & *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression ^ *and-expression*

ZING LANGUAGE SPECIFICATION

inclusive-or-expression:

exclusive-or-expression
inclusive-or-expression | *exclusive-or-expression*

conditional-and-expression:

inclusive-or-expression
conditional-and-expression && *inclusive-or-expression*

conditional-or-expression:

conditional-and-expression
conditional-or-expression || *conditional-and-expression*

invocation-expression:

primary-expression (*argument-list_{opt}*)

assignment:

unary-expression = *expression*
unary-expression = *invocation-expression*

expression:

conditional-or-expression
assignment

constant-expression:

expression

boolean-expression:

expression

integer-expression:

expression

A.2.5 Statements

statement:

labeled-statement
declaration-statement
embedded-statement

embedded-statement:

attributed-statement
block
atomic-block
empty-statement
expression-statement
if-statement
iteration-statement
jump-statement
try-statement
assert-statement
assume-statement
async-call-statement
send-statement
select-statement
trace-statement
event-statement

block:
 { *statement-list*_{opt} }

atomic-block:
 atomic { *statement-list*_{opt} }

statement-list:
statement
statement-list statement

empty-statement:
 ;

labeled-statement:
identifier : *statement*

attributed-statement:
attributes statement

attributes:
attribute-sections

attribute-sections:
attribute-section
attribute-sections attribute-section

attribute-section:
 [*attribute-list*]
 [*attribute-list* ,]

attribute-list:
attribute
attribute-list , *attribute*

attribute:
*attribute-name attribute-arguments*_{opt}

attribute-name:
identifier

attribute-arguments:
 (*attribute-argument-list*)

attribute-argument-list:
attribute-argument
attribute-argument-list , *attribute-argument*

attribute-argument:
expression

declaration-statement:
local-variable-declaration ;

local-variable-declaration:
type variable-declarator

variable-declarator:
identifier
identifier = expression

expression-statement:
statement-expression ;

statement-expression:
invocation-expression
assignment

if-statement:
if (*boolean-expression*) *embedded-statement*
if (*boolean-expression*) *embedded-statement* *el se* *embedded-statement*

iteration-statement:
while-statement
foreach-statement

while-statement:
while (*boolean-expression*) *embedded-statement*

foreach-statement:
foreach (*type identifier in expression*) *embedded-statement*

jump-statement:
goto-statement
return-statement
raise-statement

goto-statement:
goto *identifier* ;

return-statement:
return *expression*_{opt} ;

raise-statement:
raise *identifier* ;

try-statement:
try *block with* { *with-clauses* }

with-clauses:
with-clause
with-clauses with-clause

with-clause:
identifier -> *embedded-statement*
*** -> *embedded-statement*

assert-statement:
assert (*boolean-expression*) ;
assert (*boolean-expression* , *string-literal*) ;

assume-statement:
assume (*boolean-expression*) ;

trace-statement:
trace (*string-literal*) ;
trace (*string-literal* , *argument-list*) ;

event-statement:
event (*integer-expression* , *boolean-expression*) ;

async-call-statement:
 async invocation-expression ;

send-statement:
 send (expression , expression) ;

select-statement:
 select select-qualifiers_{opt} { join-statements }

select-qualifiers:
 select-qualifier
 select-qualifiers select-qualifier

select-qualifier:
 end
 first
 visible

join-statements:
 join-statement
 join-statements join-statement

join-statement:
 join-list -> embedded-statement
 timeout -> embedded-statement

join-list:
 join-pattern
 join-list && join-pattern

join-pattern:
 wait (boolean-expression)
 receive (expression , expression)
 event (integer-expression , boolean-expression)

A.2.6 Compilation Unit

compilation-unit:
 type-declarations

type-declarations:
 type-declaration
 type-declarations type-declaration

type-declaration:
 class-declaration
 struct-declaration
 array-declaration
 enum-declaration
 range-declaration
 set-declaration
 channel-declaration

