The Faculty of Engineering, Science and Medicine
Aalborg University
Department of Computer Science

Title: Cava – A New Concurrency Model for Java

Research Unit: Database and Programming Technologies

Project Period: Dat6, 1 February, 2007 – 6 June, 2007

Project Group: d617a

Participants:
Birthe Damborg
Anders Mørk Hansen

Supervisor:
Bent Thomsen

Number of Copies: 6

Number of Pages: 118

Synopsis:

This master thesis presents Cava, a new concurrency model for Java. Developing Cava is motivated by the increasing need for concurrent programs, and the fact that Java's present concurrency model has various weaknesses.
Concurrency in Cava is achieved using threads which can have either deterministic or non-deterministic behaviour. All variables are thread local unless they are marked as shared. Access to shared variables is induced with STM semantics as it must occur within transactions. Cava also features a gate construct which enables sending messages between threads.
An experimental implementation is developed by modifying Sun's Java compiler into a Cava compiler and implementing a Cava Runtime System. Moreover, Cava is applied to model various concurrency constructs and concurrent problems.
In order to evaluate how Cava meets its design criteria, and how it is compared to Java, a method for assessing concurrency models is developed and applied. Cava achieves a better score than Java, primarily caused by a better integration with the object-oriented paradigm. However, both are far from the highest possible score and hence, potential future work is presented.

Preface

This report is the result of a Dat6–project in the Department of Computer Science at Aalborg University. The project was carried out during the spring of 2007 within the Database and Programming Technologies research unit.

The report serves as a master thesis in computer science for the participants and represents a continuation of the Dat5–project which was documented in *A Study in Concurrency* [DH07].

Appendix A on page 114 contains definitions of basic concepts of concurrency. These are used from the outset of the main report.

The project includes an experimental implementation of the Cava concurrency model which is enclosed on a CD-ROM. Instructions on how to compile and execute Cava programs are also found on the CD-ROM. Finally, it contains the Cava programs which are described in Chapter 6 on page 50 along with various other examples.

Citations in the report appear like [DH07, p. 15] and refer to the bibliography on p. 112.

Aalborg University, 6 June, 2007.

Birthe Damborg

Anders Mørk Hansen

Contents

Chapter 1

Introduction

This chapter describes the motivation of the work behind this report in Section 1.1. The scope of the project is addressed in Section 1.3 on the next page. Finally, the project goals are stated in Section 1.4 on page 4.

1.1 Motivation

For several decades, computer hardware has evolved to become faster and smaller. One of the latest evolutions in computer hardware is the introduction of multi-core CPUs.

The multi-core technology is designed to enable program threads to execute in parallel. This technology makes a single CPU just as powerful as multiple CPUs have been collectively in the past. However, while it collects the power of multiple processors in a single chip, it does not increase the execution speed of non-concurrent programs. Hence, programs must be designed for concurrent execution to fully utilise the potential computational power in the new multi-core CPUs. According to [HPC07], this requires a change in how software is developed since ways must be found to "make it easy to write programs that run efficiently on manycore systems".

In order to design concurrent programs, software developers must be provided with tools which enable them to create such software. One such tool is programming languages which support concurrency and a large number of languages already exist which have this feature. The languages apply different concepts and strategies to enable the programmer to work with concurrency. Since the introduction of multi-core CPUs requires that programs

become more concurrent, concurrency support is an increasingly important part of programming language design. A focus on the quality of concurrency support should therefore be prioritised.

The various approaches to concurrency have strengths and weaknesses. This makes it unlikely that a simple and ideal concurrency model which fits every purpose and application should exist [DH07]. However, trying to improve the concurrency support in an existing language is still an interesting and important challenge.

1.2 Object-orientation and Java

The object-oriented paradigm has become dominant in relation to software development. Hence, this project only considers concurrency within the object-oriented paradigm. Furthermore, the scope is limited to Java which has contributed significantly to the success of the object-oriented paradigm. Hence, the aim of this project is to develop a new concurrency model for Java. The new concurrency model is termed Cava.

There are several reasons why concurrency in Java is targeted. It is a mainstream language with a focus on concurrency support, it is a well-tested and stable language, and it is widely used. All of these aspects imply that it comprises a solid foundation on which to build Cava. Chapter 2 on page 5 contains details about Java's present concurrency model and some of the problems which it induces. At this point, it suffices to say that the concurrency support in Java could be characterised as low-level and error-prone. Hence, a challenge lies in improving the concurrency support in Java.

1.3 Scope

This section presents the scope of the project. That is, the areas which the work of the project covers.

1.3.1 A New Concurrency Model for Java

Fundamentally, there are two motivations for applying concurrency to a program: to model a naturally concurrent problem or to gain an execution speed-up. This is described further below and is based on [DH07, Section 2.4].

Some problems are concurrent by nature. These consist of independent parts which interact in some defined manner and collectively make up a larger system. Applying concurrency as a natural abstraction for concurrent problems is simply allowing a programmer to easily express concurrent problems in a programming language. If the programming language supports this modelling, a concurrent problem becomes intuitive since the problem does not have to be fitted into a sequential programming model.

Optimising a sequential program by making it concurrent makes it possible to utilise the power of multi-core CPUs and systems with multiple CPUs. In order to make a sequential program concurrent, the programmer must identify the parts of the program which can be run concurrently, split the sequential code into these concurrent parts, and ensure that the changes do not affect the semantics of the program. When a programmer is applying optimisation through concurrency, the sequential task is actually being transformed into a set of concurrent tasks. That is, the sequential task is being fitted into a concurrent programming model. As a consequence, optimisation of sequential programs can be viewed simply as a special case of applying concurrency as a natural abstraction.

Cava should support the programmer in modelling both types of problems. The support should be present both with regard to language abstractions and modelling abstractions. That is, it should be straightforward for the programmer to reason about concurrent problems using the concurrency model. This is also in compliance with [HPC07] which states that in order to utilise the computational potential of multi-core CPUs, the parallel programming model must be "human-centric, not machine-centric".

Furthermore, Cava should not suffer from the weaknesses currently found in Java and described in Section 2.2 on page 8. The goal is to obtain a concurrency model which is modern, high-level and developer friendly both with regard to language abstractions and modelling abstractions. Chapter 3 on page 11 presents the design criteria of Cava in more detail. Note that the development of Cava is done without consideration to breaking existing Java code.

1.3.2 Implementation

Developing a new concurrency model will only remain a theoretical exercise if the new language constructs cannot be tested and evaluated by applying them to concurrent problems. Hence, the scope of the project also includes an implementation of Cava. This allows for experimenting with and testing the

model by actually writing and executing concurrent software which utilises the new language constructs.

1.3.3 Assessment of Concurrency Models

Since a new concurrency model for Java is developed, it appears natural to compare it to Java's present concurrency model. In general, it is useful to be able to assess the strengths and weaknesses of a given existing concurrency model in a programming language. Hence, the scope of the project also includes the development of a method for assessing and comparing concurrency models.

1.4 Goals

The previous section presented the areas which the project targets. To summarise, the goals of the project consist of the following:

- Developing a new concurrency model for the Java language which remedies some of the weaknesses of the present concurrency model.
- Making an implementation of the new concurrency model.
- Applying the concurrency model to various examples of classical concurrency constructs and concurrent problems to investigate its applicability.
- Developing a method for assessing and comparing concurrency models.
- Applying the assessment method to Java and Cava, and comparing the results.

Chapter 2

Concurrency in Java

When developing a new concurrency model for Java, comprehensive insight into how concurrency is presently achieved is valuable since it may reveal areas which are particularly interesting to target in the new model. This chapter contains a description of Java's concurrency model in Section 2.1 and a discussion of its strengths and weaknesses is included in Section 2.2 on page 8. The contents of the chapter are a markedly abridged version of [DH07, Chapter 5].

2.1 Concurrency Model

According to *The Java™ Language Specification, Third Edition*, Java is "a general-purpose concurrent class-based object-oriented programming language" [G+05, p. 1]. Hence, explicit attention has been given in the construction of Java to the ability of expressing concurrency. Concurrency in Java is realised using threads and synchronised objects [San04]. The following sections contain, respectively, a description of threads and synchronisation in Java.

2.1.1 Threads

The first edition of *The Java™ Language Specification* defines a thread as "a single sequential flow of control" [G+96, p. 587]. A Java program may spawn several threads, hence supporting concurrent programming. Threads in Java are encapsulated in the Thread class from the java.lang package. Hence, a

new thread can be created by making an instance of a class which extends Thread. A thread does not start to execute until its start method has been invoked. The call to start effects the *Java Virtual Machine* (JVM) to call the run method in the new thread. The run method should be overridden in the subclass since this implements what the thread actually executes once it starts running [JAS, Thread].

Inheriting from Thread poses a problem if a subclass also needs to extend some other class since multiple inheritance is not supported in Java. In that case, the inheritance from Thread can be substituted with implementing the Runnable interface which provides only the abstract method run. This allows for the creation of new threads by passing an instance of the new class to a Thread constructor. Again, the start method must be called, and the new class must implement the run method since this constitutes the implementation of what the new thread should actually realise [JAS, Runnable].

2.1.2 Synchronisation

Threads in a Java program "independently execute code that operates on values and objects residing in a shared main memory" [G+05, p. 553]. Hence, when several threads execute concurrently, synchronisation between them may be required as is always the case in the advent of shared memory. The synchronisation is realised using monitors since "Each object in Java is associated with a monitor, which a thread can lock or unlock" [G+05, p. 554]. Since only one thread can hold a given lock on a monitor at a time, other threads attempting to acquire the lock are blocked and their execution is suspended. The fact that each object is associated with a monitor implies that the programmer does not need to manipulate semaphores directly to obtain synchronisation [San04].

Competition synchronisation is implemented using the synchronized keyword which marks a critical section and may appear in relation to blocks of statements as well as methods. When a block is synchronized, a reference to an object must be specified. This is the case since a monitor must be available to realise the synchronisation. The actual statements in the block are not executed until a lock on the monitor of the specified object is obtained.

When a synchronized method is invoked, it automatically attempts to lock the monitor associated with the object on which the method must operate. The body of the method is not executed until the appropriate lock is obtained. When the execution of the synchronized block or method has finished, any locks held are automatically released [G+05].

Co-operation synchronisation can be obtained using the methods wait, notify, and notifyAll. These methods are all found in the Object class which implies that they can be invoked on every object [JAS, Object].

The wait method can either be called without arguments or with a timeout period. In either case, the thread enters a state in which it suspends its execution. The thread waits either indefinitely, for the specified amount of time, or until it is interrupted by another thread. In the latter case, a checked exception is thrown which the thread must handle. The wait method can only be called within a synchronized block or method since the thread must hold the lock on the object. This is an implication of the fact that once wait has been called, the lock on the object is released which makes it possible for another thread to obtain the lock.

A thread which is in a state of waiting continues to be so until another thread invokes the notify or notifyAll method. It may, however, also leave its state of waiting if wait was invoked with a specified amount of time and this expires. The difference between the two notification methods is that notify wakes up a single arbitrary thread waiting on the lock of the given object, whereas notifyAll wakes up all threads waiting for the lock. When one or more awaken threads are trying to acquire the given lock, they do so in competition with any other threads requesting the lock. Hence, there is no guarantee that the thread chosen to execute is the one which has been waiting for the longest period of time.

Similarly to the wait method, notify and notifyAll can only be called within a synchronized block or method. However, in contrast to the wait method, notify and notifyAll do not cause the lock to be released immediately. They only prepare one or more threads such that they can be executed when the lock is released by either a call to wait or at the end of the synchronized block or method.

2.1.3 Concurrency Utilities in Java 5

In Java 5, a Concurrency Utilities package was included. The java.util.concurrent package comprises a number of useful abstractions in relation to concurrent programming such as concurrent collections and synchronisation classes like semaphores, mutexes, and barriers.

The goal of the package was to provide the programmer with “a powerful, extensible framework of high-performance threading utilities” [JCU]. On the same note, the idea behind the framework was to simplify working with

concurrency in a program since the programmer would otherwise have to implement the utilities himself.

2.2 Strengths and Weaknesses

This section presents some strengths and weaknesses of Java's concurrency model.

2.2.1 Threads

Threads are encapsulated in the Thread class and the Runnable interface. Hence, threads appear as objects and are instantiated using the new keyword which complies with Java's object-oriented model. Having the option of either extending Thread or implementing Runnable allows a class which should realise a thread to extend another non-thread class. Furthermore, it induces flexibility that there are more ways of realising the same construction. However, it also requires the programmer to master two constructs which have fundamentally equivalent semantics. Both encapsulations of threads allow the programmer to concentrate on what the threads should execute in their run methods. Hence, threads in Java support modelling concurrent problems since it is often natural to assign the various entities in the problem to separate threads.

The separation of the start-up of a thread from its instantiation allows for manipulations on the thread before it begins its execution. However, the separation can also be a source of error if the programmer fails to invoke the start method. Similarly, it is not legal to invoke start several times.

2.2.2 Synchronisation

Synchronisation is inherently supported in Java via the monitor which is associated with every object. Competition synchronisation can be obtained without much effort using the synchronized keyword (see Section 2.1.2 on page 6). That is, the programmer does not need to manipulate semaphores directly which induces a higher degree of reliability since unblocking actions are not forgotten or lost.

When using monitors, the problem of nested monitors may appear [Lea99]. This is the scenario where a thread obtains a lock on one object and subsequently requests a lock on another object. If another thread holds the lock

on the latter object, the first thread becomes blocked but the lock on the first object is not released. Hence, other threads cannot obtain this lock which may result in the thread being blocked interminably. This scenario can be a major source of error in concurrent programs and difficult to debug.

Synchronisation can also be obtained explicitly using the wait, notify, and notifyAll methods which enable co-operation synchronisation (see Section 2.1.2). The methods are placed in the Object class making it possible to synchronise on any object. This could be seen as an advantage since the flexibility of the methods gives the "thread programmer considerable freedom" [San04, p. 24]. However, there are also disadvantages of the possibility of synchronising on any object. Firstly, the programmer has to maintain control of which objects were used for synchronisation. Secondly, invoking notify or notifyAll on an object, on which wait has not been invoked, is not an error. Hence, the programmer may think that the synchronisation patterns in a program are correct when in fact notifications are lost and threads may remain suspended interminably. That is, the flexibility which the methods provide also makes Java's concurrency model error prone [San04].

In relation to the wait, notify, and notifyAll methods, an issue is the semantics of Java's signalling mechanism. This involves there being little or no control over which thread is awaken by a notify or which thread is allowed to execute following a notifyAll. The choice is made non-deterministically and as a consequence, there are no guarantees of timing between threads since it may not be the thread which has been blocked for the longest period of time which is chosen. However, this non-determinism can also be seen as a strength since it forces the programmer not to assume any deterministic sequence of executions between threads.

2.2.3 Other Issues

The introduction of the java.util.concurrent package can be seen as an admission to the fact that designing multi-threaded programs is challenging. This is particularly prominent in relation to synchronisation issues which "often intrude into apparently unrelated aspects of class design, leading to unnecessary conceptual overhead and code complexity" [Lea99, p. 219]. More simply put, concurrency invades a program; once threads have been introduced, everything must be made thread-safe. As a consequence, the fundamental object-oriented principle of encapsulation may be violated. This is the case since it may be necessary for a class to get access to implementation details of the classes with which synchronisation must be made.

Java is prone to several classical concurrency issues. An example of these is deadlock if a thread, which has invoked wait, never receives a notification. Furthermore, a thread may experience starvation since the next thread allowed to execute is chosen non-deterministically. It is, however, possible for a programmer to exercise some control over which thread is chosen by the scheduler. Every Thread object has an associated priority which the programmer can assign and thus enforce a hierarchy between threads. However, using priorities invariably induces the danger of priority inversion.

Like some other object-oriented languages, Java suffers from the inheritance anomaly [MY93]. That is, a problem may arise when inheriting methods which are involved in obtaining concurrency. The programmer has to take the intended synchronisation into account when overriding an inherited method. This is the case whether the synchronisation is obtained using the synchronized construct or the wait, notify, and notifyAll methods.

Chapter 3

Design Criteria

Section 1.3 on page 2 introduced some rather general criteria for the new concurrency model: it is targeted at the Java language, it should support modelling naturally concurrent problems as well parallel problems, and it should be developer friendly, high-level, and modern. This chapter presents four design criteria which are more specific. Some of the criteria are derived from the problems with Java's present concurrency model which were described in Section 2.2 on page 8. A common trait in the design criteria is that they represent characteristics which a concurrency model should ideally have. It may be difficult to fulfil the criteria simultaneously which is addressed in Section 3.5 on page 14 by prioritising the criteria.

3.1 Object-oriented Model

The integration of concurrency into object-oriented languages has proven to be difficult. In [Pap89, p. 34], Papathomas states that this is the case since "Concurrency is not orthogonal to other aspects of object-oriented programming". Encapsulation and inheritance are prominent features of the object-oriented paradigm, and problems may occur in the form of broken encapsulation and the inheritance anomaly when concurrency has to be introduced into an object-oriented language [DH07, Section 2.3].

The new concurrency model is to build on the Java programming language. This implies that it should be compatible with the object-oriented programming model in general and that of Java in particular. That is, the concurrency model should adhere to the general principles of the object-oriented paradigm such as supporting inheritance, encapsulation, and reusability. Furthermore,

the model should not introduce errors or pose limits on the existing programming model.

Besides from remedying the problems introduced by concurrency in the object-oriented paradigm, the concurrency model should also appear as an integrated part of the paradigm. This entails that the concurrency model should readily support the programmer in applying an object-oriented approach to modelling concurrent programs.

3.2 Expressiveness

Fundamentally, a concurrency model can be applied to the development of concurrent software in two ways. Either when modelling naturally concurrent problems or when introducing parallelism to a program [DH07, Section 2.4]. In Java, the former is well supported whereas the latter is less so since Java was developed under the assumption that programs are executed on a single-core architecture. Ideally, the new concurrency model should support introducing parallelism and modelling naturally concurrent problems equally well. Hence, the abstractions and constructs of the model should be expressive enough to embrace both scenarios. Furthermore, it should be possible to model classical concurrency constructs like locks and semaphores using the new concurrency model in order to demonstrate its expressiveness.

A concurrency model can feature either implicit or explicit concurrency constructs or both [DH07, Section 3.1]. Explicit concurrency provides programmers with the constructs needed to model naturally concurrent problems. Explicit constructs can also be applied when introducing parallelism but they induce an overhead in the form of low-level modelling. Implicit concurrency constructs can remedy this shortcoming. Since the new concurrency model should have a high degree of expressiveness, it is desirable to provide the programmer with explicit as well as implicit concurrency constructs.

Shared data is an integral part of concurrent programming. The protection of such data can be obtained at different levels in a program, i.e. it can be specified on large blocks such as methods or small blocks such as individual variables. The new model should be able to handle both levels of concurrency such that a programmer is not restrained when introducing concurrency to a program.

3.3 Fault Restriction

Java's concurrency model has only very limited fault restriction. This implies that there is almost no support offered to the programmer with regard to preventing him from introducing concurrency errors which are not inherent to the problem which is being modelled. There are no rules in Java which prevents the programmer from combining the few available concurrency constructs in problematic ways. Java only provides the constructs, leaving the responsibility for correctness to the programmer. Since the programmer must also model his problem in a low-level manner, there is a high risk of introducing errors. The concurrency constructs in Java also have a tendency of easily creating concurrency problems under simple uses. This could e.g. be in the form of lost notifications (see Section 2.2.2 on page 8).

Protection of shared data is fundamental to concurrent programming. With regard to fault restriction, this should be lifted away from the programmer and instead primarily handled by the concurrency model. Another prominent source of error in concurrent programs is classical concurrency issues such as race conditions, deadlocks, livelocks, and resource starvation. The impact of such issues should be removed or severely reduced in the new concurrency model.

Based on the above, the new concurrency model should enable the programmer to model concurrency with some level of fault restriction. That is, the programmer should be prevented from introducing concurrency errors without his knowledge.

3.4 Simple

Java's concurrency model could be said to be simple since there are few constructs which are still very expressive. However, with regard to usability, the model leaves the majority of the work to the programmer. Furthermore, Java is very verbose which implies that the programmer has to write a lot of code in order to introduce the desired concurrency in a program.

The concurrency constructs in Java form a homogeneous concurrency API since synchronisation is achieved using the same constructs regardless of its purpose. The new concurrency model should preferably also appear homogeneous and coherent instead of consisting of a range of constructs which all serve a limited and specialised purpose.

Interaction between threads is a prerequisite of concurrent programming. This implies that interleaving of threads should be easy to comprehend and

variable protection schemes easy to apply. Furthermore, the concurrency model should not affect modelling sequential problems.

With reference to this, the new model should be simple. That is, the programmer should be able to express concurrency in a short and concise manner but only when it is necessary to model the problem in question.

3.5 Prioritisation of Criteria

The specific criteria in Sections 3.1–3.4 are all desirable properties and ideally the new concurrency should adhere fully to them all. However, this may not be possible e.g. because some of the criteria may be in conflict with each other. Hence, when developing the concurrency model, choices of which criteria to favour have to be made when a conflict is encountered and a compromise has to be reached.

