A UML/MARTE Model Analysis Approach for Detection of Concurrency Faults

by

Marwa Shousha, M.A.Sc.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering

(Ottawa-Carleton Institute for Electrical and Computer Engineering (OCIECE))

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

September 2010

Your file Votre référence
ISBN: 978-0-494-79635-1
Our file Notre référence
ISBN: 978-0-494-79635-1

Abstract

Concurrency faults, such as starvation and deadlocks, should be identified early in the design process. This is made increasingly difficult as larger and more complex concurrent systems are being developed. We propose a general approach - based on the analysis of specific models expressed in the Unified Modeling Language (UML) - that uses specifically designed genetic algorithms (GA) to detect deadlocks, starvation and data race faults in concurrent systems. Our main motivations are (1) to devise practical solutions that are applicable in the context of the UML design of concurrent systems without requiring additional modeling and (2) to use a search technique to achieve scalable automation in terms of concurrency problem detection. To achieve the first objective, we show how all relevant concurrency information is extracted from systems' UML models that comply with the UML Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile. For the second objective, a tailored GA is used to search for execution sequences exhibiting concurrency faults, namely deadlocks, starvation and data races. Scalability in terms of problem detection is achieved by showing that the detection rates of our approach are in general high and are not strongly affected by large increases in the size of complex search spaces.

Table of Contents

Table of Figures

List of Tables

Glossary of Terms and Acronyms

Access Time	Specific time unit chosen within acceptable thread access range of locks
Asynchronous Time	View of time dependent on underlying operating system on which implemented real time system will eventually run. In this model of time, execution times of threads are not known in advance, hence deadlines are defined for threads
CFD	Concurrency Fault Detector
Chromosome	Representation of solution in genetic algorithm
Crossover	Operation to create new offspring by mixing parent genes
CTL	Computation Tree Logic
DECF	Data Entry Complete Flag
Faults	Communication and resource sharing errors in concurrent systems; in particular: deadlocks, starvation and data races.
Fisher Exact Hypothesis Test	Statistical significance test used to analyze small data sets
Fitness	Quality that defines which chromosomes are closer to the optimal solution
GA	Genetic Algorithm
Gene	Elements within a chromosome
Generation	The process of crossover, mutation, fitness comparison and replacement
Global Optima	Points in search space that result in concurrency problems
GQAM	Generic Quantitative Analysis Modeling

GRM	Generic Resource Modeling
HC	Hill Climbing
HLAM	High-Level Application Modeling
Load Scalability	Property of systems that function well without undue delays or resource consumptions. Makes good use of available resources
Local Optima	Points in search space where all surrounding points have worse fitness, but point itself is not an instance of a concurrency fault
Locks	Various locking mechanisms such as semaphores, mutexes, databases, etc.
LTL	Linear Temporal Logic
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MEOS	Mode/Energy Offset
MDA	Model Driven Architecture
MDD	Model Driven Development
Mutation	Operation to mutate genes in a chromosome
Objective Function	Measure of fitness of chromosomes
Physical Time	See Asynchronous Time
Population	Collection of all chromosomes used by the GA
PUMA	Performance by Unified Model Analysis
P-Value	Probability value; the probability of obtaining test statistics as extreme as the ones observed in the data set. The lower the value, the less likely this is the case
RA	Random Generation
RAG	Resource Allocation Graph

Search Space	Set of possible sequences of thread accesses to locks or shared resources. The size of search space is design solution specific. It is affected by the number of threads, locks or shared resources, as well as the designated time interval
Selection	Process by which individuals are chosen for survival to the next generation
Space Scalability	Quality of systems dealing with system memory requirements; So long as memory requirements do not grow to intolerable levels, system has space scalability
Space-Time Scalability	Quality of system that measures its ability to grow, or expand, as the size of the problem increases
SPT	Schedulibility, Performance and Time
SRM	Software Resource Modeling
State Space	Total possible of combination of states system under test can be in, independent of the design solution
Structural Scalability	Quality referring to system enhancement by adding additional functionality with little effort
Termination Criterion	Stopping condition for GAs
Thread Access Range of Locks	Time range when a thread can accesses a lock
Thread Exec Times Within Locks	Execution time of a thread in a lock
Time Interval	Interval during which test engineer wants to study system's behavior
UML	Unified Modeling Language

1 INTRODUCTION

Concurrency abounds in many applications on a daily basis. As defined by Edsger Dijkstra, *"Concurrency occurs when two or more execution flows are able to run simultaneously."* In so doing, such systems often communicate with each other and access shared resources. Both communication and resource sharing are a common source of errors in concurrent systems. Particular instances of these two types of errors are referred to as faults in this thesis. Determining when various concurrency faults may arise is not an easy task due to the large number of possible execution schedules, or thread interleavings, commonly feasible in concurrent systems, which are non-deterministic by nature: their execution sequences may vary for the same inputs.

The earlier concurrency faults are detected, the easier and less costly they are to fix. Ideally, such concurrency faults should be identified early in the design process, when full system implementation is not available. In the design phase, systems are expressed as models. System models contain information that is not available at the code level, such as timing information. For example, a model can contain the expected execution time of a particular thread, but this information is not readily available at the code level. Instead, some means of monitoring is needed to estimate the execution time at the code level. Models also give a general overview of the system that is often difficult to grasp at the code level. However, such models can be expressed in various ways using various tools. With the recent trend towards Model Driven Development (MDD) [Klep 03], the choice of using Unified Modeling Language (UML) models and their extensions as a source of concurrency information at the design level is natural and practical. The use of UML has the additional advantage of allowing better communication

between various roles involved in development (i.e., designers, developers and testers) whereby the same UML design documents are used throughout [Bake 04].