A.2.7 Classes

class-declaration:
 class identifier class-body ;

class-body:
 { *class-member-declarations*_{opt} }

class-member-declarations:
class-member-declaration
class-member-declarations class-member-declaration

class-member-declaration:
field-declaration
method-declaration

field-declaration:
*field-modifiers*_{opt} *type* *variable-declarator* ;

field-modifiers:
field-modifier
field-modifiers field-modifier

field-modifier:
 static

variable-declarator:
identifier
identifier = variable-initializer

variable-initializer:
expression

method-declaration:
method-header method-body

method-header:
*method-modifiers*_{opt} *return-type* *identifier* (*formal-parameter-list*_{opt})

method-modifiers:
method-modifier
method-modifiers method-modifier

method-modifier:
 activate
 atomic
 static

return-type:
type
 void

method-body:
 block

formal-parameter-list:
parameters

parameters:
parameter
parameters , *parameter*

parameter:
*parameter-modifier*_{opt} *type* *identifier*

parameter-modifier:
out

A.2.8 Structs

struct-declaration:
struct *identifier* *struct-body* ;

struct-body:
{ *struct-member-declarations*_{opt} }

struct-member-declarations:
struct-member-declaration
struct-member-declarations *struct-member-declaration*

struct-member-declaration:
field-declaration

A.2.9 Arrays

array-declaration:
array *identifier* [*constant-expression*] *type* ;
array *identifier* [] *type* ;

A.2.10 Enums

enum-declaration:
enum *identifier* *enum-body* ;

enum-body:
{ *enum-member-declarations* }
{ *enum-member-declarations* , }
enum-member-declarations:
enum-member-declaration
enum-member-declarations , *enum-member-declaration*

enum-member-declaration:
identifier

A.2.11 Ranges

range-declaration:
range *identifier* *constant-expression* .. *constant-expression* ;

A.2.12 Sets

set-declaration:
set *identifier* *type* ;

A.2.13 Channels

channel-declaration:
chan *identifier* *type* ;

B. Runtime Errors

In a Zing model, it is not possible to catch or handle runtime errors such as the use of a null object reference or division by zero. These errors, and several others, are reported as runtime errors by the Zing runtime. When an error is encountered, the resulting Zing state is “erroneous”, and its `Exception` property contains the .Net exception corresponding to the runtime error. Several of these exception classes contain additional information about the error.

For reference, the complete list of Zing runtime errors is provided in this appendix. For each error, the .Net exception name and a description of the error are given.

.Net Exception Name	Runtime error description
<code>ZingAssertionFailureException</code>	Reported when a Zing <code>assert</code> statement is executed and the value of its expression is <code>false</code> .
<code>ZingAssumeFailureException</code>	This exception is used to propagate assume failures through the runtime but it is not actually reported as an error. It yields a terminal state, but not an erroneous one.
<code>ZingDivideByZeroException</code>	Reported when division by zero occurs in the Zing model.
<code>ZingOverflowException</code>	Reports an arithmetic overflow which can occur as described in §11.6.6.
<code>ZingIndexOutOfRangeException</code>	Reported when the index expression in an array reference is outside the bounds of the array.
<code>ZingInvalidEndStateException</code>	Reported when there are no runnable processes in the Zing state and one or more processes are blocked in a <code>select</code> statement that is <u>not</u> marked with the end qualifier.
<code>ZingInvalidBlockingSelectException</code>	This error occurs if a <code>select</code> statement that is within an <code>atomic</code> block, but is not the first statement in the block, is not runnable. It is an error for <code>select</code> to block in the middle of an atomic execution.
<code>ZingNullReferenceException</code>	This error occurs if a Zing pointer is dereferenced and its value is null.
<code>ZingInvalidChooseException</code>	This error is reported if a <code>choose</code> operator finds that no alternatives are available. Currently, this can only happen if <code>choose</code> is applied to a set object which happens to be empty.
<code>ZingUnexpectedFailureException</code>	This error is a catch-all for any unexpected exceptions that are caught by the Zing runtime. This usually indicates a bug in the Zing runtime or compiler.
<code>ZingUnhandledExceptionException</code>	This error is reported if a Zing exception is thrown and no handler can be found in the current scope or in any of the calling methods on the stack.