Various potential conflicts exist between the four criteria and three such conflicts are addressed in the following. The first is between the object-oriented model and fault restriction criteria. The object-oriented model enforces certain design principles which are utilised to induce structure in programs. While the structures may be intuitive when modelling sequential programs, they may not necessarily be suitable for concurrent programs. Enforcing the object-oriented model criterion may therefore result in a concurrency model which is difficult to also make fault restricted. Oppositely, it may be necessary to relax the object-oriented design principles to obtain a acceptable restriction of concurrency errors.

The second conflict is between the expressiveness and fault restriction criteria. Typically, fault restriction is induced by preventing the programmer from introducing concurrency errors. Any kind of restriction also reduces the expressiveness of the language. Similarly, if the language has a high degree of expressiveness, the programmer is able to model any kind of intricate concurrent problem, including faulty ones.

The last conflict is between the expressiveness and simple criteria. There is not necessarily a conflict involved in a concurrency model which is highly expressive while it is also simple. Java's present concurrency model could be said to fulfil both criteria. However, there may be a potential conflict if the concurrency constructs prohibit the programmer from making a simple model of the concurrent problem. Similarly, if the constructs are very simple, it may pose a limit on the expressiveness of the language since there are concurrent problems which cannot be modelled using the constructs.

In order to determine how conflicts should be resolved, the criteria are prioritised. The three priority levels are presented in Table 3.1. In case criteria at the same priority level are in conflict, they are prioritised according to the order in which they appear in Table 3.1.

Priority	Criteria
1.	Object-oriented model Expressiveness
2.	Fault restriction Simple

Table 3.1: Prioritisation of criteria.

The prioritisation is based on the importance of the design criteria and the potential conflicts. The object-oriented model and expressiveness criteria are favoured because one of the overall goals of the concurrency model is to be well integrated into the object-oriented model. At the same time, the concurrency model should be at least as useful as the present concurrency model in Java, and hence it needs to be expressive. The fault restriction and simple criteria are at the second level because of the potential conflicts with the two criteria at the first level. It is important to develop a fault restricted concurrency model but it should not be valued over expressiveness or influence the object-oriented design principles. Similarly, simplicity could be valuable in the concurrency model but it is not rated as valuable as the criteria at higher levels.

Chapter 4

Concurrency Model Assessment

As mentioned in Section 1.3.3 on page 4, it is useful to be able to assess the strengths and weaknesses of a given existing concurrency model in a programming language. There is also a need for assessment when developing a new concurrency model or modifying an existing one. Moreover, a method is needed for comparing multiple concurrency models.

This chapter presents a new question based method for assessing and comparing concurrency models. Section 4.1 contains criteria to which an optimal method should adhere. This is followed by a presentation of the actual method in Section 4.2 and an evaluation of it in Section 4.3 on page 18. Finally, the questions used in the method for this project are included in Section 4.4 on page 19.

4.1 Criteria for a Method

Section 10.1 on page 98 presents two approaches to assessing concurrency models. The descriptions reveal various problems of the methods which can be used to set up criteria to which a method should ideally adhere. The criteria are listed and described below.

- **Absolute**: The method should be absolute, implying that it is suitable for assessing a single concurrency model. That is, the assessment should not be dependent on the assessment of other models. In addition, the method should also embrace the comparison of multiple models.

- **Unbiased**: The method should be unbiased, implying that it is suitable for assessing the concurrency model of any object-oriented language. That is, the method should not favour particular languages.
- **Transparent**: The method should be transparent, implying that it is easily established how a given model has obtained its assessment. This criterion also induces objectivity in the method since the assessment of a given model may depend on who performs the assessment. However, by making the method transparent, it can easily be uncovered in which areas one disagrees with the assessment.

4.2 Question Based Method

This section describes the assessment method which is developed and applied to compare Cava and Java. The basic strategy of the method is to formulate a series of questions. The answers then form the basis for the actual assessment. In the present context, the questions are based on the design criteria for Cava found in Chapter 3 on page 11 since they are the driving force in the development. In other contexts, the questions may e.g. be based on desirable characteristics of a concurrency model.

The method has the design criteria as its starting point so it should be able to assess the model with regard to how well these criteria are fulfilled. Since it may be difficult to fulfil all the design criteria equally well, they were prioritised in Table 3.1 on page 15, and the method should take this prioritisation into account. Fundamentally, the method consists of the two steps described below. The two steps are described in further detail in the following sections.

1. Assessing the design criteria individually.
2. Constructing an overall assessment.

4.2.1 Assessment of Individual Criteria

The assessment of the individual design criteria is done by formulating a series of questions which in combination uncovers the properties of the criterion. In this case, the questions are all formulated such that it is preferable if the answer to them is yes. Based on the answers to the questions, the criterion is assigned a numerical value which shows how well the concurrency model

adheres to the given criterion. Depending on how much information about the criterion the individual question carries, a weighting between the questions could be induced. Since a numerical value is assigned to each criterion, the goal of having an absolute method is fulfilled. Similarly, it is transparent since the questions are available and an analysis of the answers and weights, if applied, reveals how a model obtained its assessment.

The above approach of formulating questions and assigning numerical values can be applied regardless of which design criteria or other characteristics they are intended to evaluate. This is simply reflected in the particular questions. Section 4.4 presents the particular questions which are formulated in relation to the design criteria in Chapter 3 on page 11.

4.2.2 Overall Assessment

The overall assessment appears as a combination of the individual assessments. That is, a single numerical value is assigned to the concurrency model based on the numerical values assigned to the individual design criteria. This value shows how well the concurrency model adheres to the design criteria. A weighting between the individual assessments could be induced according to the prioritisation between the criteria.

4.3 Evaluation of the Method

The method enables the absolute assessment of a single model since a numerical value is assigned to this. Furthermore, the questions define a maximum score which is obtained if a model can answer yes to all questions. This allows for evaluating how close a model is to possessing the qualities which are being assessed by the method and identifying the areas in which a model could be improved.

In order to fully understand and evaluate an assessment, it may be necessary to investigate this in detail. The availability of the questions and answers enables such an investigation which induces transparency to the method. This is perceived as a strength since it is possible to locate the exact areas in which one might disagree with the assessment. Another strength is that the individual models do not affect the scores of other models when assessing multiple models.

The unbiased criterion is fulfilled for the method itself but the particular questions may make the method biased. However, the availability of the

questions counters the risk. This makes the method objective since biased questions cannot be hidden.

With the above approach, two concurrency models may very well be assigned the same numerical value. However, this does not necessarily imply any kind of similarity since the overall assessment is a combination of numerous individual assessments. Hence, in order to perform an actual comparison of the characteristics of two models with the same overall assessment, the scores of the individual assessments must be investigated in detail.

4.4 Questions

This section presents the questions which are formulated when applying the question based method to the design criteria of Chapter 3. Moreover, it is described what is required for the answer to be either yes or no.

4.4.1 Object-oriented Model

The questions in relation to the object-oriented model criterion are listed below. There are eight questions which reveal various aspects of the criterion such as fundamental object-oriented characteristics like encapsulation, inheritance, and reusability. Furthermore, mismatches between concurrency and the object-oriented model are addressed.

- *Is the concurrency model integrated into the object-oriented model?*
  - The answer is yes if some part of the concurrency model is integrated into the main programming language. If the concurrency model is provided only as a library extension without affecting the main programming language, the answer is no.
- *Does the concurrency model protect against broken encapsulation?*
  - The answer is yes if the concurrency model is not disposed towards introducing broken encapsulation. Oppositely, the answer is no if the concurrency constructs are difficult to apply without breaking the encapsulation established by the object-oriented model. This includes if the programmer is forced to open the encapsulation and allow direct access to the internal synchronisation of a given object when it is applied in another (part of) the program.

- *Does the concurrency model protect against the inheritance anomaly?*
  - The answer is yes if the concurrency model is not disposed towards introducing the inheritance anomaly. The answer is no if overriding a class containing concurrency constructs can lead to the inheritance anomaly in the subclass.

- *Does the concurrency model support reuse of existing classes?*
  - The answer is yes if a class, which was implemented utilising concurrency features, can be reused without modifications and with correct synchronisation when implementing other classes. If the class requires modifications in order to function with the intended semantics, the answer is no.

- *Are objects utilised to encapsulate concurrency constructs?*
  - The answer is yes if the concurrency model contains some kind of concurrency constructs or concepts which are encapsulated inside some kind of object. That is, if objects are utilised to induce structure to concurrency constructs.

- *Are thread constructs encapsulated in objects?*
  - The answer is yes if threads are bound to object instances such that threads can be referenced as any other object.

- *Are co-operation synchronisation constructs encapsulated in objects?*
  - The answer is yes if co-operation synchronisation constructs are bound to object instances such that they can be referenced as any other object. This is an advantage since co-operation synchronisation is an integral part of concurrent problems, and thus modelling co-operation can be done in an object-oriented way.

- *Are competition synchronisation constructs integrated directly into the language?*
  - The answer is yes if competition synchronisation constructs are independent of object instances. This is an advantage since competition synchronisation is an inherent and unavoidable part of concurrent programming and should not be modelled as part of the object-oriented design. Hence, competition synchronisation is better handled at the language level in a orthogonal manner.

4.4.2 Expressiveness

The questions in relation to the expressiveness criterion are listed below. There are 13 questions which reveal various aspects of the criterion such as how well the model supports modelling concurrent and parallel problems, the level on which concurrency is obtained, and whether it is possible to model classical concurrency constructs like locks and semaphores.

- *Are constructs available to the programmer to introduce concurrency?*
  - The answer is yes if the concurrency model includes constructs which can be applied to model concurrent problems in a natural way. If the programmer is forced to convert the concurrent problem into a parallel problem, the answer is no.

- *Are naturally concurrent problems easily modelled without introducing concurrency errors which are not inherent to the problem?*
  - The answer is yes if the concurrency model allows a programmer to model a naturally concurrent problem without introducing concurrency errors which are not part of the original problem. Furthermore, the programmer should be able to do this in a timely fashion if he is fairly knowledgeable about the problem and the programming language.

- *Are parallel constructs available to the programmer to introduce parallelism?*
  - The answer is yes if the concurrency model includes constructs which can be applied to model parallelism in a natural way. An example of such a construct is for-loops which can be executed in parallel without the programmer having to implement this explicitly. If the programmer is forced to convert the parallel problem intro a concurrent problem to achieve parallelism, the answer is no.

- *Are parallel problems easily modelled without introducing concurrency errors?*
  - The answer is yes if the concurrency model allows a programmer to model a parallel problem without introducing concurrency errors. Furthermore, the programmer should be able to do this in a timely fashion if he is fairly knowledgeable about the problem and the programming language.

- *Is coarse-grained concurrency easily obtained?*
  - The answer is yes if the concurrency model allows the programmer to directly model coarse-grained concurrency which is characterised in that concurrent access to data is granted in large blocks. Note, the coarse-grained protection pattern may result in inefficient concurrency since a thread may protect data which it does not access. Thus, it prevents other threads from accessing it.

- *Is fine-grained concurrency easily obtained?*
  - The answer is yes if the concurrency model allows the programmer to directly model fine-grained concurrency which is characterised in that concurrent access to data is granted in small blocks. Note, the fine-grained protection pattern enables achieving optimal concurrency since a thread only protects the data it needs against race conditions. That is, any data which is not accessed by a thread may be accessed by any other thread.

- *Are the synchronisation constructs composable?*
  - The answer is yes if the synchronisation constructs can be combined to implement more intricate synchronisation constructs and this can be done without introducing concurrency problems.

- *Is it possible to model a lock?*
  - The answer is yes if a lock can be modelled in the concurrency model. The lock should be usable as a synchronisation mechanism when implementing concurrent problems.

- *Is it possible to model a semaphore?*
  - The answer is yes if a semaphore can be modelled in the concurrency model. The semaphore should be usable as a synchronisation mechanism when implementing concurrent problems.

- *Is it possible to model message passing?*
  - The answer is yes if message passing can be modelled in the concurrency model. The message passing construct should be usable as a synchronisation mechanism when implementing concurrent problems.

- *Does the concurrency model include shared variables?*
  - The answer is yes if the concurrency model features shared variables. These can be accessed by multiple threads and if they are changed by one thread, other threads are able to see the changes.

The answer is no if the concurrency model does not include such variables.

- *Can thread execution be prioritised?*
  - The answer is yes if the concurrency model includes a mechanism for prioritising the order of thread execution. This allows the programmer to specify important threads and ensure that these threads receive sufficient resources. If the programmer must build prioritisation into the thread implementation manually, the answer is no.

- *Can suspended threads be interrupted?*
  - The answer is yes if the concurrency model includes a mechanism for resuming a thread which has been suspended. This implies that the programmer should be allowed to resume a suspended thread even though the condition, on which the thread was suspended, has not been fulfilled. If a suspended thread can only be resumed by fulfilling the condition, the answer is no.

4.4.3 Fault Restriction

The questions in relation to the fault restriction criterion are listed below. There are 14 questions which reveal various aspects of the criterion such as how threads are managed, whether the model avoids classical concurrency problems such as race conditions, deadlocks, livelocks, and resource starvation, and how variables are protected against corruption.

- *Is thread management handled by the concurrency model?*
  - The answer is yes if the concurrency model exclusively handles thread management such as starting, suspending, and stopping threads. If the programmer is allowed to exercise such low-level management, the answer is no.

- *Is the model free from resource starvation?*
  - The answer is yes if waiting threads are resumed in the order which they were suspended. That is, scheduling and resource sharing are fair between threads. If a thread is allowed access to a resource which other threads have waited on for a longer period of time, the answer is no.

- *Is the model free from priority inversion?*
  - The answer is yes if a low priority thread is never allowed to block high priority threads competing for the same resources.

- *Does the fundamental concurrency model rely on a mathematical foundation?*
  - The answer is yes if the concurrency model is built on top of a mathematical model, e.g. a complete formal semantics for the concurrency model. If the model is built without an underlying mathematical system, the answer is no.

- *Is non-determinism only present as an option to the programmer?*
  - The answer is yes if program execution is guaranteed to be deterministic unless the programmer explicitly has requested non-deterministic behaviour. The request could apply to the entire program or only parts of it. If the concurrency model only supports non-deterministic behaviour, the answer is no.

- *Are shared variables protected against corruption by default?*
  - The answer is yes if shared variables by default are not subjected to race conditions, i.e. if the programmer does not have to express the actual synchronisation. If changes to shared variables may result in race conditions without the programmer being forced by the concurrency model to apply explicit synchronisation, the answer is no.

- *Is a variable initialised for all occurrences once it has been initialised?*
  - The answer is yes if a variable is initialised for all thread instances once it has been initialised. That is, if a thread initialises a variable, it can be accessed by any other thread without these have to initialise the variable again. If this is not the case, and other threads have to initialise the variable themselves, the answer is no.

- *Is the concurrency model disinclined to runtime exceptions which appear due to timing issues?*
  - The answer is yes if the concurrency model automatically handles synchronisation of concurrency constructs. If runtime exceptions can be raised because of timing issues when using the concurrency constructs, the answer is no. That is, if the programmer does not ensure sufficient synchronisation between threads, an exception can be raised when invoking concurrency mechanisms.

- *Is it impossible to experience race conditions?*
  - The answer is yes if the programmer cannot create a race condition between threads.

- *Is it impossible to create a deadlock between two threads?*
  - The answer is yes if the programmer cannot create a deadlock between two threads in the concurrency model. If a deadlock can be created, the answer is no. The latter constitutes the minimal case which implies that deadlocks between more than two threads can also be created.

- *Is it impossible to create a livelock between two threads?*
  - The answer is yes if the programmer cannot create a livelock between two threads in the concurrency model. If a livelock can be created, the answer is no. The latter constitutes the minimal case which implies that livelocks between more than two threads can also be created.

- *Does the concurrency model by default protect against race conditions?*
  - The answer is yes if the programmer must take explicit action to introduce the possibility of race conditions. That is, race conditions do not appear in the program unless the programmer allows it. If race conditions can arise without the programmer having explicitly allowed it, the answer is no.

- *Does the concurrency model by default protect against deadlock?*
  - The answer is yes if the programmer must take explicit action to introduce the possibility of deadlocks. That is, deadlocks do not appear in the program unless the programmer allows it. If deadlocks can arise without the programmer having explicitly allowed it, the answer is no.

- *Does the concurrency model by default protect against livelock?*
  - The answer is yes if the programmer must take explicit action to introduce the possibility of livelocks. That is, livelocks do not appear in the program unless the programmer allows it. If livelocks can arise without the programmer having explicitly allowed it, the answer is no.

4.4.4 Simple

The questions in relation to the simple criterion are listed below. There are nine questions which reveal various aspects of the criterion such as how the model affects modelling sequential problems, whether the rules of the concurrency model are enforced on compile time, how easy it is to comprehend e.g. interleaving of threads and variable protection schemes, and the homogeneity of the various constructs of the concurrency model.

- *Is it possible to model sequential problems without considering concurrency?*
  - The answer is yes if the concurrency model does not interfere with writing sequential programs. Hence, these do not become more complex than they would be if the programming language did not support concurrency. If the programmer is forced to consider concurrency features when modelling sequential problems, the answer is no.

- *Is code executed in a sequential order similar to sequential programs?*
  - The answer is yes if the concurrency model executes code sequentially, statement upon statement. By applying this execution pattern, the concurrent programming model simulates sequential programs, and therefore the concurrency model becomes readily understandable to sequential programmers. If the concurrency model executes code non-sequentially, e.g. by trying and retrying to execute blocks, the answer is no.

- *Does the model consist only of a few simple elements?*
  - The answer is yes if concurrency is built around a small number of behaviourally simple elements. That is, the concurrency model consists of a number of elements which can be combined into larger constructs. If the concurrency model has a large number of constructs, each handling a dedicated usage scenario, the answer is no.

- *Is competition and co-operation synchronisation achieved using the same language constructs?*
  - The answer is yes if the concurrency model applies the same keywords and constructs to achieve competition and co-operation synchronisation. That is, the concurrency model contains one united set of keywords and constructs which are applicable to all purposes. If the concurrency model features one set of keywords and

constructs for competition synchronisation, and another for co-operation synchronisation, the answer is no.

- *Are sequential and implicitly parallel constructs applied using the same keywords?*
  - The answer is yes if the concurrency model applies traditionally sequential keywords for implicitly parallel constructs. An example is if the keyword for is applied to both sequential for-loops and parallel for-loops. If the parallel for-loop instead applies a keyword like e.g. parfor, the answer is no.

- *Are most rules enforced on compile time?*
  - The answer is yes if rules, which are defined by the concurrency model, are checkable on compile time. Furthermore, the answer is only yes if the concurrency model requires that the rules are checked. If all rules defined by the concurrency model are only checkable on runtime, the answer is no.

- *Is potential interleaving between threads easy to comprehend and identify?*
  - The answer is yes if the concurrency model only exhibits a small number of possible interleaving patterns between threads. That is, the programmer can identify potential thread interleaving in a straightforward manner and take this into consideration. If a large number of possible interleaving patterns are possible, and the patterns are complex, the answer is no.

- *Are variable protection schemes easy to comprehend?*
  - The answer is yes if the concurrency model has a simple, easily understandable, and structured way of protecting variables from race conditions. Hence, the concurrency model applies a general and straightforward scheme for variable protection. If the concurrency model applies a large number of specialised constructs to enforce different kinds of variable protection, the answer is no.

- *Is the concurrency model separated from the computer architecture?*
  - The answer is yes if the concurrency model is not bound to specific computer architectures. By keeping the concurrency model separate, concurrency abstractions are not subjected to hardware limitations. If the concurrency constructs are based on specific hardware features, the answer is no.

Chapter 5

Concurrency in Cava

This chapter describes the design of Cava. The new concurrency constructs are introduced by first presenting a high-level overview of Cava in Section 5.1. This is followed by the Cava syntax and informal semantics in Section 5.2 on page 31. Note that examples of applying Cava are included in Chapter 6 on page 50.

5.1 Introduction

This section contains a high-level introduction to the constructs in Cava.

5.1.1 Differences from Java

The foundation of Cava is the Java programming language. Most parts of Java are still available in Cava with a few exceptions: the synchronized keyword is removed as are the methods wait, notify, and notifyAll on the Java Object class. All remaining parts of Java are still valid in Cava.

5.1.2 Threads

In Cava, concurrency is introduced using a thread construct. It bears some resemblance to the threads in Java, but there are also differences between the constructs. The strategy in both languages is to encapsulate threads inside special Thread objects, which are instances of the Thread class. In Cava, a thread starts to execute immediately when the thread object has

been created. The code which the thread executes can be defined in two ways: in the run method in a class extending the Thread class or in the run method in a class implementing the Runnable interface.

Threads in Cava have one significant runtime difference compared to threads in Java. In Cava, all variables are by default local to each thread. Hence, such thread local variables can potentially have different values in each thread instance. When a variable is thread local, the thread can read and change the variable but the values are only visible to the given thread.

Cava features two different kinds of threads. These are termed deterministic and non-deterministic threads with reference to the way they are executed, or more precisely how they interact. When a deterministic thread needs to perform an operation, and the result could be affected by or affect another deterministic thread, the thread is suspended until it can be ensured that the thread operations will proceed in a predefined order. This semantics implies that programs, which solely use deterministic threads, always return the same result.