C. Example Source Code

C.1 Dining Philosophers

```
class Fork {
    Philosopher holder;

    void Pickup(Philosopher eater) {
        atomic {
            select {
                wait(holder == null) -> holder = eater;
            }
        }
    }

    void PutDown() {
        holder = null;
    }
};

class Philosopher {
    Fork leftFork;
    Fork rightFork;

    void Run() {
        while (true) {
            // pick up forks
            leftFork.Pickup(this);
            rightFork.Pickup(this);
            // eat for a while
            leftFork.PutDown();
            rightFork.PutDown();
            // think for a while
        }
    }
};
```

ZING LANGUAGE SPECIFICATION

```
array Philosophers[5] Philosopher;
array Forks[5] Fork;

class Init {
    activate static void Run() {
        Philosophers p;
        Forks f;
        int i;

        atomic {
            // Allocate the arrays of forks and philosophers
            p = new Philosophers;
            f = new Forks;

            // Allocate the individual fork and philosopher objects
            i = 0;
            while (i < sizeof(Philosophers)) {
                p[i] = new Philosopher;
                f[i] = new Fork;
                i = i + 1;
            }

            // Associate the philosophers with their forks and let them begin
            i = 0;
            while (i < sizeof(Philosophers)) {
                p[i].leftFork = f[i];
                p[i].rightFork = f[(i+1) % sizeof(Philosophers)];

                async p[i].Run();
                i = i + 1;
            }
        }
    }
};
```

C.2 Alternating-bit protocol

```
class Msg {
    bool body;
    bool bit;
};

class Ack {
    bool bit;
};

chan MsgChan Msg;
chan AckChan Ack;

chan BoolChan bool;

class Sender {
    static MsgChan xmit;
    static AckChan recv;

    static void TransmitMsg(bool body, bool bit)
    {
        Msg m;

        select {
            wait(true) -> {
                assume(sizeof(xmit) < Main.QueueSize);
                m = new Msg;
                m.body = body;
                m.bit = bit;
                send(xmit, m);
            }
            wait(true) -> /* lost message */ ;
        }
    }

    static void Run()
    {
        bool currentBit = false;
        Ack a;
```

ZING LANGUAGE SPECIFICATION

```
bool body;
bool gotAck;

while (true) {
    atomic {
        body = choose(bool);
        send(Main.reliableChan, body);

        TransmitMsg(body, currentBit);

        gotAck = false;
    }

    while (!gotAck) {
        atomic {
            select first {
                receive(recv, a) -> gotAck = (a.bit == currentBit);
                timeout -> TransmitMsg(body, currentBit);
            }
        }
    }

    currentBit = !currentBit;
}
};
```

```
class Receiver {
    static MsgChan recv;
    static AckChan xmit;

    static void TransmitAck(bool bit)
    {
        Ack a;

        select {
            wait(true) -> {
                a = new Ack;
                a.bit = bit;
                send(xmit, a);
            }
        }
    }
};
```

```

        }
        wait(true) -> /* lost ack */ ;
    }
}

static void Run()
{
    bool expectedBit = false;
    bool trueBody;
    Msg m;

    // Loop forever consuming messages
    while (true) {
        select { receive(recv, m) -> ; }

        atomic {
            // Always send an ack with the same bit
            TransmitAck(m.bit);

            if (expectedBit == m.bit) {
                // Consume the message here and verify it's body matches
                // what we received through the reliable channel
                select { receive(Main.reliableChan, trueBody) -> ; }
                assert(trueBody == m.body);

                expectedBit = !expectedBit;
            }
        }
    }
}
};

```

```

class Main {

    static int QueueSize = 2;

    static BoolChan reliableChan;

    activate static void Run()

```

ZING LANGUAGE SPECIFICATION

```
{
  atomic {
    reliableChan = new BoolChan;

    Sender.xmit = Receiver.recv = new MsgChan;
    Sender.recv = Receiver.xmit = new AckChan;

    async Sender.Run();
    async Receiver.Run();
  }
};
```