The execution of non-deterministic threads is similar to the execution of threads in Java. That is, non-deterministic threads are not subjected to an order on operations which can affect other threads.

5.1.3 Shared Variables

As described in the previous section, all variables in Cava are thread local by default. If a variable should not be thread local, it can be made available to several threads by using the shared keyword in the variable declaration. If a variable is shared, all threads will be reading or changing the same variable, and they all see the results of other threads changing the variable.

The usage of shared variables is restricted since they cannot be read or changed inside a conventional method. This is only allowed inside transaction methods or transaction blocks (see the following section).

5.1.4 Transactions

Any non-constructor method in Cava can be specified as a transaction method by using the transaction keyword in the method declaration. When a method is marked as a transaction method, the semantics of its execution is changed compared to conventional methods. The difference lies in that Cava applies *Software Transactional Memory* (STM). Hence, a transaction method is executed as an STM transaction which implies that it is executed atomically,

i.e. without interleaving [HLM06]. Seen from all other threads, a transaction method may commit its operations and complete successfully or abort its operations and appear to have not run at all. If a transaction is aborted, its operations are rolled back and it is re-executed at some later time. Section 10.2 on page 101 contains a more detailed description of STM and how it can be implemented.

Cava also features transaction blocks. These are defined inside methods and code inside such a block is executed with STM semantics. Transaction blocks are also specified using the transaction keyword.

A transaction method or block can be programmed in much the same way as a conventional method since it can call other methods, both conventional and transactions. However, these calls are part of the transactional semantics of the transaction method or block, implying that the changes made by the called methods are also under STM semantics. In contradiction to conventional methods, transaction methods and blocks are allowed to access both thread local and shared variables. This implies that shared variables can be used similarly to local variables.

There is one special rule attached to the usage of transaction methods. This states that a method, which contains multiple calls to transaction methods, must itself be a transaction method. This rule, however, may be circumvented by enclosing the calls to transaction methods in transaction blocks instead.

5.1.5 Gates

Gates are a message passing mechanism encapsulated in objects. The idea is that a message, which may or may not carry a value in the form of an object, can be sent to a gate. The messages are then retrieved by invoking a receive method on the gate which returns the sent message. If receive is invoked at a time when a message is not waiting in the gate, the thread is blocked until a message becomes available.

Gates are divided into two different kinds: gates with "or" semantics and gates with "and" semantics. An or-gate forwards a message to one and only one recipient (i.e. thread). An and-gate forwards all messages to all recipients which subscribe to the gate. That is, a thread has to invoke a subscribe method before it can receive messages from an and-gate. Two classes OrGate and AndGate define the two kinds of gates.

Gates can be joined together to form a network of gates. In the network, messages are passed between the gates when a reception occurs on one of the

gates. That is, when receive is invoked on one of the gates, this propagates through the network until it reaches a gate which contains a message. If there are no signals available in the network, the thread becomes blocked on the invocation of receive. The individual gates in a network preserve their semantics as or-gates or and-gates.

5.1.6 Inheritance

Inheritance in Cava is obtained in the same way as in Java with a few exceptions. The main difference is that transaction methods are visible on inheritance. This implies that on interfaces and classes, it is possible to see which methods are marked with the transaction keyword and which are not. Furthermore, when overriding a method from a super class or an interface, the transactional status of the method must be preserved. Hence, a transaction method must be overridden by a transaction method and a conventional method must be overridden by a conventional method.

5.2 Syntax and Semantics

This section describes Cava's concurrency model in more detail. This is done by specifying the syntax of the new constructs. Furthermore, the semantics of the constructs is addressed at an informal level.

5.2.1 Threads

This section describes the syntax and semantics of threads in Cava. The section builds on the description provided in Section 5.1.2 on page 28.

5.2.1.1 The Thread Class

Cava threads are bound to the Thread class which provides an encapsulation of the construct. A new thread is created when an instance is made of either the Thread class or a class which extends the Thread class. Each thread object only provides one executing thread.

The Thread class contains a run method which is invoked by the thread when it is started. Thus, the actual functionality of the Thread object is implemented in the run method which therefore should be overridden in the derived class. In case a class needs to inherit from another class than Thread, the

thread functionality can be obtained by implementing the Runnable interface. This design is necessary since Cava does not support multiple inheritance. Runnable itself consists only of a run method which has to be overridden in a class which implements the interface. Again, the run method implements the actual functionality of the thread. The Thread class includes a number of constructors which take a Runnable object as a parameter.

The different constructors available in the Thread class are presented in Table 5.1. Note, the difference between deterministic and non-deterministic threads is addressed in Section 5.2.1.2.

Constructor Summary:
Thread() Creates a deterministic thread which executes the run method on the Thread object.
Thread(Runnable runnable) Creates a deterministic thread which executes the run method on the Runnable object.
Thread(boolean nondeterministic) Creates a thread which can be either deterministic or non-deterministic and which executes the run method on the Thread object.
Thread(Runnable runnable, boolean nondeterministic) Creates a thread which can be either deterministic or non-deterministic and which executes the run method on the Runnable object.

Table 5.1: The constructors in the Thread class.

In Cava, all threads are started automatically when they are created. The exact start time of the thread is defined based on whether the Thread object is created inside or outside of a transaction. If the thread is created inside a transaction method or block, the execution of the thread is delayed until the transaction has been committed. This implies that when a transaction commits successfully, any Thread object created inside the transaction is started as an effect of the commit. When a thread is created outside of a transaction, the thread is started as soon as the Thread object has been fully created and initialised. That is, when a Thread object is created with the new keyword, all appropriate constructors on the object and parent objects must complete their execution inside the creating thread before the new thread is started. The result of this is that the thread is started after all object

constructors have been executed but before the object is returned by the new keyword.

Threads in Cava cannot be managed directly by the programmer. This implies that when a thread is first created, the programmer does not have any default handles to control the execution of the thread. That is, methods like stop, suspend, and resume, which were available in early version of Java, are not available in Cava. If a programmer needs to perform explicit management, this must be implemented as part of the functionality of the thread which is easily obtained since threads are bound to Thread objects.

5.2.1.2 Deterministic and Non-deterministic Threads

Deterministic threads are threads which do not display non-deterministic behaviour. The natural non-determinism of threads is removed by enforcing a global ordering on operations performed by the threads. This implies that when a deterministic thread needs to perform an operation which may be affected by or affect another thread, the thread is suspended until it can be ensured that the thread operations will proceed in a predefined order. This semantics implies that programs, which solely use deterministic threads, always return the same result.

Cava automatically manages the order on deterministic threads, so a programmer cannot directly influence the order of threads. The ordering is achieved by assigning a unique id to each thread based on the order in which they are created. All deterministic threads execute concurrently as long as they only access local variables. When a deterministic thread needs to create a new thread or access a shared variable or gate, the thread is suspended. When all deterministic threads have been suspended, the threads are allowed to resume execution in ascending order on their id and perform their non-local operations. This pattern is repeated every time the threads need to perform non-local operations. A collection of threads, which run deterministically to each other, are said to be in the same deterministic zone.

The execution of non-deterministic threads is similar to the execution of threads in Java. That is, non-deterministic threads are not subjected to an order on operations which can affect other threads. This implies that the programmer must ensure that the non-deterministic behaviour of the threads does not cause problems with the semantics of a program.

The default behaviour of threads is deterministic execution. Hence, a call to new Thread() results in a deterministic thread. However, the Thread class has overloaded constructors which makes it possible for the programmer to

specify a Boolean parameter indicating whether the thread should have non-deterministic behaviour (see Table 5.1). Hence, a call to new Thread(true) creates a non-deterministic thread.

A program may utilise both deterministic and non-deterministic threads. When compared to the thread which created it, a thread always exhibits the semantics with which it was created. That is, if a thread creates a deterministic thread, this is in the same deterministic zone as the creating thread. If the thread creates a non-deterministic thread, this is placed in a new deterministic zone. The result of this is that a non-deterministic thread runs non-deterministically compared to its creating thread while a deterministic thread runs deterministically compared to its creating thread.

5.2.1.3 Thread Local Variables

Variables in Cava are thread local by default. That is, unless a variable is marked with the shared keyword, the variable is thread local and each thread operates on its own copy of the variable. This implies that a thread can read and change the values of variables but the values are only visible to the given thread.

When a thread is started, it inherits the values of thread local variables from its creating thread. Hence, all thread local variables, which have been assigned a value in the thread which creates a new Thread object, are copied to the new thread. This implies that a new thread has an updated copy of the values of thread local variables.

When a thread creates an object, the thread local variables inside the object are only initialised for the current thread. That is, the initialisation code provided for the object is executed by the current thread making assignments to thread local variables. The result of this is that thread local variables in the newly created object will not have a defined value for existing threads.

Code 5.1 shows an example of two threads working on the same thread local variable, *value*. The code shows how the threads are created and how the functionality is implemented in the run method. After execution, *value* is 1 in each of the threads. In the main program thread, *value* is still 0.

5.2.2 Shared Variables

This section describes the syntax and semantics of shared variables in Cava. The section builds on the description provided in Section 5.1.3 on page 29.

```
1  public class Example {
2
3    private int value = 0;
4
5    public class ExampleThread extends Thread {
6      public void run() {
7        value++;
8        System.out.println(value);
9      }
10   }
11
12   public void execute() {
13     new ExampleThread();
14     new ExampleThread();
15   }
16 }
```

Code 5.1: Creating two threads with a thread local variable, *value*.

5.2.2.1 The shared Keyword

In some scenarios, it is necessary to have variables which are shared between several threads. This can be obtained by adding the shared keyword to the variable declaration. That is, shared is added to the set of modifiers in Cava which can be seen in Table 5.2 on the next page. Note that transaction is also added as a modifier (see Section 5.2.3 on page 37 for more details). Furthermore, the synchronized keyword is not an access modifier in Cava. All other modifiers from Java are preserved and taken from the formal grammar found in "The Java™ Language Specification, Third Edition" [G+05, pp. 591–592].

5.2.2.2 Application of shared

The shared modifier can only be applied to variables in Cava. That is, methods, classes, and interfaces cannot be marked as shared. Furthermore, only instance variables can be marked as shared. This implies that variables, which are declared within a method, are always local to the given thread invoking the method. If a variable is marked as shared, all threads will be reading or changing the same variable. As a result, they all see the results of other threads changing the variable.

Modifier:
Annotation
public
protected
private
shared
transaction
static
abstract
final
native
transient
volatile
strictfp

Table 5.2: Access modifiers in Cava.

The shared modifier can be combined with the private, default, protected, and public access control modifiers. The access control modifiers define from where a variable can be accessed, and the shared modifier defines the semantics of the variable when it is being accessed. The difference in semantics implies that all combinations of the access control modifiers and the shared modifier are allowed, and that each combination induces a different overall semantics. All combinations of the shared modifier and the access control modifiers can be seen in Code 5.2.

```
1 public shared int publicSharedValue;
2 protected shared int protectedSharedValue;
3 shared int defaultSharedValue;
4 private shared int privateSharedValue;
```

Code 5.2: The combinations of access control modifiers and the shared modifier.

Since shared variables can be accessed by multiple threads, they have to be protected from corruption. This is obtained by requiring that shared variables are only accessed within methods or blocks marked with the transaction keyword.

5.2.3 Transactions

This section describes the syntax and semantics of transactions in Cava. The section builds on the description provided in Section 5.1.4 on page 29.

As mentioned above, access to shared variables has to be protected against corruption. The key idea in transaction methods and blocks is that access is induced with STM semantics. This implies that in case two threads try to access a shared variable simultaneously, only one is allowed to manipulate the variable and the execution of the other thread is aborted. The result of this is that access to the variable becomes interleaved.

5.2.3.1 The transaction Keyword

The STM semantics is obtained by adding the transaction keyword to the set of modifiers in Cava which can be seen in Table 5.2. The transaction keyword can be applied to either a non-constructor method or a block. This is described in the following sections.

5.2.3.2 Transaction Methods

A non-constructor method can be defined as a transaction method by including the transaction keyword in the method declaration. This implies that the entire body of the method is performed atomically as a single transaction. Unlike conventional methods, transaction methods are allowed to access both shared and thread local variables. The changes made by a transaction method on variables can be either committed or rolled back, regardless of whether the variables are thread local or shared.

Code 5.3 on the next page shows an example with a shared variable, *value*, which is incremented using a transaction method, incrementValue. Depending on the order in which the two threads are allowed access to *value*, the output will either be "Thread #1 incremented value to 1" followed by "Thread #2 incremented value to 2", or "Thread #2 incremented value to 1" followed by "Thread #1 incremented value to 2". Notice that the two different results are only possible because the constructor of the ExampleThread class includes a call to super(true) which implies that the threads execute non-deterministically.

Transaction methods are allowed to call both conventional and transaction methods, including constructors. Calls to conventional methods become part of the transactional semantics which implies that the changes made by the

```
1  public class Example {
2
3    private shared int value = 0;
4    private int id = 0;
5
6    private class ExampleThread extends Thread {
7      public ExampleThread () {
8        super(true);
9        id++;
10     }
11
12     public void run() {
13       incrementValue();
14     }
15
16     public transaction void incrementValue() {
17       value++;
18       printValue(value);
19     }
20
21     public void printValue(int value) {
22       System.out.println("Thread #" + id + " incremented value to " + value)
             ;
23     }
24   }
25
26   public void execute() {
27     new ExampleThread();
28     new ExampleThread();
29   }
30 }
```

Code 5.3: Two threads accessing a shared variable, *value*, using a transaction method, incrementValue.

conventional methods are also under STM semantics. Similar to transaction methods, conventional methods are also allowed to invoke both conventional and transaction methods. Code 5.3 illustrates some of these points. The conventional run method invokes a transaction method, incrementValue. The incrementValue method invokes the conventional printValue method. Since the call is made inside a transaction method, printValue is under STM semantics similarly to incrementValue.

Even though conventional methods are allowed to invoke transaction methods, they may only invoke one such method. If multiple transaction methods are invoked, the method must itself be a transaction method or utilise transaction blocks. Code 5.4 shows an example of this. The incrementValue method must be a transaction method since it invokes the transaction methods getValue and setValue. Had incrementValue not been a transaction method, interleaving between two threads invoking incrementValue could produce incorrect results.

```
1  public class Example {
2
3    private shared int value = 0;
4
5    public transaction void incrementValue() {
6        int temp;
7
8        temp = getValue();
9        temp++;
10       setValue(temp);
11   }
12
13   private transaction int getValue() {
14     return value;
15   }
16
17   private transaction void setValue(int value) {
18     this.value = value;
19   }
20 }
```

Code 5.4: A method, incrementValue, invoking multiple transaction methods.

When a transaction method invokes another transaction method, the two methods are executed as a single transaction. That is, Cava features nested

transactions. These are handled as one single large transaction which has several implications. The first is that an inner transaction can see all changes made by parent transactions even before the parent transaction has been committed. This also implies that an inner transaction does not cause the parent transaction to be aborted by accessing the same variables as the parent transaction. Another consequence is that inner and parent transactions are committed or aborted collectively. That is, inner transactions are rolled back together with the outer transaction if a conflict arises. An example of nested transactions can be seen in Code 5.4 where the transaction method, incrementValue, invokes two transaction methods. The getValue and setValue methods are executed as nested transactions to incrementValue.

As mentioned, transaction methods are allowed to access both thread local and shared variables. As a consequence, it is not incorrect to have a transaction method which only accesses thread local variables. However, the problem of two threads wishing to access a variable simultaneously never arises. Hence, the method never has to perform roll backs or re-exectutions and thus, the transaction keyword has no effect on the semantics of the program.

5.2.3.3 Transaction Blocks

The transaction keyword can also be applied to a block of code. This can be seen in Table 5.3 which shows the statements in Cava. The other statements are taken from the formal grammar found in "The Java™ Language Specification, Third Edition" [G+05, p. 590]. Note that since synchronized is not an access modifier in Cava, the Java statement synchronized *ParExpression Block* is excluded from Table 5.3.

The operations within a transaction block are under STM semantics which implies that either none of the operations are performed or all are performed. A method may contain several transaction blocks and each block is under STM semantics. That is, other threads can see the result of changes to shared variables at the end of each block, i.e. before the entire method has completed its execution. Code 5.5 shows an example with a transaction block. The code implements the same functionality as Code 5.4.

As is the case with transaction methods, having a transaction block which only accesses thread local variables is not incorrect. However, once again the transaction keyword is without effect on program semantics. Similarly, it is not incorrect to have transaction blocks within a transaction method or block. This has the semantics of nested transactions.

Using transaction blocks makes it possible to circumvent the rule which states

Statement:
Block
assert *Expression [: Expression]* ;
if *ParExpression Statement [* else *Statement]*
for (*ForControl*) *Statement*
while *ParExpression Statement*
do *Statement* while *ParExpression* ;
try *Block (Catches | [Catches]* finally *Block)*
switch *ParExpression* { *SwitchBlockStatementGroups* }
transaction *Block*
return *[Expression]* ;
throw *Expression* ;
break *[Identifier]*
continue *[Identifier]*
;
StatementExpression ;
Identifier : *Statement*

Table 5.3: Statements in Cava.

```
1  public class Example {
2
3    private shared int value = 0;
4
5    public void incrementValue() {
6        int temp;
7
8        transaction {
9          temp = value;
10         temp++;
11         value = temp;
12       }
13   }
14 }
```

Code 5.5: A transaction block protecting a shared variable, *value*.

that a method invoking multiple transaction methods must itself be a transaction method. This is because the rule does not apply to transaction blocks so all methods are allowed to contain multiple transaction blocks. Similarly, invocations of transaction methods inside a transaction block does not require the enclosing method to become a transaction method. That is, unless the method also contains multiple invocations of transaction methods which are not enclosed in transaction blocks.

5.2.4 Gates

This section describes the gates which are part of Cava. The section builds on the description provided in Section 5.1.5 on page 30.

Gates constitute a message passing mechanism which is encapsulated in objects. That is, gates are not applied in a program by using new keywords or other language constructs. The fundamental idea is that threads can send messages to gates. The message may carry a value in the form of some object but this is not required. Other threads, including the sender, can retrieve the messages from the gate which enables communication and co-operation between threads.

There are two different types of gates in Cava: gates with "or" semantics encapsulated in the OrGate class, and gates with "and" semantics encapsulated in the AndGate class. The two types of gates are described in Sections 5.2.4.3 and 5.2.4.4 on page 44. However, some of the constructs are common to the two types of gates and these are described in the following two sections.

5.2.4.1 The Gate Interface

Both types of gates implement the Gate interface which is seen in Table 5.4.

Method Summary:
public transaction abstract void signal() Sends an empty signal to the gate.
public transaction abstract void signal(T message) Sends a signal to the gate with a message attached.
public transaction abstract void tap(Gate gate) Connects to the gate to receive signals from the gate.

Table 5.4: The methods in the Gate interface.

The interface declares an overloaded signal method which is used to send a new signal to the gate. The last method, tap, is used to create gate networks (see Section 5.2.4.5 on page 46).

There are two signal methods since one is used to send signals without a message and the other is used to send a signal carrying a message. Each signal can only carry one message but there is no restriction on the type of messages. This is the case since the messages are handled in the form of objects and hence, any type of object can be sent through a gate. Both types of gates utilise the generic feature introduced in Java 5. This implies that when a gate is declared, the object type of the messages can be specified.

The signal methods on the Gate interface are transaction methods. This implies that only one thread has access to the gate at a time. If two threads invoke signal on the gate simultaneously, it is not possible to assume any order on which thread is allowed to actually execute signal before the other. This is the case since the operations are transactions which may interfere and interrupt each other. However, in case a single thread invokes signal several times, these messages arrive in order at the gate.

5.2.4.2 The Receiver Interface

The Receiver interface, which can be seen in Table 5.5, defines a receive method which is applied to retrieve signals from gates. If the signal carried a message, this becomes the return value af receive. Otherwise, the return value is null.

Method Summary:
public transaction T receive() Retrieves a signal from the gate. The method call is blocked if no signals are available.

Table 5.5: The method in the Receiver interface.

Both or-gates and and-gates use the Receiver interface. However, because of the difference in semantics in the two types of gates, the Receiver interface is used on different constructs which is addressed in Section 5.2.4.3 and Section 5.2.4.4.

5.2.4.3 The OrGate Class

The semantics of an or-gate is to forward a given message to one and only one recipient (i.e. thread). The OrGate class implements both the Gate and Receiver interfaces. Hence, the methods available on or-gates are the methods in Table 5.4 and Table 5.5.

As mentioned, a thread may send a signal, which may or may not carry an actual message, to an or-gate by invoking the signal method on the gate. A message can be retrieved from an or-gate by invoking the receive method on the same gate. A call to receive is matched against an invocation of signal, and the parameter of signal becomes available if a such was provided. If no signals are available when receive is invoked, the receiving thread is blocked until a signal arrives to the gate. The signal and receive methods are transaction methods.

The semantics of the or-gate implies that any thread may send a signal to an or-gate but only one thread receives a given signal by invoking receive on the gate. An example of this is seen in Code 5.6. Here, the instance of the SendThread class sends a signal to *gate* which the instance of the ReceptionThread class receives. Note, the run methods need not be transaction methods since they only invoke one transaction method each.

5.2.4.4 The AndGate Class

The semantics of an and-gate is to forward a given message to all recipients (i.e. threads) wishing to receive messages from the gate. The AndGate class implements the Gate interface but not the Receiver interface. Hence, the methods available on and-gates are the methods in Table 5.4 along with the subscribe method shown in Table 5.6.

Method Summary:
public transaction Receiver<T> subscribe() Returns a Receiver object which can be used to retrieve signals from the gate.

Table 5.6: The additional method in the AndGate class.

As mentioned, a thread may send a signal, which may or may not carry an actual message, to an and-gate by invoking the signal method on the gate. However, since the AndGate class does not implement the Receiver interface,

```
1  public class Example {
2
3    class ReceptionThread extends Thread {
4      Receiver receiver;
5
6      public ReceptionThread(Receiver receiver) {
7        this.receiver = receiver;
8      }
9
10     public void run() {
11       receiver.receive();
12     }
13   }
14
15   class SendThread extends Thread {
16     Gate gate;
17
18     public SendThread(Gate gate) {
19       this.gate = gate;
20     }
21
22     public void run() {
23       gate.signal();
24     }
25   }
26
27   public void execute() {
28     OrGate gate = new OrGate();
29     new ReceptionThread(gate);
30     new SendThread(gate);
31   }
32 }
```

Code 5.6: One thread sending a signal to another thread using an or-gate.

there is no receive method directly on and-gates. Instead, threads wishing to receive the signals have to invoke the subscribe method. This results in a Receiver object which handles the signals from the and-gate. Once a signal arrives at the and-gate, it is forwarded to the Receiver objects of all threads which have subscribed to the and-gate. The effect of this is that old messages are not retained in the and-gate. The receive method on the Receiver object has to be invoked to actually receive the signals from the gate. A call to receive is matched against an invocation of signal in the Receiver object, and the parameter of signal becomes available if a such was provided. If no signals are available when receive is invoked, the receiving thread is blocked until a signal arrives to the Receiver object.

The semantics of an and-gate implies that any thread may send a signal to an and-gate but only threads which have invoked subscribe on the gate receive the signal. A thread may subscribe to an and-gate at any time but it is not possible to unsubscribe to and-gates. If there are no subscribers to an and-gate when a signal is sent to it, the signal could be said to be lost. However, all threads subscribing to an and-gate receive all signals sent after they have subscribed to the gate.

An example using an and-gate is seen in Code 5.7. Here, the instance of the SendThread class sends a signal to *gate*. The instance of the ReceptionThread class receives the signal by invoking receive on the Receiver object, *receiver*, which was returned from the invocation of subscribe on *gate*. Note, it would be possible to have more threads subscribing to the and-gate.

5.2.4.5 Networks of Gates

Both types of gates may be linked together to form a gate network which is a decentralised construct formed by the individual gates which are connected into the network structure. That is, it is not manifested in a network structure object or any other representation, it is only present in the semantics of the gates. When gates are connected in a network, signals sent to the gates can propagate between the gates in the network. This implies that if a gate has been connected to another gate, a call to receive on the gate can also return a signal which was sent to the other gate.

A gate network is built by making connections between different gates. The connections are established by invoking the tap method on the gate which should receive signals from another gate. The other gate in the connection is defined by passing it as a parameter to the tap method. Invoking tap on a gate using itself as a parameter is ignored. Similarly, if the same connection is

```
1  public class Example {
2
3    class ReceptionThread extends Thread {
4      Receiver receiver;
5
6      public ReceptionThread(Receiver receiver) {
7        this.receiver = receiver;
8      }
9
10     public void run() {
11       receiver.receive();
12     }
13   }
14
15   class SendThread extends Thread {
16     Gate gate;
17
18     public SendThread(Gate gate) {
19       this.gate = gate;
20     }
21
22     public void run() {
23       gate.signal();
24     }
25   }
26
27   public void execute() {
28     AndGate gate = new AndGate();
29     new ReceptionThread(gate.subscribe());
30     new SendThread(gate);
31   }
32 }
```

Code 5.7: One thread sending a signal to another thread using an and-gate.

established multiple times. Note that the gates, which are being connected, can already be part of existing gate networks. Joining the gates together introduces a connection between the two gate networks, resulting in a larger gate network composed of the two previous networks. When two gates have been connected they cannot be disconnected again. There is no restriction on how gates can be connected which implies that any two gates can be linked together. Loops in the network are also allowed.

Gate connections are asymmetric, implying that signals only flow in one direction. That is, when two gates, a and b, are connected by $a.\mathsf{tap}(b)$, only a is able to receive signals from b. A signal sent to a cannot be retrieved through b. If this should also be the case, tap must also be called on b with a as the parameter.

The individual gate in a network preserves its semantics as either an or-gate or an and-gate which has two implications. The first is that the individual gate can still function as a stand-alone or-gate or and-gate when it is part of a gate network. This occurs if no calls are made to signal or receive on other gates in the network. The other implication is that signals are propagated in the network in accordance with the semantics of the individual gates. This implies that when a signal passes through an or-gate, only one of the gates tapping the gate receives the signal, and when a signal passes through an and-gate, all gates tapping the gate receive the signal. Note that when a signal passes through an and-gate, all subscribers receive a copy of the signal, including gates tapping the and-gate and all non-gate subscribers (i.e. threads). When a signal passes through an or-gate, it is passed either to a gate tapping the or-gate or a thread which has invoked receive directly on the gate.

The propagation of signals in a gate network is under lazy evaluation, implying that signals are passed between gates when a reception occurs on one of the gates, and not when a signal is sent to a gate. That is, when receive is invoked on one of the gates, this propagates through the network until it reaches a gate which contains a message. If there are no signals available in the network, the thread becomes blocked on the invocation of receive. Note, the propagation of a signal takes place immediately if a call to receive has been made somewhere in the gate network prior to the signal being sent.

5.2.5 Inheritance

This section presents points in relation to inheritance in Cava. The section builds on the description provided in Section 5.1.6 on page 31.

Inheritance in Cava is obtained using the extends keyword and only single inheritance is supported. If a class needs to have the functionality of several other classes, this can be obtained by utilising interfaces.

5.2.5.1 Overriding Methods

Transaction methods may be viewed as code which implements synchronisation between threads. Transaction methods in Cava are visible on inheritance which implies that the transactional status of methods is part of the interface of a class. Hence, when overriding a method, the programmer is able to see whether or not it is a transaction method. This is necessary since transaction methods should be overridden with transaction methods and conventional methods should be overridden with conventional methods.

If a method in a super-class is a transaction method, it is likely because it accesses some shared variables. The method in the subclass may not necessarily need to access shared variables but in order to comply with the interface of the method in the super-class, the overriding method should also be a transaction method. If the overriding method does not access any shared variables, the transaction keyword is without effect.

If a method in a super-class is a conventional method, it cannot be upgraded to a transaction method in the subclass. This is the case since it could cause violations of the rule that a method has to be a transaction method if it invokes multiple transaction methods. That is, a method, which invokes a transaction method as well as the overriding method in question, would have to be upgraded as well. In case the overriding method needs to access shared variables, this can be handled within transaction blocks.

Transaction blocks within methods also implement synchronisation between threads. However, this is not visible on inheritance since it is not part of the interface of the method. This does not pose a problem since any shared variables accessed within the overridden transaction blocks cannot be changed in the new method without applying a transaction block. Therefore, race conditions cannot be introduced by overriding a method containing transaction blocks.

Chapter 6

Application of Cava

This chapter contains various applications of Cava. Examples of how classical concurrency constructs can be modelled in Cava are included in Sections 6.1–6.2. Furthermore, extracts are presented from implementations of Dining Philosophers in Section 6.3 on page 53 and the Santa Claus problem in Section 6.4 on page 58. Finally, Cava is applied to model a parallel version of the Quicksort algorithm in Section 6.5 on page 62. Note, all examples presented in this chapter can be found in their entirety on the enclosed CD-ROM.

6.1 Lock

This section illustrates how a lock can be modelled in Cava. Code 6.1 shows a Lock class with an or-gate, *gate*, as the only instance variable. It is shared since multiple threads should be able to access it when using the lock.

As part of the Lock constructor, a signal is sent to *gate*. This ensures that the first thread, which invokes the acquire method, is allowed to continue its execution since a signal is available in *gate* when receive is invoked. Any thread which subsequently invokes acquire is blocked since there is no signal available on *gate*. Once the current thread invokes the release method, a signal is sent to *gate* allowing another thread to obtain the lock since it becomes unblocked on the receive method.

Code 6.1 shows that a lock is very easily modelled in Cava. The functionality of or-gates makes it straightforward to block a thread when it invokes acquire on a Lock object which is not available.

```
3  public class Lock {
4
5    private shared OrGate gate = new OrGate();
6
7    public Lock() {
8      transaction {
9        gate.signal();
10     }
11   }
12
13   public transaction void acquire() {
14     gate.receive();
15   }
16
17   public transaction void release() {
18     gate.signal();
19   }
20 }
```

Code 6.1: Modelling a lock in Cava.

6.2 Emulating wait, notify, and notifyAll

This section describes how the Java methods wait, notify, and notifyAll can be emulated in Cava.

Code 6.2 on the next page shows a WaitNotify class with an or-gate, *gate*, and an integer, *waitingThreads*, as instance variables. The latter registers how many threads have invoked halt. Both variables are shared since multiple threads should be able to access them. The halt method is the equivalent of wait, while proceed and proceedAll are the equivalents of notify and notifyAll.

When a thread invokes halt, the *waitingThreads* variable is incremented to indicate that there are threads which are suspended on *gate*. Furthermore, *waitingThreads* registers exactly how many threads are blocked on a call to halt which is needed in the proceedAll method. The *gate* object is replaced with a new or-gate as part of the proceed and proceedAll methods which is addressed later in this section. Hence, the or-gate, which is available when *waitingThreads* is incremented, is saved in *waitGate*. The next step is to actually suspend the thread by invoking receive on *waitGate*. The two steps are surrounded by separate transaction blocks to ensure that the invocation

```
3  public class WaitNotify {
4
5    private shared int waitingThreads = 0;
6    private shared OrGate gate = new OrGate();
7
8    public void halt() {
9      OrGate waitGate = null;
10
11     transaction {
12       waitingThreads++;
13       waitGate = gate;
14     }
15
16     transaction {
17       waitGate.receive();
18     }
19   }
20
21   public transaction void proceed() {
22     OrGate signalGate;
23
24     if (waitingThreads > 0) {
25       gate.signal();
26       waitingThreads--;
27
28       signalGate = gate;
29       gate = new OrGate();
30
31       if (waitingThreads > 0) {
32         signalGate.tap(gate);
33       }
34     }
35   }
36
37   public transaction void proceedAll() {
38     while (waitingThreads > 0) {
39       gate.signal();
40       waitingThreads--;
41     }
42
43     gate = new OrGate();
44   }
45 }
```

Code 6.2: Emulating wait, notify, and notifyAll.

of halt is registered in *waitingThreads* before the thread is suspended on *waitGate*.

An invocation of proceed sends a single signal to *gate* if *waitingThreads* is positive and decrements *waitingThreads*. The signal is received by a single thread blocked on receive on the given gate and thus emulates notify which only wakes up a single thread. The fact that a signal is only sent if *waitingThreads* is positive emulates the possibility of losing notifications in Java in case notify or notifyAll is invoked before wait. Hence, if no threads have invoked halt, a signal should not be sent to *gate* since this would be available in a subsequent call to halt. As part of the method, a new or-gate object is assigned to *gate* in line 29. This is done to ensure that the signal just sent only can be received by already waiting threads and not by new threads invoking halt. However, if multiple threads are waiting on the previous gate, these should also be able to receive signals from future invocation of proceed. This semantics is achieved by constructing a gate network in line 32 consisting of the old gate and the new one.

The proceedAll method also emulates the possibility of losing notifications since signals are only sent to *gate* if *waitingThreads* is positive. The proceedAll method sends a number of signals to *gate* which matches the number of threads which have invoked halt at the time proceedAll is invoked. The correct number of signals is ensured since proceedAll is a transaction method which implies that any following calls to halt are delayed until proceedAll has finished its execution. Because these latter threads should not be able to receive one of the current signals on the *gate*, a new or-gate is assigned to the *gate* variable in line 43.

Code 6.2 shows that an emulation of wait, notify, and notifyAll can be obtained in Cava. As was the case with the Lock class in Section 6.1, the functionality of or-gates makes it straightforward to block a thread when it invokes halt on a WaitNotify object. Furthermore, the semantics of transaction methods implies that inference in the proceedAll method is easily prevented. However, in order to obtain the correct semantics, the *gate* variable must be manipulated and gate networks are required. This is also easily obtained since the manipulations are protected using transactions.

6.3 Dining Philosophers

Dining Philosophers is a classical concurrent problem and this section presents how it can be modelled in Cava. Fundamentally, the allocation of forks

between philosophers can be viewed as either competition or co-operation. Both scenarios are included in the following sections.

6.3.1 Competition

This section describes the Dining Philosophers problem when the philosophers compete for the forks. Code 6.3 shows the constructor declaration of the Philosopher class and it is seen that his forks are parameters to this. This is the case since the philosophers have to share the forks so these have to be instantiated outside the Philosopher class.

```
9   public Philosopher(int id, Fork leftFork, Fork rightFork) {
```

Code 6.3: The constructor declaration of the Philosopher class.

The run method of the Philosopher class is seen in Code 6.4. It simply shows that a philosopher alternates between thinking and eating. The think method consists only of printing a message and a sleep period and is not included.

```
16  public void run() {
17    while (true) {
18      think();
19      eat();
20    }
21  }
```

Code 6.4: The run method of the Philosopher class.

Code 6.5 shows the eat method. This is a transaction method since the forksUp and forksDown methods are transaction methods. Hence, the rule, that a method invoking multiple transaction methods should be a transaction method, is applied. Moreover, it ensures that a philosopher is allowed to pick up his forks, eat, and put down his forks without interleaving with other philosophers.

The forksUp method is seen in Code 6.6 and it shows that a pickUp method is invoked on each fork. It is marked as a transaction method since pickUp is a transaction method. Furthermore, it ensures that a philosopher picks

```
28  private transaction void eat() {
29    forksUp();
30    System.out.println(this + " took " + leftFork + " and " + rightFork);
31    System.out.println(this + " eating");
32    Thread.sleep(200);
33    forksDown();
34    System.out.println(this + " put down " + leftFork + " and " + rightFork);
35  }
```

Code 6.5: The eat method of the Philosopher class.

```
37  private transaction void forksUp() {
38    leftFork.pickUp();
39    rightFork.pickUp();
40  }
```

Code 6.6: The forksUp method of the Philosopher class.

up both forks in one go. The forksDown method only differs from forksUp in that a putDown method is invoked on the forks and is not included.

Code 6.7 on the following page shows the Fork class. It contains a Boolean instance variable, *taken* which indicates whether the fork is presently being used. The pickUp and putDown methods change the status of the fork and they are transaction methods since *taken* is a shared variable.

That two philosophers cannot pick up the same fork is ensured by the shared variable *taken*. This is the case since a philosopher which has invoked pickUp on a fork accesses *taken*. Hence, if another philosopher invokes pickUp on the same fork, the forksUp method of that philosopher will experience a conflict and not be allowed to access the fork until the first philosopher's eat method, which is a transaction, has finished its execution or is aborted.

The above shows that the STM semantics of transaction methods makes it easy to prevent two philosophers from picking up the same fork. Furthermore, a philosopher is allowed to pick up his forks, eat, and put down his forks uninterrupted simply because eat is a transaction method. It is noted that the implementation is deadlock-free due to the STM semantics of transaction methods. However, it is not free of the possibility of starvation.

The implementation of Dining Philosophers appears more elegant than an equivalent one made in Java. This is based on the fact that the main focus can

```
3  public class Fork {
4
5    private int id;
6    private shared boolean taken;
7
8    public Fork (int id) {
9      this.id = id;
10     transaction {
11       taken = false;
12     }
13   }
14
15   public transaction void pickUp() {
16     taken = true;
17   }
18
19   public transaction void putDown() {
20     taken = false;
21   }
```

Code 6.7: The Fork class.

be on implementing the functionality and less effort is put into the required synchronisation since this is obtained using transaction methods. Moreover, this also makes the code more readable since the synchronisation does not have to be made explicit.

6.3.2 Co-operation

This section describes the Dining Philosophers problem when the philosophers co-operate about the forks. Code 6.8 shows the Fork class. It contains an or-gate instance variable, *gate*, which is used to obtain co-operation between the philosophers. The pickUp method simply invokes receive on *gate* which implies that a philosopher is only allowed to pick up the fork in case there is a signal. If this is not the case, the philosopher is blocked on receive until a signal becomes available. This is provided in the putDown method. Note, the first signal to *gate* is sent as part of the Fork constructor.

The run method of the Philosopher class is seen in Code 6.9. It shows that a philosopher repeats thinking, picking up his forks, eating, and putting down

```
3  public class Fork {
4
5    private int id;
6    private OrGate gate = new OrGate();
7
8    public Fork (int id) {
9      this.id = id;
10     gate.signal();
11   }
12
13   public void pickUp() {
14     gate.receive();
15   }
16
17   public void putDown() {
18     gate.signal();
19   }
```

Code 6.8: The Fork class.

his forks. Similarly to the competing philosophers, the think method only consists of printing a message and a sleep period and is not included. The same is the case for the eat method.

```
16   public void run() {
17     while (true) {
18       think();
19       forksUp();
20       eat();
21       forksDown();
22     }
23   }
```

Code 6.9: The run method of the Philosopher class.

Code 6.10 on the next page shows the forksUp method which is invoked in the run method. It shows that pickUp is invoked on each fork which effectively corresponds to invoking receive on the or-gates in the forks. The invocations of pickUp are enclosed in a transaction block to ensure that the philosopher is granted both forks collectively. This prevents the deadlock scenario where

each philosopher picks up e.g. his left fork and waits for access to his right fork. This would never be granted since all philosophers are blocked on a call to receive on his right fork. Again, the forksDown method only differs in that putDown is invoked on the forks. Furthermore, there is no need for a transaction block around the invocations of putDown since they correspond to invoking the non-blocking signal method on the or-gates in the forks.

```
35  private void forksUp() {
36    transaction {
37      leftFork.pickUp();
38      rightFork.pickUp();
39    }
40
41    System.out.println(this + " took " + leftFork + " and " + rightFork);
42  }
```

Code 6.10: The forksUp method of the Philosopher class.

The above shows that the functionality of or-gates makes it straightforward to block a philosopher when he tries to pick up a fork which is already in use. The co-operation is obtained solely by the transaction block in the forksUp method in Code 6.10 and the functionality of or-gates. Furthermore, the transaction block in the forksUp method prevents the philosophers from introducing a circular dependency. Hence, similarly to the competing philosophers, the implementation is deadlock-free. However, it is again possible to witness starvation.

6.4 The Santa Claus Problem

This section contains extracts from an implementation of the Santa Claus problem in Cava. A description of the problem can be found in Appendix B on page 117.

Code 6.11 shows part of the code which initialises the Santa Claus problem. Note, the Santa, Pixy, and Reindeer classes all extend the Thread class.

Lines 16 and 19 in Code 6.11 illustrate that there is a circle of dependency between the Santa and Barn classes since they appear as variables in each other. This is initialised by invoking setSanta on *barn* and invoking setBarn on *santa*. The invocation of setBarn on *santa* has to be made before the

```
8   public static void main(String[] args) {
9     System.out.println("System starting");
10
11    Barn barn = new Barn();
12    Shop shop = new Shop();
13    Sleigh  sleigh  = new Sleigh();
14    Santa santa = new Santa();
15
16    barn.setSanta(santa);
17    shop.setSanta(santa);
18
19    santa.setBarn(barn);
20    santa.setSleigh(sleigh);
21
22    santa.start();
```

Code 6.11: Part of the initialisation of the Santa Claus problem.

Santa thread is allowed to start its execution. However, threads are started as part as their constructors so a delayed start, similarly to that of Java, has to be implemented. This is observed in line 22 where an explicit start method is invoked on *santa*.

Code 6.12 shows how the delayed start is obtained by letting the very first operation in the run method of the Santa class be an invocation of receive on an or-gate, *waitForStart*. This blocks the Santa thread until a signal arrives on the or-gate.

```
27  public void run() {
28    waitForStart.receive();
```

Code 6.12: The first operation in the run method of the Santa class.

The start method, which is seen in Code 6.13 on the next page, then simply sends this signal to *waitForStart* which effectively starts the actual execution of the Santa thread. It is noted that a similar construction is applied to the Pixy and Reindeer classes which need to know the Shop and Barn objects, respectively.

Part of Santa's functionality is to wait for the reindeer and pixies. This is implemented using an or-gate, *santaSleeping*, on which receive is invoked,

```
106   public void start() {
107     waitForStart.signal();
108   }
```

Code 6.13: The start method of the Santa class.

blocking the Santa thread. The signal to *santaSleeping* is sent as part of the wakeUp method which is invoked by the last reindeer or a group of pixies with problems. This method appears in Code 6.14.

```
110   public transaction void wakeup(Object source) {
111     if (source instanceof Reindeer) {
112       reindeersReady = true;
113     } else {
114       pixies = (List<Pixy>) source;
115     }
116     santaSleeping.signal();
117   }
```

Code 6.14: The wakeUp method of the Santa class.

The run method in the Reindeer class is seen in Code 6.15. It shows that the behaviour of a reindeer is implemented using two or-gates: *reindeerWait* and *santaWait*. Two gates are needed because the Santa thread must wait for an acknowledgement from the Reindeer thread when it has completed harnessing or unharnessing. In order to ensure that the right thread receives the signal, the reindeer and Santa cannot wait on the same or-gate.

Once a reindeer has arrived at the barn in lines 29–32, it awaits harnessing by invoking receive on *reindeerWait*. The matching invocation of signal on *reindeerWait* is provided by the harness method which is invoked by Santa and is seen in Code 6.16. The harnessing is completed when the reindeer signals an acknowledgment to Santa on *santaWait*. Unharnessing is achieved by a similar synchronisation pattern, and is therefore not included. Furthermore, a similar construction is used when pixies need to go to Santa's shop to get their problems fixed.

Code 6.17 on page 62 shows the arrival method of the Barn class which is invoked by a reindeer in line 31 of the run method in the Reindeer class (see Code 6.15). When a reindeer arrives at the barn, it is added to a list of

```
21  public void run() {
22    waitForStart.receive();
23
24    while (true) {
25      System.out.println(this + " going on holiday");
26
27      Thread.sleep(1000);
28
29      transaction {
30        System.out.println(this + " back for Christmas");
31        barn.arrival(this);
32      }
33
34      transaction {
35        reindeerWait.receive();
36        System.out.println(this + " being harnessed");
37        santaWait.signal();
38      }
39
40      transaction {
41        reindeerWait.receive();
42        System.out.println(this + " being unharnessed");
43        santaWait.signal();
44      }
45    }
46  }
```

Code 6.15: The run method of the Reindeer class.

```
52  public void harness() {
53    transaction {
54      reindeerWait.signal();
55    }
56
57    transaction {
58      santaWait.receive();
59    }
60  }
```

Code 6.16: The harness method of the Reindeer class.

reindeer, *reindeers*. Following this, it is checked whether the reindeer was the last one to arrive. If this is the case, it wakes up Santa in line 20 by invoking the wakeup method which is seen in Code 6.14 on page 60.

```
16   public transaction void arrival(Reindeer reindeer) {
17     reindeers.add(reindeer);
18
19     if (reindeers.size() == 9) {
20       santa.wakeup(reindeer);
21     }
22   }
```

Code 6.17: The arrival method of the Barn class.

The functionality of or-gates enables implementing a very fine-grained, yet simple, management of the behaviour of the actors in the Santa Claus problem. The possibility of modelling a delayed start of a thread, similar to that of Java, using an or-gate also illustrates the Cava's expressiveness.

6.5 Quicksort

This section contains extracts from a parallel implementation of the Quicksort algorithm in Cava. The description of the algorithm is based on [C+01]. Quicksort performs the sorting of an array in place using a divide-and-conquer strategy. This makes it well-suited for introducing parallelism which is addressed later.

The three steps of the divide-and-conquer process applied to a subarray, $A[p\,..\,r]$ is shown below [C+01, p. 145]:

Divide: Partition $A[p\,..\,r]$ into two subarrays $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$. These subarrays fulfil that the elements in $A[p..q-1]$ are less than or equal to $A[q]$ which, in turn, is less than or equal to the elements in $A[q+1\,..\,r]$. The value of the index q is returned as part of the partitioning.

Conquer: Sort the subarrays $A[p..q-1]$ and $A[q+1..r]$ by making recursive calls to the Quicksort algorithm.

Combine: Since the algorithm sorts in place, there is no need for combining the subarrays.

Code 6.18 shows the instance variable, *data*, and sort method of the QuickSort class. The array, *data*, is marked as shared since several threads are going to perform the sorting concurrently. The sort method utilises an inner class SortThread and an or-gate, *gate*, to block the current thread until the SortThread has finished its execution.

```
5  public class QuickSort {
6
7    private shared int[] data;
8
9    public int[] sort(int[] input) {
10     OrGate gate = new OrGate();
11     int[] output;
12
13     transaction {
14       data = input;
15     }
16
17     new SortThread(0, input.length - 1, gate);
18
19     transaction {
20       gate.receive();
21       output = data;
22     }
23
24     return output;
25   }
```

Code 6.18: The instance variable, *data*, and sort method of the QuickSort class.

The constructor of the inner SortThread class is seen in Code 6.19 on the following page. An instance of SortThread saves the gate which was provided by its creating thread since it needs to signal this gate when it has finished its execution. It is the run method of the SortThread class which actually implements the divide-and-conquer strategy of Quicksort. This method is seen in Code 6.20 on the next page.

The run method follows the steps which were described above closely by calculating the dividing index, q, and creating two new SortThreads to handle the subarrays. The current SortThread is blocked until the two threads have finished their part of the sorting by invoking receive twice on *gate* which

```
27    private class SortThread extends Thread {
28      OrGate parentGate;
29      int p;
30      int r;
31
32      public SortThread(int p, int r, OrGate gate) {
33        this.p = p;
34        this.r = r;
35        parentGate = gate;
36      }
```

Code 6.19: The constructor of the inner SortThread class in the QuickSort class.

```
38      public void run() {
39        if (p < r) {
40          OrGate gate = new OrGate();
41
42          int q = partition(p, r);
43
44          new SortThread(p, q - 1, gate);
45          new SortThread(q + 1, r, gate);
46
47          transaction {
48            gate.receive();
49            gate.receive();
50          }
51        }
52
53        transaction {
54          parentGate.signal();
55        }
56      }
```

Code 6.20: The run method of the SortThread class.

was sent as a parameter to the threads. Once the signals from the threads have been received, the current thread completes its execution by signalling *parentGate*.

The partition method performs the actual sorting. This method is seen in Code 6.21. It is a transaction method since it manipulates the shared variable, *data*. The exchange method is trivial and not included.

```
58    private transaction int partition(int p, int r) {
59      int x = data[r];
60      int i = p - 1;
61
62      for (int j = p; j < r; j++) {
63        if (data[j] <= x) {
64          i++;
65          exchange(i, j);
66        }
67      }
68
69      exchange(i + 1, r);
70
71      return i + 1;
72    }
```

Code 6.21: The partition method of the SortThread class.

The above shows that parallelism is easily introduced in the implementation of Quicksort. The fact that the array is marked as shared allows for multiple threads to operate on it concurrently. The synchronisation between the threads is also easily obtained using or-gates.

Chapter 7

Implementation of Cava

As mentioned in Section 1.3.2 on page 3, developing a new concurrency model remains a theoretical exercise if the new language constructs cannot be tested and evaluated by applying them to concurrent problems. This chapter describes a partial and experimental implementation of Cava which was developed as part of the project. Requirements for an implementation are described in Section 7.1 along with the overall structure of the implementation. Section 7.2 on page 69 and Section 7.3 on page 74 describe the implemented compiler and runtime system, respectively. In the remainder of the report, the compiler is referred to as the Cava compiler and the runtime system as the Cava Runtime System.

7.1 Specification

This section contains a specification for the implementation of Cava. That is, the goals and requirements for the implementation along with the selection of which features to implement. Finally, the overall structure of the implementation is presented.

7.1.1 Implementation Goals and Requirements

The experimental implementation of Cava has two goals: ensuring that the Cava concurrency constructs are indeed implementable and creating a platform for experiments with applying Cava. The first goal is important if the model should not remain a purely theoretical entity. The second goal creates

the opportunity of writing executable code in Cava which renders reliable insight into how the concurrency model is to work with for a programmer. This is important since the concurrency constructs may appear sound in theory but may not be so in practical usage.

Evidently, having a full implementation of Cava is preferable but in order to fulfil the goals, only part of the concurrency model needs to be implemented. That is, only the features, which are especially interesting in relation to experiments, are covered (see Section 7.1.2). A positive effect of the goals is that it induces flexibility in the design of the implementation. This includes that it does not have to consist of production quality code and it does not have to be optimised. This implies that simpler designs can be applied. Furthermore, it makes the entire process of ensuring that the Cava concurrency model is implemented correctly more reliable.

7.1.2 Implemented Features

As mentioned above only a subset of the features in Cava are selected for implementation. These are selected based on a principle of implementing features which are required to execute examples of applying Cava, including the examples found in Chapter 6 on page 50.

The implemented subset consists of: non-deterministic threads, thread local variables, shared variables, transaction methods and blocks, the Gate and Receiver interfaces, and the OrGate and AndGate classes. Hence, the following parts are not implemented in the experimental system: deterministic threads and gate networks.

Some of Cava's concurrency concepts are not new inventions. They may have appeared in other systems or contexts and thus have been implemented before. However, they may not have been implemented in the combination, and with the modifications, that they appear in Cava. An example of this is the STM semantics which e.g. has been implemented in the DSTM2 framework (see Section 10.2 on page 101).

The implementation is experimental in relation to which three points can be made. The first is that static constraint checks are not implemented. Examples of this are whether shared variables are accessed outside of transactions, whether a conventional method invokes multiple transaction methods, and whether the transactional status of methods is preserved on inheritance. This implies that it becomes the programmer's responsibility to ensure that Cava programs comply with these rules.

The second point is that the implemented features are not systematically tested. That is, the constructs are confirmed to work on representative examples but the system may not recognise all usages even though these are correct according to the Cava syntax and semantics found in Chapter 5 on page 28.

The last point is that the implementation has not been optimised. However, an optimisation would not impact the semantics of the various constructs and it is therefore not considered important when it comes to gaining insight into how the concurrency model is to work with for the programmer. This choice also implies that the implementation cannot be used to compare the performance of Cava to that of Java. If such a comparison should be performed, the Cava implementation should first be optimised.

7.1.3 Overall Design

Since Cava is built on top of Java, an implementation of Cava can be made by modifying an existing Java implementation of which several are available. The basic design behind Java implementations is that of a compiler which translates Java code into Java bytecode, i.e. class files, which can then be executed on the target computer on a JVM. An adapted version of this strategy is applied when implementing Cava. The Cava implementation consists of two parts: a compiler and a runtime system.

The Cava compiler compiles Cava code into Java bytecode. This is achieved by hijacking a Java compiler and modifying the Cava code into equivalent Java code before the class files are generated. This way, the compiled Cava code can be executed on a standard JVM. However, some Cava constructs are not easily translated into Java code. This implies that in many cases, the Cava code is translated into Java code which utilises features implemented in a special Cava Runtime System (CRS). Hence, the CRS must be installed in the JVM before the translated Cava classes can be executed. The CRS is designed as a number of class files which the compiler imports into the generated class files by default. The CRS is compiled on a standard Java compiler and packed into a JAR file. This JAR file should then be specified on the JVM Classpath, thereby making the CRS available for the execution of Cava code.

7.2 Cava Compiler

This section contains a description of the Cava compiler and details on how it is implemented.

7.2.1 Sun's Java Compiler

As mentioned, an obvious base point for a Cava compiler is an existing Java compiler. There exist several implementations of the Java programming language, and the compiler chosen in this project is Sun's Java compiler. Early 2007, Sun released a copy of its Java compiler under a project called OpenJDK [Ope]. The compiler is still under development but stable releases are provided roughly once a month. The Cava compiler is based on the b07 version which was released on 1 February, 2007. The Java compiler is written in Java and is compiled using a standard Java compiler.

There are several reasons why the Sun compiler was chosen as a base point. It is a full implementation of a Java compiler and since it is widely used, it is also very stable. The benefit of this is that it is not necessary to implement existing Java constructs. Another reason is that the compiler directly creates class files. This is an advantage since the Cava compiler thereby also can produce class files and not just java files which must then be compiled with a Java compiler.

7.2.2 Structure of the Java Compiler

In general, the Java compiler is reasonably structured and applies object-oriented principles to create a division of components. However, the compiler is a fairly large piece of software since it contains 20 packages and 251 classes.

The Java compiler has a logical structure which is comparable to a standard compiler layout. The basic steps are: an abstract syntax tree (AST) is built by parsing the Java source files, the AST is annotated and used for verification of the code, and the compiler generates class files based on the AST. All this resembles the normal layout of a compiler. However, in the end, the Java compiler is more complex since a number of other steps are added. These steps involve adding implicit code like default constructors, type checking the AST, making dataflow analyses, and desugaring Java constructs. Each step is defined in a separate class which induces encapsulation and prevents interference between the steps.

7.2.3 Compiler Modifications

In order to turn the Java compiler into a Cava compiler, a strategy is applied of hijacking the former just after the source files are parsed into an AST as shown in Figure 7.1. The AST is then traversed by a modifier component such that Cava specific constructs are replaced with equivalent Java code. The modified AST can then follow the normal flow of the Java compiler. That is, there is no need to change the most complex parts of the compiler such as the code generation system which would also require changes in the JVM. Another positive side of the strategy is that the compiler can check whether the code modifications are indeed valid Java constructs. This ensures that the code generated by the compiler is valid and executional. The negative side of the strategy is that errors reported by the compiler are on the modified code, implying that error messages are more difficult to understand for the compiler user.

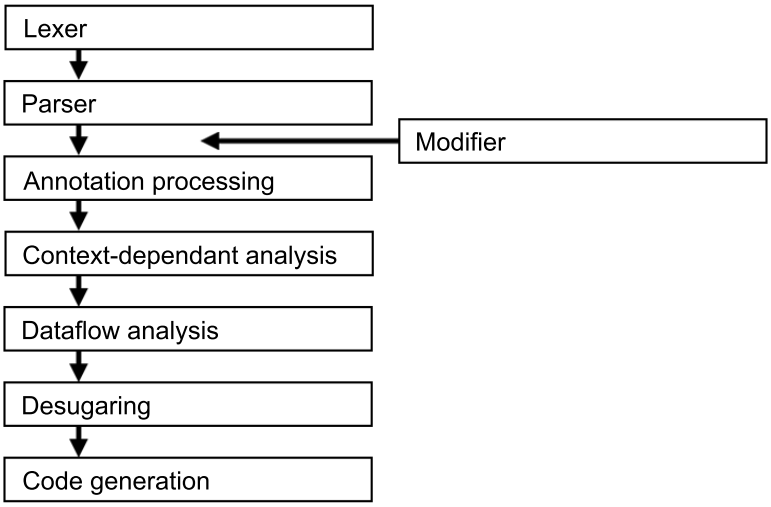

Figure 7.1: Component structure of the Cava compiler.

Because of the structure of the compiler, the strategy is easily adapted at component level. The parser component returns the AST which can then be directed into a Modifier object which returns the modified AST. The functionality of the modifier component is to traverse the AST and replace all Cava constructs with equivalent Java code. The Java code is generated as ASTs and inserted in the overall AST structure.

The implementation of the modifier component is less straightforward. The translation process from Cava to Java can be divided into two scenarios: simple Cava elements are identified and translated into predefined closed code

segments, or the surrounding code must be taken into consideration to generate a correct code segment. In the latter scenario, the modifier component must identify the Cava construct along with the context in which it is applied. This implies that more scenarios must be embraced in the translation which adds to the complexity of the modifier component. However, because of the overall design of having both a compiler and a runtime system, the amount of code inserted into the AST can be limited since it can utilise the CRS.

7.2.4 Translation Examples

This section presents two examples of how Cava code is translated into Java code in the modifier component. The purpose of this is to illustrate the translation of basic Cava constructs since implementing these on top of Java is a non-trivial process. Note that the translation occurs according to the design of the CRS which is described in Section 7.3 on page 74.

7.2.4.1 Translation of Variables

The first example shows how local and shared variables are translated. The Cava example to be translated appears in Code 7.1 on the next page. The example has two variables: a thread local variable, *localVariable*, and a shared variable, *sharedVariable*. The resulting Java code from the translation is shown in Code 7.2 on the following page.

The modifier component identifies the declaration of variables one at a time as part of a traversal of the AST. During this, the traversal can be interrupted and replacements of code can be performed. In Code 7.1, the modifier component applies the same processing when meeting the *localVariable* and *sharedVariable* declarations. The strategy is to encapsulate the variable in an object called a MemField which appears in two different forms: LocalMemField and SharedMemField. Which one to apply is determined by whether the variable is thread local or shared. Hence, the modifier component looks for the shared access modifier to identify which of the two MemFields to apply.

Encapsulating variables in MemFields changes the way they should be accessed. Two methods, get and set, are defined on MemField to allow reading and changing the value of a variable. The compiler has to identify the usage of the variables and replace them with calls to the get and set methods. In the example code, this is shown in the increment method where *localVariable* is read, incremented by one, and saved back to the variable. The read and write operations can be translated individually by the compiler.

```
1 int localVariable = 2;
2 shared int sharedVariable = 4;
3
4 public void increment() {
5   localVariable = localVariable + 1;
6 }
```

Code 7.1: Two class variables defined in Cava. A thread local variable, *localVariable*, and a shared variable, *sharedVariable*.

```
1 MemField localVariable = new LocalMemField(2);
2 MemField sharedVariable = new SharedMemField(4);
3
4 public void increment() {
5   localVariable.set((Integer)localVariable.get() + 1);
6 }
```

Code 7.2: Java code produced by the compiler when translating the Cava code from Code 7.1.

7.2.4.2 Translation of Transactions

The second example shows how a transaction method and a transaction block are translated. The Cava example to be translated appears in Code 7.3. The example includes a transaction method, increment, and a method containing a transaction block, incrementInBlock. The two methods both perform the same operation on *sharedVariable*. The resulting Java code from the translation is shown in Code 7.4.

In general, the translation process of transactions is fairly simple. When processing the method declaration in the Cava code, the modifier component identifies that the method is a transaction method. It then removes the transaction keyword from the access modifiers and encapsulates the entire method statement block in a large transactional Java construct. When the modifier component identifies a transaction block inside a method similar action is taken. A large Java construct is required since some of the operations needed to make Java execute transactionally cannot be implemented in the CRS.

The first part of the transactional construct is a do-while loop which allows the CRS to execute the transaction multiple times. Each time the loop is

```
1  public transaction void increment() {
2    sharedVariable = sharedVariable + 1;
3  }
4
5  public void incrementInBlock() {
6    transaction {
7      sharedVariable = sharedVariable + 1;
8    }
9  }
```

Code 7.3: A transaction method, increment, and a method containing a transaction block, incrementInBlock.

```
1  public void increment() {
2      Transaction trans;
3      do {
4          trans = new Transaction();
5          try {
6              sharedVariable.set((Integer)sharedVariable.get() + 1);
7          } catch (TransactionAbortException e) {
8          }
9      } while (!trans.commit());
10 }
11
12 public void incrementInBlock() {
13     {
14         Transaction trans1;
15         do {
16             trans1 = new Transaction();
17             try {
18                 sharedVariable.set((Integer)sharedVariable.get() + 1);
19             } catch (TransactionAbortException e1) {
20             }
21         } while (!trans1.commit());
22     }
23 }
```

Code 7.4: Java code produced by the compiler when translating the Cava code from Code 7.3.

executed, a new Transaction object is created e.g. in line 4 of Code 7.4. This object is utilised by the CRS to identify the transaction and to enable roll back and commit operations. A new Transaction object must be created each time since the CRS does not allow Transaction objects to be reused.

The inner part of the transaction is a try-catch block. Semantically, this block is not necessary but it increases the performance of the system since a transaction can be aborted at any time. The corresponding thread eventually detects the change in the state of the transaction and throws a TransactionAbortException. This is caught at the end of the transaction code block, effectively short-circuiting the execution of the transaction.

When the entire transaction code block has been executed, a call is made on the Transaction object requesting a commit of the transaction. If this commit fails, the do loop re-executes the transaction. If it succeeds, the thread continues by executing the code following the transaction.

In both cases in Code 7.3, the translation of the transactions does not require any major changes inside the transactional construct. However, if e.g. a return statement is present in the code, this should be replaced with a special transactional return construct.

7.3 Cava Runtime System

This section contains a description of the CRS and details on how it is implemented.

In order to execute Cava program, two things are needed: a JVM and the CRS. The CRS contains a number of class files which collectively create the platform against which Cava programs are compiled. The CRS is packed into a JAR file which must be defined as part of the JVM Classpath, otherwise Cava programs are not able to run. The CRS is compiled using a Java compiler.

7.3.1 Library Structure

The CRS consists of two different parts: the Cava System Library (CSL) and a class library which are described in the following two sections. The two parts serve different purposes and apply different implementation strategies since the CSL is implemented in Java whereas the class library is implemented in Cava.

7.3.2 Cava Class Library

The Cava class library is the equivalent of the API of the Java Runtime Environment, which is also called the standard class library. The Java standard class library contains a large number of components, data structures, and other kinds of partial programs with everything from linked lists to GUI components. The Cava class library is not of the same structure nor size but Cava requires its own standard library due to how the CSL system works. This is the case since e.g. data structures have to be under STM semantics which they are not in the Java class library.

In its current form, the Cava class library consists only of a very basic set of elements. The most noticeable part is the cava.lang package. The contents of the package is seen in Table 7.1. The System class contains a *out* variable as known from Java. The InputSerialiser and OutputSerialiser classes are used to make otherwise sequential processes work in a concurrent program (see Section 7.3.2.1 for details).

Interfaces:	Gate, Receiver, Runnable
Classes:	AndGate, OrGate, InputSerialiser, OutputSerialiser, Thread, System, InputStream, PrintStream

Table 7.1: The contents of the cava.lang package.

The contents of the cava.util package is seen in Table 7.2. The interface and classes all have the functionality indicated by their names.

Interfaces:	List
Classes:	ArrayList, LinkedList, Random

Table 7.2: The contents of the cava.util package.

7.3.2.1 InputSerialiser and OutputSerialiser

The purpose of the two classes is to provide a standard way of converting components which execute sequentially into executing concurrently. The classes are abstract so that instances cannot be made directly of the classes.

The general concept is to let a single thread handle and manage the sequential component. The functionality of the thread is to wait for signals on an

internal gate and perform some operation, defined in the message of a signal, on the sequential component. This way only one operation is performed at a time on the component, even though multiple threads can request operations on the component concurrently. It is the responsibility of the programmer to ensure that the concurrent operations are performed correctly since the InputSerialiser and OutputSerialiser classes only enable concurrent access.

An example of the application of the OutputSerialiser class appears in the variable System.*out*. Messages on the screen must be written sequentially such that printing from different threads does not interleave. If a thread has to print a large block of text, the thread can simply perform the printing in a single transaction.

7.3.3 Cava System Library

The CSL consists of the cava.system package which is implemented in Java and forms the CRS. Even though the package is in fact a Java library, it must not be called directly by the programmer. Instead, the classes and methods should only be used in the code generated by the Cava compiler.

The package contains a total of 24 classes and interfaces which can be divided into a few categories. The first is classes used directly by the Java code generated by the Cava compiler. These classes are seen in Table 7.3.

Classes:	MemField, LocalMemField, SharedMemField, LocalMemoryException, SharedAccessDeniedException, Transaction, TransactionAbortException, Creator, Array, Operator

Table 7.3: Part of the contents of the cava.system package.

The second category is classes which implements parts of the Cava class library. These classes are seen in Table 7.4.

Interfaces:	SystemRunnable
Classes:	SystemOrGate, SystemThread

Table 7.4: Part of the contents of the cava.system package.

Some parts of the Cava library are in fact implemented in the CSL since the components are so fundamental to Cava that they could not be implemented

using other Cava constructs. The solution has been to implement these components in the cava.system package and then let the cava.lang classes extend the cava.system classes.

The third category is classes which are used internally in the CSL to make the entire CRS work. These can be divided further into sub-categories: classes used to create the internal data structures of the CRS, seen in Table 7.5, and classes which manage the execution of Cava programs, seen in Table 7.6.

Interfaces:	Waitable
Classes:	ThreadLocalMem, ThreadGlobalMem, ThreadStorage, ThreadSuspend, TransactionData, TransactionStatistics, SystemList

Table 7.5: Part of the contents of the cava.system package.

Classes:	Manager, MemManager, TransactionManager

Table 7.6: Part of the contents of the cava.system package.

While the categories divide the classes into a structure, it only shows the structure of the system not the way it operates. The operations in the system normally involve several of the classes. In the following sections some of the runtime operations are described.

7.3.4 Thread Local Memory System

Each thread local variable in Cava is encapsulated in a LocalMemField object by the Cava compiler (see Code 7.1 and 7.2 on page 72). To read the variable, the compiler inserts a call to the get method on the variable object. To write a new value to the variable, the compiler inserts a call to the set method on the variable object. The two methods in the LocalMemField object call a method on the MemManager object. When the CSL is initialised, a single MemManager object is created which is used during the entire execution period of the program and for all variables. The MemManager object handles the mapping of the MemField of a variable into an actual value.

Figure 7.2 on the next page shows the data structures which are involved in this mapping in the case of thread local variables. The manager first identifies which thread accessed the variable. A ThreadStorage data structure

is attached to each Cava thread which contains all the data needed by the CRS about a thread instance. One of the items in the ThreadStorage is a ThreadLocalMem object which in essence is a hash map structure. This hash map stores the values of all initialised thread local variables. This implies that when the MemManager needs to retrieve the value of a thread local variable, it must identify the ThreadLocalMem object of the thread, and then fetch the actual value of the variable from the hash map.

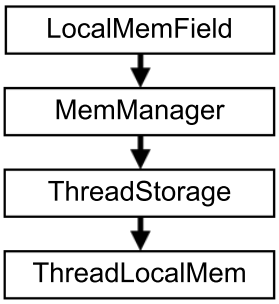

Figure 7.2: Data structures involved in mapping a LocalMemField to its value.

This is the basic implementation behind thread local variables. However, in reality the implementation is more complex since thread local variables must also be transactional. Hence, the data structures which implement STM semantics are also applied in the implementation (see Section vrefsub-sec:sharedmemorysystem).

While the thread local variable implementation in the CSL introduces a large overhead on reading and changing variables, the implementation captures the isolation scenario which Cava requires. The implementation also enables easily capturing the feature that the values of thread local variables should be copied into new threads from the creator thread. This is achieved with the CSL implementation design by making a copy of the hash map in the ThreadLocalMem object. This effectively makes a copy of the values of all variables which are known to the creator thread and induces a relative small overhead when creating new threads.

7.3.5 Shared Memory System

Fundamentally, shared variables are implemented similarly to thread local variables. Each shared variable in Cava is encapsulated in a SharedMemField object by the Cava compiler (see Code 7.1 and 7.2). To read the variable,

the compiler inserts a call to the get method on the variable object. To write a new value to the variable, the compiler inserts a call to the set method on the variable object. The two methods in the SharedMemField object call a method on the MemManager object.

The implementation of shared variables differs from local variables in that the actual value of a variable is directly attached to the SharedMemField object. However, because of the STM semantics, the value is encapsulated inside a number of data structures which create an atomic environment. The data structures, which are seen in Figure 7.3, are based on the concepts of obstruction-free STM from [HLM06] (see Section 10.2.3 on page 102).

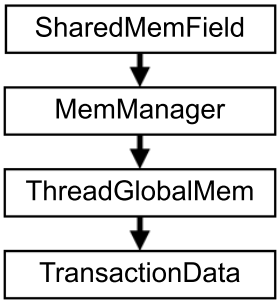

Figure 7.3: Data structures involved in mapping a SharedMemField to its value.

The basic design is that the SharedMemField has a ThreadGlobalMem data structure attached. The ThreadGlobalMem object contains a reference to a data structure called TransactionData. The reference is atomic which is obtained using the AtomicReference class from the java.concurrent.util package introduced in Java 5. The TransactionData object contains a reference to a Transaction object, and an old and a new actual value of the variable. The Transaction object points to the transaction which currently operates on the variable or to the transaction which was the last to do so. The old value stored is the actual value of the variable before the transaction took control of the variable, and the new value stored is the current value of the variable. The operations on the data structures utilise the algorithms described in [HLM06] and Section 10.2.3.

7.3.6 Transaction System

The fundamental construct of the transaction system is a data structure called Transaction. Each transaction has its own Transaction object attached

and each thread has a Transaction object. The latter is saved inside the ThreadStorage object of the thread and represents either a currently active transaction or the most recent transaction executed by the thread. This construction enables calls to LocalMemFields and SharedMemFields to retrieve the transaction to which they belong. Only one Transaction object is attached to a given thread at any given time. This effectively implies that nested transactions are executed as if they were part of the outermost transaction.

A Transaction object can be in four different states: active, aborted, committing, and committed. In general, the Transaction objects functions like the ones described in [HLM06]. However, the extra state of committing has been added which is used when a transaction is in the process of performing a commit. This is necessary since some of the transactions in Cava cannot be committed completely atomically. Hence, the transaction is put into the committing state which cannot be aborted.

7.3.7 Contention Management System

When two transactions attempt to access the same shared variable, the transactions are said to collide. In the CSL, collisions are detected by the second transaction when it tries to access the variable. The detection occurs since the Transaction object attached to the variable data structure indicates that it is already part of an active transaction. When a collision is detected, the second transaction reports the collision to a TransactionManager object.

When the CSL is initialised, a single TransactionManager object is created which is used during the entire execution period of the program. The TransactionManager object implements contention management between threads. When a collision is reported to the transaction manager, this decides on a course of action to resolve the collision. The transaction manager in the CSL applies a simple exponential back off strategy. This implies that the first two times a transaction reports a collision, the thread is simply aborted leaving the other thread alone. If the thread has already been aborted twice on the transaction, both the reporting thread and the other thread are aborted. This is repeated as long as the collision continues to occur. Each time, the potential back off period is doubled.

Chapter 8

Assessment of Cava

This chapter contains an assessment of Cava and a comparison Java in Section 8.1. Both are obtained by applying the question based assessment method which was developed in Chapter 4 on page 16. The assessment and comparison includes a discussion of the results. This is followed by a summary of Cava's most important strengths and weaknesses in Section 8.2 on page 91.

8.1 Assessment and Comparison to Java

The assessment and comparison is obtained by applying the question based assessment method which consists of two steps: an assessment of the individual design criteria which is contained in Section 8.1.1, and an overall assessment which is presented in Section 8.1.2 on page 89.

8.1.1 Assessment of Individual Criteria

The assessment of the individual design criteria is made by answering the questions which were formulated in Section 4.4 on page 19. Note that what is required in order to answer yes or no can also be seen in that section. Following each assessment is a discussion of the most noticeable results along with any points which are not directly uncovered by the assessment.

8.1.1.1 Object-oriented Model

Table 8.1 contains the answers to the questions about the object-oriented model criterion. The results show that Cava complies very well with the object-oriented model criterion since there is a positive answer to every single question. This is considered a strength of Cava since the criterion is at priority level 1 (see Table 3.1 on page 15).

Object-oriented model	Java	Cava
Is the concurrency model integrated into the object-oriented model?	√	√
Does the concurrency model protect against broken encapsulation?	÷	√
Does the concurrency model protect against the inheritance anomaly?	÷	√
Does the concurrency model support reuse of existing classes?	÷	√
Are objects utilised to encapsulate concurrency constructs?	√	√
Are thread constructs encapsulated in objects?	√	√
Are co-operation synchronisation constructs encapsulated in objects?	÷	√
Are competition synchronisation constructs integrated directly into the language?	√	√

Table 8.1: Object-oriented model.

Compared to Java, the most distinctive results are that Cava avoids the problems of broken encapsulation and the inheritance anomaly. The former is the case since Cava's transactions are composable. The latter since the transactional status of a method is part of its interface on inheritance. However, there is one scenario in which encapsulation may be threatened. This appears in the following specific communication pattern of conventional methods. It consists of a conventional method sending a signal to a gate within a transaction block and then waiting for a response on another gate within another transaction block. In case a transaction method invokes such a conventional method, the communication is broken since this results in one single transaction. Hence, other threads are not able to receive the signal from the method and respond to that signal. However, this type of error is deterministic since the errors are produced every time the invocation is made.

Another interesting result is that co-operation synchronisation constructs are encapsulated in objects in Cava. This is obtained by the OrGate and AndGate classes which enable modelling intricate synchronisation patterns at a higher level than in Java. Concurrency in Cava can be introduced using gates without the risk of losing signals as opposed to how notifications can be lost in Java. Furthermore, it is easier to identify synchronisation constructs in Cava since gates represent dedicated synchronisation objects. This is in contrast to Java where the wait, notify, and notifyAll methods are implemented in every object. A final point is that the semantics of gates is not dependent on the state of any threads which is perceived as a major strength of the construct.

8.1.1.2 Expressiveness

Table 8.2 on the following page contains the answers to the questions about the expressiveness criterion. The results show that Cava complies fairly well with the expressiveness criterion but so does Java. The result for Cava is considered a strength of the concurrency model since the criterion is at priority level 1 (see Table 3.1).

A noticeable result is that naturally concurrent problems are not easily modelled without introducing concurrency errors which are not inherent to the problem. This is perceived as a significantly poor result since Cava has a fundamental objective of supporting the programmer in modelling concurrent problems (see Section 1.3.1 on page 2). Co-operation synchronisation is an integral part of modelling concurrent problems which is handled in Cava using gates. The semantics of these (see Section 5.2.4 on page 42) makes them easier to reason about than Java's wait, notify, and notifyAll methods. An example of this is that signals cannot be lost using gates as opposed to how notifications can be lost in Java. Still, the programmer has to consider the following when modelling a concurrent problem: where to apply gates, when they should be signalled, which threads should receive the signals, and so on. Hence, the modelling is not made significantly easier by the straightforward semantics of gates.

Contrary to concurrent problems, parallel problems are easily modelled in Cava without introducing concurrency errors which is the result of the deterministic threads in Cava. The execution of these is predictable which makes them ideal for modelling parallel problems. This is considered a significant result since Cava also has a fundamental objective of supporting the programmer in modelling parallel problems (see Section 1.3.1).

Expressiveness	Java	Cava
Are constructs available to the programmer to introduce concurrency?	√	√
Are naturally concurrent problems easily modelled without introducing concurrency errors which are not inherent to the problem?	÷	÷
Are parallel constructs available to the programmer to introduce parallelism?	÷	÷
Are parallel problems easily modelled without introducing concurrency errors?	÷	√
Is coarse-grained concurrency easily obtained?	√	√
Is fine-grained concurrency easily obtained?	÷	√
Are the synchronisation constructs composable?	√	√
Is it possible to model a lock?	√	√
Is it possible to model a semaphore?	√	√
Is it possible to model message passing?	√	√
Does the concurrency model include shared variables?	√	√
Can thread execution be prioritised? [1]	√	÷
Can suspended threads be interrupted?	√	÷

[1] Cava does not feature priorities on threads

Table 8.2: Expressiveness.

Another interesting result is that fine-grained concurrency is easily obtained in Cava. The main factor behind this is the STM semantics of transactions which ensures that only shared variables, which are accessed by a given thread, are locked. That is, two transactions which do not access the same shared variables can be executed concurrently. This all appears as the default behaviour without the programmer having to express it. The concurrency achieved using transactions appears at a higher level than in Java since the responsibility of implementing the actual protection of shared variables is lifted away from the programmer.

In contrast to Java, Cava fails the question about the possibility of interrupting threads. Not having this possibility could be perceived as a strength since it disallows the programmer from manipulating the suspended threads directly and instead, he must handle the threads by respecting their intended functionality. This implies that if the programmer wants to resume a suspended thread without fulfilling a certain condition, he must build that func-

tionality into the wait condition. However, with regard to the expressiveness criterion, it is a disadvantage since the programmer does not have a default way of interrupting a thread.

A characteristic of threads in Cava which does not directly appear in the assessment is concerning start-up. Unlike in Java, threads in Cava are automatically started as the last operation in the constructor of the Thread object which can be perceived as both a strength and a weakness. It implies that there is no risk of forgetting to invoke e.g. a start method or invoking it multiple times which is considered a strength of Cava's thread implementation. Conversely, it also rules out the default possibility of manipulating threads after instantiation but prior to execution which could be seen as a reduction in expressive power compared to Java. However, the behaviour of Java can easily be emulated using an or-gate which allows for blocking and manipulating the thread until a signal has been to the gate. The technique was utilised in the implementation of the Santa Claus problem (see Section 6.4 on page 58). This shows that starting threads immediately does not pose any limits on the expressive power of Cava.

8.1.1.3 Fault Restriction

Table 8.3 on the following page contains the answers to the questions about the fault restriction criterion. The results show that Cava only complies partially with the fault restriction criterion. The result for Cava is still considered acceptable since the criterion is at priority level 2 (see Table 3.1).

A noticeable result is that non-determinism is only present as an option to the programmer. This result is considered a major strength of Cava since non-determinism is the source of most classical concurrency problems. Threads in Cava exhibit deterministic behaviour unless they are explicitly instantiated as non-deterministic threads (see Section 5.2.1.2 on page 33). This implies that the default behaviour of a Cava program may resemble sequential execution. In order to obtain a higher level of concurrency in a program, non-deterministic execution of threads can be induced.

Shared variables in Cava are protected against corruption by default which induces fault restriction. The protection is a consequence of the fact that variables which are marked with shared may only be accessed within transaction methods or blocks (see Section 5.2.2 on page 34). The transactions ensure that only one thread at a time accesses a shared variable. Furthermore, the explicit indication of shared data heightens the programmer's consciousness about what data should be protected against race conditions and what should not.

Fault restriction	Java	Cava
Is thread management handled by the concurrency model?	√	√
Is the model free from resource starvation?	÷	÷
Is the model free from priority inversion? [1]	÷	√
Does the fundamental concurrency model rely on a mathematical foundation?	÷	÷
Is non-determinism only present as an option to the programmer?	÷	√
Are shared variables protected against corruption by default?	÷	√
Is a variable initialised for all occurrences once it has been initialised?	√	÷
Is the concurrency model disinclined to runtime exceptions which appear due to timing issues?	÷	√
Is it impossible to experience race conditions?	÷	÷
Is it impossible to create a deadlock between two threads?	÷	÷
Is it impossible to create a livelock between two threads?	÷	÷
Does the concurrency model by default protect against race conditions?	÷	√
Does the concurrency model by default protect against deadlock?	÷	√
Does the concurrency model by default protect against livelock?	√	÷

[1] Cava does not feature priorities on threads

Table 8.3: Fault restriction.

A factor, which could be perceived as a weakness of Cava's concurrency model, is that variables may not necessarily be initialised for all occurrences when it has been initialised once. When an object is created, its variables are initialised to some specific value. However, if the object contains thread local variables, these are only initialised for the current thread. All threads, which are descendants of the thread, receive a copy of the values of the thread local variables on thread creation, so the variable is initialised for these threads. If a thread is not a descendant, this thread does not inherit an initial value for the variable, and the value will therefore be undefined. This is a problem since there is no default way of knowing whether a variable was created by an ancestor. Hence, unforeseen errors may occur if the variable is referenced before initialisation.

Table 8.3 shows that it is possible to experience classical concurrency problems like race conditions, deadlock, and livelock in Cava. Hence, Cava does not eliminate these problems which are a common source of error in concurrent programs. However, Cava's concurrency model offers some protection against the problems since it by default protects against race conditions and deadlock. This is a consequence of the fact that threads only share the variables which are necessary to obtain the required concurrency and these have to be accessed within transactions which cannot exhibit neither race conditions nor deadlocks. However, they may exhibit livelocks if several transactions need to access the same shared variable. This may result in a scenario in which the transactions repeatedly abort each other and thus induce a loss of progress in the program.

It is possible in Cava to experience a scenario similar to the problem of nested monitors in Java [Lea99]. However, this does not cause Cava's concurrency model to fail on the question whether it protects against deadlock by default since it may be categorised as a deadlock which is the result of the programmer taking explicit action to introduce it (see the description of the question about protection against deadlock in Section 4.4.3 on page 25). The problem may arise if the programmer utilises two gates and lets a thread invoke receive on the gates in one order and another thread invoke receive in the opposite order. In order to produce the problem the receive calls must be made in separate transaction blocks. If the calls are made in a single transaction, it is not a problem since both signals would be received transactionally.

8.1.1.4 Simple

Table 8.4 on the next page contains the answers to the questions about the simple criterion. The results show that Cava complies fairly well with the

simple criterion but so does Java. The result for Cava is considered acceptable since the criterion is at priority level 2 (see Table 3.1).

Simple	Java	Cava
Is it possible to model sequential problems without considering concurrency?	√	√
Is code executed in a sequential order similar to sequential programs?	√	÷
Does the model consist only of a few simple elements?	√	÷
Is competition and co-operation synchronisation achieved using the same language constructs?	√	÷
Are sequential and implicitly parallel constructs applied using the same keywords? [1]	÷	÷
Are most rules enforced on compile time?	÷	√
Is potential interleaving between threads easy to comprehend and identify?	÷	√
Are variable protection schemes easy to comprehend?	√	√
Is the concurrency model separated from the computer architecture?	√	√

[1] Neither Java nor Cava feature parallel constructs

Table 8.4: Simple.

A distinctive strength of Cava's concurrency model is that potential interleaving between threads is easy to comprehend and identify. This is the case since transaction methods and blocks divide the code into larger entities which reduces the amount of possible interleaving between threads. This supports the programmer in modelling and reasoning about concurrent and parallel problems. Furthermore, it makes debugging easier.

Cava's concurrency model fails on the question about few simple elements since it comprises variables marked with shared, methods and blocks marked with transaction, and gates. Moreover, the constructs have dedicated application areas, i.e. the same language constructs cannot be used to achieve competition as well as co-operation synchronisation which affects the homogeneity of Cava.

The fact that Cava features transaction methods as well as blocks which have the same fundamental semantics is a weakness with regard to simplicity. However, it is necessary since the transactional status of methods is visible on inheritance. Conventional methods have to be overridden by conventional

methods so they are not allowed to access shared variables directly. This can be circumvented by accessing such variables within transaction blocks. Had this not been possible, the original method would have to be a transaction method as well even though this might not be necessary in itself. This could lead to a scenario where methods are made into transactions just in case they need to be transactional in a future subclass. Hence, it could also be seen as a strength to have both constructs in Cava.

A final point on simplicity is that there is an exception to the rule of threads starting immediately in Cava. This occurs when the thread is created within a transaction method or block since such threads are not started until the transaction commits. It may be viewed as an unfortunate feature for the programmer that there are two timings with regard to when threads are started.

8.1.2 Overall Assessment

This section presents the overall assessment of Java and Cava which is constructed by calculating the score based on the answers to the questions in Section 8.1.1.

8.1.2.1 Calculated Score

The four design criteria were prioritised in Table 3.1 on page 15, and in order to take this into account, the priority levels are assigned weights when calculating the overall score. The level 1 criteria of the object-oriented model and expressiveness are given weight 3 while the level 2 criteria of fault restriction and simple are given weight 2.

It varies greatly among the criteria how many questions are used to uncover the individual criterion which also has to be taken into account when calculating the overall score. This is done by assigning each criterion a maximum obtainable score based on its priority level, i.e. the object-oriented model and expressiveness criteria are each assigned the maximum value 15, and the fault restriction and simple criteria are each assigned the maximum value 10. Note that this reflects the weighing described above and results in a maximum obtainable overall score of 50. Furthermore, weighing is not applied between the question in the individual criteria.

The overall score of a given criterion is then calculated by multiplying the maximum value by the ratio of questions which could be answered positively. E.g. the simple criterion has a maximum value of 10, and 5 out of 7 questions

were answered positively for Cava. Hence, the overall score of the criterion becomes $10 \cdot \frac{5}{7} \approx 7.1$ for Cava.

The calculated score of Java's and Cava's concurrency models is seen in Table 8.5.

Assessment	Java	Cava	Maximum
Object-oriented model	7.5	15	15
Expressiveness	10.4	10.4	15
Fault restriction	2.1	5	10
Simple	6.7	5.6	10
Score	26.7	35.9	50

Table 8.5: Calculated score for Java and Cava.

8.1.2.2 Evaluation of Calculated Score

The scores in Table 8.5 show that Cava's concurrency model rates better than that of Java. However, both models rate significantly below the maximum score of 50. The difference in score between Java and Cava is primarily related to how Cava complies with the object-oriented model criterion. Here, Cava obtains the maximal score of 15 which is twice the score of Java. The result is attributed to Cava being better integrated with the object-oriented paradigm which was one of the main goals of the new concurrency model.

Cava also scores better than Java on the fault restriction criterion. However, while the score is significantly better than Java's, 5 compared to 2.1, the score is well below the maximum score of 10. Cava scores slightly worse than Java on the simple criterion, 5.6 compared to 6.7. This is mainly attributed to Cava's lack of homogeneity. On the expressiveness criterion, Cava and Java are tied.

If the scores are viewed as the percentage of positive answers, Cava achieves a 100% on the object-oriented model criterion, 69% on the expressiveness criterion, and 56% on the simple criterion. The first of these results is very satisfying in particular since the integration of concurrency into object-oriented programming languages is considered difficult (see Section 3.1 on page 11).

With regard to the results in Table 8.5, it should be noted that the scores rely heavily on the questions related to the individual criterion. That is,

the scores would probably be different if someone else had defined the questions. Furthermore, the fact that Java and Cava obtain the same score on the expressiveness criterion does not necessarily imply that the two models are similar in that area. Table 8.2 on page 84 shows that Java and Cava have identical answers to the majority of questions but it is also possible to identify the areas in which they differ. The opportunity to make such analyses is exactly what is considered one of the strengths of the question based assessment method since it is transparent (see also Section 4.3 on page 18).

8.2 Strengths and Weaknesses

This section lists the most important strengths and weaknesses of Cava which were identified in the previous section.

8.2.1 Strengths

- Broken encapsulation and the inheritance anomaly are avoided. Both are well-known and fundamental problems when integrating concurrency into object-oriented programming languages.
- Threads only share the necessary data. This implies that threads which only operate on local variables, and thus do not require any synchronisation, can be executed safely in a concurrent way since they can be executed in isolation.
- Non-determinism is only introduced by choice. The non-determinism which is inherent to thread execution in most programming languages is a complicating factor when writing concurrent software.
- Signals cannot be lost on gates. The application of gates is not dependent on the state of the threads which have to utilise the gate to obtain synchronisation.

8.2.2 Weaknesses

- Concurrent problems are not easily modelled which was a fundamental goal of the Cava concurrency model. The concurrency constructs themselves are high-level and semantically easy to reason about but their application in a concurrent problem is still challenging.

- A variable may not necessarily be initialised for all occurrences when it has been initialised once. Hence, it has to be initialised locally which may be a source of errors.
- Invoking a conventional method inside a transaction method may break the implementation of the conventional method. The programmer can therefore be required to know how a method is implemented. However, this type of error is deterministic since the errors are produced every time the invocation is made.
- Transactions may exhibit livelocks. This problem can be reduced by the choice of implementation of the contention manager.

Chapter 9

Future Work

This chapter presents areas of possible future work on Cava. Section 9.1 discusses how further experiments with the concurrency model are needed to verify the value of the model. Section 9.2 on the following page presents known problems and ideas for future development of the concurrency model.

9.1 Experiments

As was concluded in Section 8.1.2 on page 89, Cava obtains a better score than Java in the overall assessment. While this is a noticeable feat for Cava, the result is mainly theoretical since the assessment method is based on key concurrency concepts rather than on practical experiments. What is missing in the assessment are less tangible issues like how the model functions in real development projects, whether the programmer gets the right sense of control over synchronisation, and whether it is easily ensured that a concurrent program design is solid. These types of questions can only be answered by making extensive experiments and getting feedback from real programmers.

Some experiments have already been conducted with Cava but the majority of these were small and rather simple, so larger and more complex experiments are needed. A selection of experiments were documented in Chapter 6 on page 50, and they were conducted using the experimental implementation described in Chapter 7 on page 66. This implementation could be used for further experiments but some general improvements to it are required if the merits of the concurrency model should be verified.

First of all, the implementation should cover the entire concurrency model. In its current incarnation, the implementation does not support deterministic

threads and gate networks. Since these are fundamental constructs in Cava, they should be supported when doing further experiments. Another missing feature is static constraint checks in the compiler. Cava has various constraints built into the model: whether shared variables are accessed outside of transactions, whether a conventional method invokes multiple transaction methods, and whether the transactional status of methods is preserved on inheritance. At present, the Cava compiler ignores these constraints.

Because of differences in semantics, Cava cannot reuse the Java class library so the Cava implementation includes a new class library. However, this library is severely limited compared to the one provided with Java 6 so in order to use Cava for more experiments, the size of the class library has to be increased. This extension should be rather straightforward since it can be constructed on top of Cava. Besides, implementing an extended class library would serve as a very large experiment and thus test the capabilities of Cava.

9.2 Improvements of Cava

The goal of extended experiments is to identify areas of Cava which functions less satisfactorily and therefore should receive further attention. Some areas already appear to be problematic but this cannot be verified until further experiments are performed. Hence, some of the problem areas could simply prove not to be problems in reality.

9.2.1 Thread Local Variable Initialisation

Currently, there may be a problem with the initialisation of thread local variables. If an object with thread local variables was created by a thread, which is not an ancestor of the current thread, the variables may not have been initialised. In the experimental implementation, this scenario is handled by letting the CRS raise an exception when a thread accesses a variable with an undefined value. This is a problematic approach since it is impossible for a thread to know if a thread local variable has been initialised. Therefore, any access to thread local variables in an object involves the risk of raising an exception. If a programmer is to ensure that this does not happen, it must be modelled explicitly. This can be very tedious so a simpler solution for the programmer may be to simply mark all variables as shared. However, this introduces more and larger transactions and would go against the design of Cava.

A way to remedy the thread local variable initialisation problem is to change the semantics of initialisation of variables. One approach would be to cache the first value of the variable and then apply this as the initial value for all non-descendant threads. This implies that a thread, which is not in relation with the thread which created an object, sees the value which the variable was originally initialised with. This would solve the problem with exceptions being thrown because all thread local variables have a default value for any thread. However, it could end up creating more problems than it fixes. The problem is that the initial value may not be the correct value for a specific thread. When the thread executes, it performs calculations based on a wrong value and therefore produces a wrong result. This type of error is harder to detect and correct than runtime exceptions and because of this, the solution was not applied in Cava. In order to determine which solution is the best course of action, more experiments are needed. This could render insight into whether the initialisation poses a genuine problem and how it could be solved.

9.2.2 Nested Transactions

The transactions in Cava appear to be easier to manage than monitors in Java. However, they also display problematic behaviour since certain transactional constructs may result in errors. One particular problem arises if a conventional method contains multiple transaction blocks in order to enable communication with other threads. When such a method is invoked inside a transaction, all transaction blocks are executed as a single transaction, preventing the intended communication.

The problem with this scenario is that a programmer needs to know the inner workings of a method to discover that calling the method inside a transaction poses a problem. If the programmer is not aware of this, the problem manifests itself when the method is invoked. However, the semantics which produces the problem is well-defined and deterministic, implying that the problem appears on every execution of the program. Hence, it is easy to debug since the problem is reproducible. An open question is how common the problem is. If it is very common, extensive testing of programs is required to ensure that the problem does not appear in the end program.

A solution would be if the transaction invoking pattern could be detected at compile time. One way to allow this detection is to introduce a "non-transactional" keyword which could be visible in the class interfaces just like the transaction keyword already is in Cava. Any calls to a method marked

with the non-transactional keyword inside a transaction would then constitute a compile time error. However, a problem with this approach is that any conventional method can be invoked from inside a transaction and thereby become transactional. Such a conventional method could have calls to methods marked as non-transactional which would then also be executed as a single transaction. This would produce the same problem but now in a more complex pattern. However, even though the solution would not remove the problem completely, it may reduce the number of occurrences at runtime.

Another solution is to introduce the possibility of marking transactions as "local" which implies that they commit their changes even though they appear in nested transactions. Hence, calling a method containing multiple local transaction blocks from inside a transaction would allow communication with other threads. However, this approach also introduces problems since the parent transaction may be aborted at some point. If this occurs, changes made by local transactions cannot be rolled back since they have been committed. This could be remedied by making transactions span multiple threads such that the local transactions only appear to have committed their operations to other threads. This would allow the thread communication to be rolled back as well. However, this would be a fundamental and extensive change to the concurrency model.

An ingenious solution to this problem is not obvious and further investigation of the problem is therefore needed.

9.2.3 Other Improvements

An area, which was not given priority in the development of Cava, was performance. This was intentional since the goal was to develop a concurrency model which was easy to work with rather than a fast concurrency model. However, if Cava is to gain widespread use and popularity, performance cannot be ignored.

In its current form, Cava should be able to function in most concurrent settings. This is reflected in the assessment of the expressiveness criterion where Cava and Java obtain the same score (see Table 8.5 on page 90). While this indicates that the current model is expressive, it does not indicate whether the model is unnecessarily extensive. In the conducted experiments, the deterministic thread construct has not come into use. Moreover, the gate network construct has only been applied in one case, namely in modelling Java's wait, notify, and notifyAll constructs (see Section 6.2 on page 51). Deterministic threads and gate networks are therefore constructs which must be investigated further to determine how they actually contribute to Cava.

In the assessment of the expressiveness criterion in Table 8.2 on page 84, Cava failed a number of questions concerning parallel constructs since Cava presently does not feature such constructs. An obvious improvement to the model would therefore be to introduce various parallel constructs which would make it easier to model parallel problems. However, parallel constructs should first be introduced when the underlying concurrency model is solid and stable. If this is the case, introducing parallel constructs should be straightforward since they can be implemented on top of Cava's existing features.

Chapter 10

Related Work

This chapter presents related work along several lines. This includes methods for assessing and comparing concurrency models in Section 10.1, the DSTM2 framework for implementing *Software Transactional Memory* in Section 10.2 on page 101, revocation techniques for Java in Section 10.3 on page 104, and the X10 programming language in Section 10.4 on page 106.

10.1 Concurrency Model Assessment

This sections presents two methods for assessing concurrency models

10.1.1 Assessment by Concurrency Characteristics

In [DH07], Damborg and Hansen describe a method of assessing concurrency models using various characteristics of these. The characteristics are all defined by two opposing extremes which then define a scale on which a concurrency model can be placed. The characteristics are listed and described below and is based on [DH07, Chapter 3].

- **Implicit or explicit concurrency**: In an implicit concurrency model, the concurrency constructs are encapsulated within e.g. libraries, methods, or language constructs. In an explicit concurrency model, the concurrency constructs tend to be relatively low-level, e.g. in the form of locks and semaphores.

- **Fault restricted or expressive model**: A fault restricted concurrency model prevents the programmer from introducing concurrency problems into programs. Opposed to this, an expressive concurrency model places no restrictions on the programmer when expressing the required concurrency.
- **Pessimistic or optimistic model**: A pessimistic concurrency model only runs threads when it can be guaranteed that this can be done safely. Contrary, an optimistic model runs as many threads as possible at any given time. The latter approach requires a recovery mechanism to correct any problems of interfering executions.
- **Automatic or manual parallelisation**: A fundamental objective in concurrent programming is the simultaneous execution of several tasks. When introducing parallelism to a program, the concurrency model provides either automatic parallelisation through the compiler or runtime system, or manual parallelisation which is performed exclusively by the programmer.

Using the above characteristics for assessing concurrency models by placing them on the scales defined by the extremes is suitable for comparing multiple models. However, the placement of a given concurrency model is always subjective since it depends on the experience and knowledge of the person performing the placement. Furthermore, the characteristics are less suitable for assessing a single model since the scales are not absolute. That is, the characteristics cannot be used to assign an objective, e.g. numeric, value to a given concurrency model. It is noted that the characteristics were never intended to serve this purpose.

Another issue in relation to applying the concurrency characteristics is the question whether other or additional characteristics would yield more information about concurrency models.

10.1.2 Assessment by Categories

In [ST98], Skillicorn and Talia present a survey of numerous models and languages for parallel computation. They argue that "an ideal model should be easy to program, should have a software development methodology, should be architecture-independent, should be easy to understand, should guarantee performance, and should provide accurate information about the cost of programs" [ST98, p. 123]. On the basis of this, the various models are placed into categories depending on the level of abstraction which the models provide.

The categories are defined by exploring the criterion "easy to program". That is, a model should conceal various aspects of parallelism from the programmer. These aspects are: decomposition of a program into parallel threads, mapping of threads to processors, communication among threads, and synchronisation among threads. Hence, six categories can be made from this where the level of abstraction decreases [ST98, pp. 135–36].

- Models which abstract from parallelism completely.
- Models in which parallelism is made explicit, but the decomposition of programs into threads is implicit (and hence so is mapping, communication, and synchronisation).
- Models in which parallelism and decomposition both are made explicit, but mapping, communication, and synchronisation are implicit.
- Models in which parallelism, decomposition, and mapping are explicit, but communication and synchronisation are implicit.
- Models in which parallelism, decomposition, mapping, and communication are explicit, but synchronisation is implicit.
- Models in which everything is explicit.

The above levels of abstraction are used as the primary criterion when assessing various parallel models and languages. However, the six categories are subdivided further according to the degree of control over thread structure and communication. This results in three sub-categories: models in which thread structure is dynamic, models in which thread structure is static but communication is not limited, and models in which thread structure is static and communication is limited.

The categories and sub-categories can be viewed as a 6×3-matrix into which models and languages can be placed. Note, Java is placed in the entry where everything is explicit and as having dynamic thread structure. The 6×3-matrix allows for a very fine-grained categorisation of parallel models and languages which must be perceived as a strength of the method. Models and languages can be distinguished at a detailed level which makes the method useful for assessing a single model or language. Similarly, the comparison of multiple models or languages is supported since it is clear exactly how they differ along the two axes of the matrix.

A negative side of the method appears in relation to the six categories and the aspects of parallelism which define the categories. The aspects of decomposition, mapping, communication, and synchronisation represent a hierarchy

in which only one aspect is moved from implicit to explicit when the level of abstraction decreases. Hence, the method does not embrace the assessment of a model in which the aspects appear in a combination different from the six categories.

There is a strong focus on performance and cost when a given model or language is categorised. These issues are not the primary focus of Cava which implies that applying the method to Cava would result in an assessment which does not comply with the design criteria of Chapter 3 on page 11.

10.2 DSTM2

In [HLM06], Herlihy, Luchangco, and Moir present the DSTM2 framework for implementing *Software Transactional Memory*. Some of the concepts in the article are utilised in the transaction system of the CRS (see Section 7.3.6 on page 79).

10.2.1 Overview

The DTSM2 framework is a Java software library which provides a transactional model of synchronisation rather than utilising locks and monitors to manage concurrent access to shared data. This entails that "code that accesses shared memory is divided into *transactions*, which are intended to be executed atomically: operations of two different transactions should not appear to be interleaved" [HLM06, p. 253]. The atomicity of a transaction implies that its operations have either completed successfully and atomically or that they appear to have not taken place at all. The first scenario arises when a transaction commits its operations whereas the second arises when it aborts its operations. A transaction may abort if a conflict occurs between two transactions which is the case if they both need to access the same shared data, and at least one of the transactions needs to write to the data. When a transaction is aborted, it is retried until it can commit and complete its operations successfully. In order to reduce the risk of another conflict, the transaction usually takes precautions against this e.g. by waiting a period of time before re-executing. Different strategies can be implemented in what is called a contention manager.

10.2.2 The DSTM2 Library

A fundamental construct of the DSTM2 library is a special dedicated thread implementation which "can execute *transactions*, which access shared *atomic objects*" [HLM06, p. 254]. That is, DSTM2 features atomic classes which are also at the core of obtaining the STM semantics.

An atomic class is declared in the shape of an atomic interface annotated with an atomic attribute. The interface consists of a number of methods defining the so-called properties of the class. A property defines a virtual field in the class since it is comprised of a pair of methods, getX and setX, where X is the name of the property. The atomic interfaces are passed to the constructor of a so-called transactional factory which can then be used to create instances of the atomic class. This is done by invoking create which is the only available method on the factory. A factory also implements the pairs of getX and setX methods which were defined in the atomic interface passed to it. The DSTM2 library provides a variety of factories but it is also possible to implement new factories.

Transactions in DSTM2 are written in much the same way as a regular sequential method in Java. The only difference is that objects are instantiated by invoking the create method rather than using the new keyword, and that access to the fields in a given object must be made using the getX and setX methods which were defined in the atomic interface of the atomic class. It is the sole responsibility of the programmer to ensure that any objects, which should be shared between threads, are indeed instances of atomic classes. If this is the case, DSTM2 manages all access to such shared objects, alleviating the programmer from ensuring correct synchronisation between threads.

10.2.3 The Obstruction-free Factory

This section describes the obstruction-free factory in DSTM2 since it is applied in the CRS.

A concurrent object is said to be obstruction-free "if any thread that runs by itself for long enough makes progress" [HLM06, p. 257]. The obstruction-free factory creates obstruction-free objects which are represented in the three levels seen in Figure 10.1.

An object consists of a start cell which points to a locator. This has three fields, namely a pointer to the last transaction to write to the object, the old version of the object, and the new version of the object.

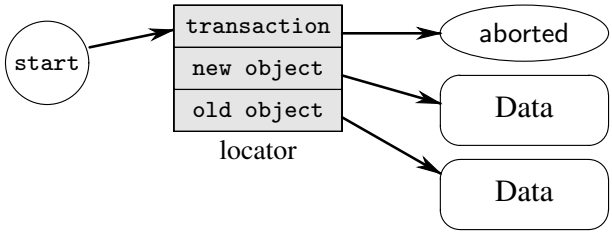

Figure 10.1: Structure of objects created by the obstruction-free factory [HLM06, p. 257].

When a thread needs to write to an object, it checks whether the object is being manipulated by an active transaction. If so, the thread consults the contention manager on which action to take. The actions can be either for the thread to wait a while, to abort the conflicting transaction, or to abort itself. In case the last thread either committed or aborted, the object is available to the current thread. Depending on the status of the previous thread, two scenarios may occur.

If the last transaction committed, the new version of the object is valid. Hence, the thread makes a new locator which points to the current transaction, the previous new version is cloned to the current new version, and the current old version points to the previous new version. This is seen in Figure 10.2. This ensures that if the thread commits successfully, the current new version is valid. Contrary, if the thread aborts, the current old version is valid since it points to the prior new version.

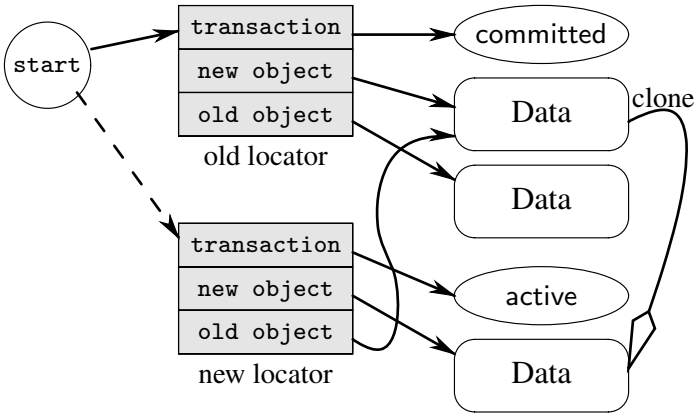

Figure 10.2: The previous thread committed successfully [HLM06, p. 258].

If the last transaction aborted, the old version of the object is valid. Hence,

the thread makes a new locator which points to the current transaction, the previous old version is cloned to the current new version, and the current old version points to the previous old version. This is seen in Figure 10.3. This ensures that if the thread commits successfully, the current new version is valid. Contrary, if the thread aborts, the current old version is valid since it points to the prior old version.

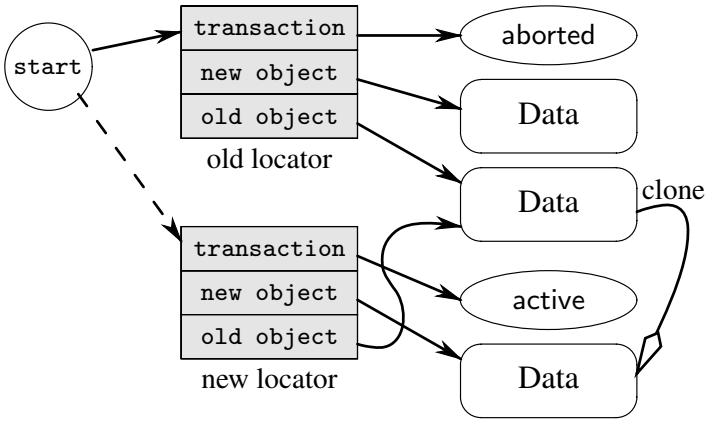

Figure 10.3: The previous thread aborted [HLM06, p. 258].

In both cases, DSTM2 achieves atomicity by swinging the start reference to the new locator object using a compare-and-swap reference pointer. This makes the entire creation of the locator object appear atomic when other threads look at the object.

10.3 Revocation Techniques for Java

In *Revocation techniques for Java concurrency* [WJH06], Welc, Jagannathan, and Hosking describe an approach to managing concurrency in Java. Some of the concepts in the article are similar to transactions in Cava.

10.3.1 Overview

In concurrent programming, a major challenge lies in protecting shared data. Many languages support this by offering the concept of guarded code regions. In [WJH06, p. 1614], these are characterised by "accesses to shared data performed by one thread are *isolated* from accesses performed by other threads,

and all updates performed by a thread within a guarded region become visible to the other threads *atomically*, once the executing thread exits the region". In Java, guarded regions are provided by synchronised methods and blocks and in Cava by transaction methods and blocks.

Normally, guarded regions are implemented by applying some scheme of mutual exclusion, e.g. locks or monitors. A number of concurrency problems are related to using locks and monitors, e.g. race conditions, deadlocks, livelocks, etc. [WJH06] presents two new implementations of monitors which offer guarded regions and may alleviate the problems of monitors while also increasing performance of a concurrent program. The two types are denoted revocable monitors and transactional monitors where the latter is more interesting in relation to Cava.

10.3.2 Revocable Monitors

The main principle behind revocable monitors is the ability to revoke the operations which a thread has performed within a monitor. This is necessary when a conflict on the monitor is detected by the runtime system. That is, the operations of a thread may be rolled back and retried at a later point. The revocation of operations is enabled by logging the operations of each thread.

Revocable monitors solve the problem of priority inversion and thus increases throughput for threads with high priority. Deadlocks may also be resolved but if they are inherent to the program, and thus not the result of scheduling, they appear as livelocks instead [WJH06, p. 1615].

10.3.3 Transactional Monitors

Transactional monitors extend the functionality of revocable monitors. They are described as "an entirely new synchronization framework that addresses the performance impact of classical mutual exclusion while simplifying concurrent programming" [WJH06, p. 1615]. Transactional monitors are able to increase throughput of a program since they do not rely on mutual exclusion to guarded regions the way revocable monitors do. Instead, it is only required that transactional monitors appear to acquire monitors serially. That is, when multiple threads operate within the same monitor, they are allowed to do so as long as their operations are serialisable, i.e. the operations of the threads are equivalent to some serial schedule without interleaving. It is the responsibility of the implementation to manage transactional monitors

which implies that the programmer can specify guarded regions more extensively without affecting performance. The implementation simply relaxes the guarded regions by inducing transactional semantics on these.

The transactional semantics is enabled by logging the operations of each thread in a thread specific log. Once a thread is ready to leave the monitor, it attempts to commit its operations using the log. In case this fails, the thread re-executes its operations and retries the commit operation.

10.4 The X10 Programming Language

This section presents the X10 programming language. The description is based on [ESS04] and [Sar06] and has a particular focus on the concurrency constructs of the language. This is the case since there are some similarities to thread local and shared variables as well as transactions in Cava.

X10 is an experimental programming language currently under development at IBM. It is based on Java and has the goal of being an object-oriented programming language designed specifically for high-performance and high-productivity programming of large-scale computer systems [ESS04, p. 1]. Since X10 is based on Java, the basic parts of the language resemble those of Java. However, some of Java's constructs have been removed such as the Thread class and the wait, notify, and notifyAll methods. All other constructs from Java have been adopted largely unchanged.

The basic construct in X10's concurrency model is the concept of a place. This is a logical unit which encapsulates data and processing power and a typical place is a single multi-core processor. A place consists of a finite collection of light-weight threads, which are termed activities, and data [Sar06]. Activities are located at one place for their entire lifetime and unlike the thread construct in Java, activities in X10 are not bound to an object. Objects in X10 are separated into two categories: value objects and reference objects. Value objects are immutable and stateless and can therefore be copied between places. Reference objects may contain state and are located at a particular place for their entire lifetime. The data in an X10 program is divided into four storage classes [ESS04, p. 2] which are listed below.

- *Activity-local*: Data in this class is private to an activity and is located at the same place as the activity.
- *Place-local*: Data in this class is shared between all activities at the given place.

- *Partitioned-global*: Data in this class can be accessed both by activities in the same place and activities at other places. This storage class represents a global address space between places and activities.
- *Values*: Data in this class is immutable and stateless which implies that it can be copied between places. Furthermore, methods on such data can be invoked from any place.

There are various constructs in X10 for obtaining synchronisation between activities: futures, unconditional or conditional atomic sections, and clocks. In case an activity needs to access data which is neither activity-local nor place-local, it does so by spawning an asynchronous activity at the place where the data is located. The creating activity continues its execution in parallel with the spawned activity. An asynchronous activity may return a value to the creating activity which is done in the form of a future. When an activity wishes to access the future, it becomes blocked until the value is available.

To protect activities from interfering with each other, X10 features atomic sections which have STM semantics. An atomic section may be either unconditional or conditional. The latter does not process the block statements until a certain Boolean condition is satisfied.

The last major concurrency concept is that of a clock which is a generalisation of the classical synchronisation barrier. However, the collection of activities which synchronise on a given clock may change dynamically which makes clocks more expressive than classical barriers. Clocks support “deadlock-free parallel computation” [Sar06, p. 4].

Chapter 11

Conclusion

This chapter addresses how the results of the project relate to the goals in Section 1.4 on page 4. Cava and differences from Java are discussed in Section 11.1. The assessment of Cava and Java is the focus of Section 11.2. Finally, Section 11.3 on page 110 addresses the Cava implementation and the examples modelled in Cava.

11.1 Cava and Differences from Java

The motivation behind Cava was to create a new concurrency model for Java which would enable the programmer to easily design concurrent programs in order to utilise the processing potential introduced by multi-core CPUs. The new concurrency model should be tightly integrated with the object-oriented paradigm and embrace both modelling naturally concurrent problems and parallel optimisation.

Cava grew out of two different studies. The first was of Java's present concurrency model to gather information about its constructs and problem areas. The second had the purpose of identifying design criteria for the new model to pinpoint the goals of the design.

Cava's most important differences from Java are addressed in the following. In Cava, non-deterministic execution of threads is a choice available to the programmer which implies that it is only introduced if the programmer explicitly requests it. This is contrary to Java where non-deterministic behaviour is the only option. Transactions in Cava restrict interleaving on shared variables to large blocks. This reduces the potential interleaving to

be considered by the programmer in contrast to Java where interleaving may appear at any level. Even though a Cava program may have large transactional code regions, Cava maintains a high degree of concurrency. In Java, a programmer must manually specify the level of concurrency. Cava features a reliable communication construct in the form of gates which remedies the problem of losing notifications in Java if an object is not in the right state.

The differences described above appear at different levels and have different complexity. However, the changes in Cava improve various aspects of Java such that they collectively resulted in a concurrency model which is stronger than Java's present one.

11.2 Assessment of Cava

When designing a new concurrency model, a natural issue becomes how to measure the result. A comparison is important since it helps decide whether the new concurrency model is actually an improvement over the old one. As a result, a new assessment method was introduced in Chapter 4 on page 16.

The assessment method was developed in accordance with three criteria: the method should produce an absolute result, it should be unbiased, and it should be transparent. These criteria led to a question based assessment method which in this context had its starting point in the four design criteria of Cava found in Chapter 3 on page 11. However, the questions in the method could be used to uncover other characteristics of concurrency models.

The assessment method was applied to Java and Cava in Section 8.1 on page 81 with the questions which were defined in Section 4.4 on page 19. The result of the assessment, seen in Table 8.5 on page 90, was that Java had an overall score of 26.7 while Cava had an overall score of 35.9 compared to a maximum obtainable score of 50. This showed that Cava constitutes an improvement to Java's concurrency model. However, Cava was still far from the maximum score.

The difference in score between Java and Cava was primarily related to how Cava complies with the object-oriented model criterion. Here, Cava obtained the maximal score and twice the score of Java. The result can be attributed to Cava being better integrated with the object-oriented paradigm which was one of the main goals of the new concurrency model. Cava also scored better than Java on the fault restriction criterion. However, while the score was significantly better than Java's, the score was well below the maximum score of the criterion. Cava scored slightly worse than Java on the simple

criterion which was mainly attributed to Cava's lack of homogeneity. On the expressiveness criterion, Cava and Java were tied.

The criteria of Chapter 3 drove the design of Cava as well as the development of the assessment method. Hence, it could be argued that Cava's better score was predictable. However, the overall score reflected that the design of Cava focussed mainly on the high priority criteria and relaxed the low priority criteria. Moreover, the assessment method is rather theoretical and will therefore probably favour individual constructs and concepts rather than the applicability of the complete concurrency model. The result of the assessment should therefore be taken as an indication that Cava's constructs are stronger than those of Java but they may not necessarily be easier to apply in programs.

11.3 Implementation and Examples

As mentioned in the previous section, Cava's applicability was not targeted in the assessment method. Hence, it is difficult to predict how easy it is to comprehend the concurrency constructs and whether the model supports the programmer when designing concurrent programs. How well Cava performs with regard to such issues can only be determined by actually applying it to concurrent and parallel problems.

In order to facilitate gaining practical experience with Cava, an experimental implementation was developed. This was obtained by changing Sun's Java compiler into a Cava compiler and implementing a Cava Runtime System. The fundamental implementation strategy is to let the Cava compiler hijack the Java compiler after the latter has parsed the source files into an abstract syntax tree. A tree traversal is performed during which the Cava code is replaced with equivalent Java code. The component, which performs the replacements, is rather complex since the surrounding code and the context in general may influence how the equivalent Java code should be constructed. Since the rest of the Java compiler is left unchanged, the Cava compiler generates class files which can be executed on a standard JVM as long as the Cava Runtime System is specified to this.

The implementation was utilised to write and execute various Cava programs. These included concurrency constructs such as a lock and Java's wait, notify, and notifyAll methods. Moreover, classical concurrent problems such as Dining Philosophers and the Santa Claus Problem were modelled along with a parallel version of Quicksort. The examples showed that synchronisation between threads was easily obtained, regardless of whether the purpose was to

protect shared data or to enable co-operation. An important point in relation to this is that all examples were modelled by the project participants who necessarily have extensive prior knowledge of the syntax and semantics of the various constructs. Hence, it would be fruitful to let programmers, who are inexperienced with Cava, apply it to concurrent and parallel problems. Still, the overall impression is that Cava contributes positively to making modelling easier than in Java but it remains challenging to design concurrent programs.

Bibliography

[C+01] T.H. Cormen et al. *Introduction to Algorithms*. MIT Press, second udgave, 2001.

[DH07] B. Damborg and A.M. Hansen. A Study in Concurrency. Dat 5 Project Report, Department of Computer Science, Aalborg University, Denmark, January 2007.

[ESS04] K. Ebcioğlu, V. Saraswat and V. Sarkar. X10: Programming for Hierarchical Parallelism and Non-Uniform Data Access (Extended Abstract). In *Language Runtimes '04 Workshop: Impact of Next Generation Processor Architectures On Virtual Machines (colocated with OOPSLA 2004)*, October 2004.

[G+96] J. Gosling et al. *The Java™ Language Specification*. Addison Wesley Publishing Company, 1996.

[G+05] J. Gosling et al. *The Java™ Language Specification, Third Edition*. Addison Wesley Professional, 2005.

[HLM06] M. Herlihy, V. Luchangco and M. Moir. A Flexible Framework for Implementing Software Transactional Memory. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 253–262, 2006.

[HPC07] *Confronting Parallelism: The View from Berkeley*. `http://www.hpcwire.com/hpc/1288079.html`. Accessed: 8 March, 2007.

[JAS] *Java™ Platform, Standard Edition 6 API Specification*. `http://java.sun.com/javase/6/docs/api`. 2006.

[JCU] *Java Concurrency Utilities Documentation*. `http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html`. Accessed: 11 November, 2006.

[Lea99] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Professional, 1999.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150, 1993.

[Ope] *OpenJDK – The Java programming-language compiler*, `https://openjdk.dev.java.net/compiler`. Downloaded: 9 February, 2007.

[Pap89] M. Papathomas. Concurrency Issues in Object-Oriented Programming Languages. *Object Oriented Development*, pages 207–245, 1989.

[San04] B. Sandén. Coping with Java threads. *Computer*, 37(4):20–27, 2004.

[Sar06] V. Saraswat. Report on the Experimental Language X10, Version 1.01. IBM Research, December 2006. Downloaded: 25 April, 2007 from `http://x10.sourceforge.net/docs/x10-101.pdf`.

[Seb03] R.W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Longman Publishing Co., Inc. – Boston, MA, USA, 2003.

[ST98] D.B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.

[Tro94] J.A. Trono. A New Exercise in Concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[WJH06] A. Welc, S. Jagannathan and A.L. Hosking. Revocation techniques for Java concurrency. *Concurrency and Computation: Practice and Experience*, 18(12):1613–1656, 2006.

Appendix A

Basic Concepts

This appendix presents some basic concepts which are related to concurrency. Several views on and definitions of these basic concepts exist, depending on the approach taken to concurrency. The purpose of this appendix is to establish how the various concepts are used in the main report. With the exception of Definition A.3 and Definition A.7 on page 116, the definitions are similar to these ones presented in [DH07, Appendix A].

A.1 Definitions

A.1.1 Process, Thread, and Task

Definition A.1 (Process)**.** *A* process *is an instance of a program executed in isolation from other processes. It consists of program code and data.*

With the above definition, examples of processes are an instance of a word processor or an instance of a CD player. It is noted that Definition A.1 is largely consistent with the prevalent process definition used within the operating systems research field.

Definition A.2 (Thread)**.** *A* thread *is associated with a specific process and it realises (part of) the execution of the program which the process represents. The thread operates on the same memory as the process which created the thread.*

With the above definition, a process may be an instance of a word processor and this may spawn a thread handling the editor, a thread handling the spell

checker, and a thread handling pagination. The above shows that it is the threads which realise the actual execution of the program which the process represents. Furthermore, a given process always spawns at least one thread.

Definition A.3 (Task)**.** *A* task *realises (part of) what an entire program is intended to realise.*

With the above definition, a single statement or method within a program is a task. On the other hand, an entire program is also perceived as a task.

A.1.2 Concurrency and Parallelism

Definition A.4 (Concurrency)**.** Concurrency *is the simultaneous execution of tasks.*

With the above definition, concurrency is observed in the simultaneous execution of instances of a word processor and a CD player. Likewise, a server which handles requests by setting up a new thread for each request exhibits concurrency. Concurrency can appear at several levels: instruction level, statement level, subprogram level, and program level [Seb03, p. 496]. Note, a task as defined in Definition A.3 can be either of these four entities. Furthermore, concurrency can be characterised as either physical (or true) where several processors are available or as logical where only one processor is present but concurrency is simulated by interleaving the execution of several threads [Seb03, p. 498].

Definition A.5 (Parallelism)**.** Parallelism *involves the partitioning of tasks into a number of subtasks which can be executed independently.*

With the above definition, parallelism can be applied when calculating the factorial of 100. This can safely be divided into several subtasks which each calculates only part of the result.

A.1.3 Synchronisation

Definition A.6 (Synchronisation)**.** Synchronisation *involves the co-ordination between threads such that they can perform a task by executing concurrently.*

The goal of synchronisation can either be to prevent competition if access to shared memory is required by multiple tasks or to obtain co-operation between multiple tasks [Seb03, p. 500].

With the above definition, an example of competition synchronisation is the scenario in which two threads both have to write to a shared variable. Conversely, a buffer constitutes a scenario in which co-operation synchronisation is required. Note, Definition A.6 also embraces synchronisation between processes since every process spawns at least one thread.

A.1.4 Concurrency Model

Definition A.7 (Concurrency Model)**.** *The* concurrency model *of a programming language consists of the constructs which are included in the language with the sole purpose of supporting modelling concurrent and introducing parallelism.*

With the above definition, the following elements can be part of a concurrency model: keywords, methods, dedicated classes (e.g. in libraries), and dedicated objects.

Appendix B

Santa Claus Problem

This chapter introduces a concurrent problem known as the Santa Claus problem. The description of the problem is largely identical to [DH07, Chapter 4]. The Santa Claus problem was designed by John A. Trono and published in 1994 in the article *A New Exercise in Concurrency* [Tro94].

B.1 Problem Description

The participants in the problem are Santa Claus, nine reindeer, and an unspecified number of pixies who manufacture toys.

> *Santa likes to sleep in his shop so he does so as often as he can. However, he is eventually awakened by pixies who are having problems making toys, or by the reindeer when all nine are back from their year long vacation. One pixy with a problem is not important enough to wake up Santa, so the pixies always visit Santa in groups of three. While a group of pixies are in Santa's shop to have their problems fixed, any other pixy who wishes to visit Santa must wait for the other pixies to return.*
>
> *The reindeer do not return from their vacations until the last possible moment. The last reindeer to arrive back must get Santa while the others wait in a warming hut until Santa is ready to harness them to the sleigh. Once they are harnessed, they are ready to deliver the presents. When they return from this trip, Santa releases the reindeer and sends them back on vacation.*

In case Santa wakes up to find three pixies waiting at the door to his shop along with the last of the reindeer having returned from vacation, Santa has decided that the pixies must wait until after Christmas. This is because it is more important to get his sleigh ready as soon as possible.

B.2 Properties

There are several reasons why the Santa Claus problem was selected as an example of a concurrent problem. Firstly, the problem does not assume any specific concurrency model. Furthermore, the Santa Claus problem was designed as a problem where concurrency appears as a natural part. Lastly, the nature of the Santa Claus problem contributes with several non-trivial synchronisation patterns.

Summary

The goal of the project was to develop a new concurrency model, called Cava, for the Java programming language. The motivation was the increasing need for concurrent programs e.g. to utilise the power of multi-core CPUs. Moreover, Java's present concurrency model suffers from various weaknesses such as broken encapsulation and the inheritance anomaly.

Based on an investigation of Java's present concurrency model, design criteria for a new concurrency model were defined: it should comply with the underlying object-oriented model, be expressive, offer fault restriction, and be simple. Assessing the new model was important to evaluate how it met the design criteria. Moreover, it should be investigated whether the new concurrency model is stronger than Java and offers better support for modelling concurrent problems and introducing parallelism. Hence, an assessment method, which assigns a numerical value to a concurrency model, was developed based on posing questions about each design criterion.

Cava builds on Java but the wait, notify, and notifyAll methods were removed along with the synchronized keyword. The fundamental concurrency construct in Cava is still the thread. However, non-deterministic behaviour of threads appears only by choice. Moreover, all variables are thread local unless they are marked as shared between threads with the new shared modifier. Access to shared variables is restricted since it must occur within methods or blocks which are marked with the new transaction keyword. Such methods and blocks apply *Software Transactional Memory* which allows for performing roll backs in case of conflicts on variables. The transactional status of methods is preserved on inheritance. Cava also features a gate construct which enables sending messages between threads. There are two types of gates: or-gates which forward a message to a single thread, and and-gates which forward a message to multiple threads.

An experimental implementation of Cava was developed as part of the project by changing Sun's Java compiler into a Cava compiler and implementing a Cava Runtime System. The strategy was to hijack the Java compiler which implies that the generated class files can be executed on a standard JVM if the Cava Runtime System is specified to this. The implementation enabled writing and testing Cava programs, e.g. Cava was applied to model a lock, Dining Philosophers, the Santa Claus Problem, and a parallel version of Quicksort.

Cava achieved a better score than Java in the comparison. This was primarily obtained on a significantly better integration with the object-oriented paradigm and more fault restriction. There was a tie with regard to expressiveness while Cava scored slightly worse than Java on simplicity. Since Cava was far from the maximum score, areas of potential future work were presented along with examples of related work.