1.1 Problem Description

Concurrency faults, such as deadlocks and starvation, should be identified early in the design process; when they are the least costly to fix. This is made increasingly difficult as larger and more complex concurrent systems are being developed. With the recent trend towards MDD, models are commonly being expressed using UML. The analysis of concurrency faults should be possible using UML, without requiring additional modeling or a high learning curve on the part of designers, or should at least minimize it. We hence propose an approach for the detection of concurrency faults that is based on design models expressed in UML [OMG 09].

1.2 Thesis Contribution

Our aim is to develop a scalable, automated method that can be easily tailored to several types of concurrency faults, and that can be easily integrated into a Model Driven Architecture (MDA) approach, the UML-based MDD standard by the OMG [Klep 03]. This is achieved through three steps:

1. Demonstrating the feasibility of extracting relevant concurrency information from UML/MARTE design diagrams.

2. Showing the effectiveness of the proposed search based technique in detecting concurrency faults.

3. Demonstrating scalability in terms of fault detection as the size of the problem grows.

Demonstrating the feasibility of extracting relevant concurrency information from UML/MARTE design diagrams: To achieve this objective, we show how relevant concurrency information is extracted from systems' UML models. When the UML notation is not enough to model vital aspects of a system for a given purpose, the notation is extended via profiles. Of particular interest is the standardization of the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile [MART 09] that addresses domain specific aspects of real-time, concurrent system modeling.

Showing the effectiveness of the proposed search based technique in detecting concurrency faults: Three tailored genetic algorithms are used to search for execution sequences exhibiting one of three types of concurrency faults, namely deadlock, starvation, or data race faults.

Demonstrating scalability in terms of fault detection as the size of the problem grows: Scalability in terms of fault detection is achieved by showing that the detection rates of our approach are in general high and are not strongly affected by large increases in the size of complex search spaces.

The approach we describe is based on a UML/MARTE annotated front end model of the system under test, coupled with a backend based on tailored GAs, each directed at finding a particular fault. We begin by automatically collecting information relevant to a fault from the system's UML design model extended with the MARTE profile. This step can be automated using one of various existing UML tools and the underlying UML/MARTE metamodel. The extracted information is then fed to the appropriate GA, depending on the type of fault the designer is interested in uncovering. Since concurrency faults can be revealed by specific interleavings of thread executions, the GA specifically searches

for such thread execution interleavings that have a high probability of exhibiting a particular type of fault.

Our approach is meant to be general and can be adapted to a variety of concurrency faults by tailoring the fitness function of our specifically designed GA in order to address faults such as deadlock, starvation and data races. The proposed method is also geared towards large, complex systems characterized by numerous interacting threads and frequent synchronization. For this particular type of problem, GAs seem to fare well, although as discussed in [Harm 09], there is no conclusive way to determine the most suitable optimization technique for a given problem. We target non-distributed, real-time, concurrent systems in general, with a particular emphasis on soft real-time systems. Our approach is best used on new systems being developed in an MDD environment. Hence, model driven development is considered from the onset. Our approach can still be used at later stages, when parts of the system to be tested are coded. However, any errors uncovered at this stage may be costly to fix.

1.3 Thesis Organization

Section 2 presents concurrency background information. Readers familiar with these concepts may choose to skip this section. Section 3 provides a very general outline of the MARTE profile, highlighting particular aspects needed in our approach. Section 4 presents related work. Section 5 provides the details of our approach. Tool support is presented in Section 6, which is used to run the case studies presented in Section 7. We conclude in Section 8.

2 BACKGROUND

This section serves to familiarize readers with the fundamental concepts used in this work. Section 2.1 describes aspects of concurrency. Important concepts that are required to detect faults in our approach (later in Section 3) appear in italics in this section. A concurrency fault taxonomy is presented in Section 2.2, with a continuation of the important concepts appearing in italics. Section 2.3 gives an overview of concurrency concepts present in UML. This section ends with a general overview of genetic algorithms.

2.1 Concurrency

Concurrency is defined as the simultaneous execution of more than one activity. Hence, many activities occur in parallel [Goma 00]. Concurrency may be distributed across many processors or simulated on a single processor. In the former case, each activity is assigned to a different processor. All executions thus occur simultaneously. In the latter case, activities are time sliced on a single processor [Kram 06]. Each activity is also called a *thread.*

Activities executing in parallel require some means of sharing information. Information sharing occurs in one of two ways: 1. Through shared memory, and 2. Through the use of messages.

Use of shared memory is prevalent in a non-distributed environment where all threads have access to common data and memory. Here, semaphores (along with variations such as mutexes, databases, etc...) are prevalent to ensure mutual access to these data and memory when needed [Down 05]. Semaphores, or counting semaphores, represent multiple access locks. They ensure that at most `n` threads

will have access to shared data or memory, where `n>1` [Down 05]. This is ensured through *acquire* and *release* operations which keep track of the number of threads accessing the semaphore. The value of `n` determines the *capacity* of the semaphore. Semaphores along with all other variations will collectively be referred to as *locks* in the remainder of this thesis. Whenever a number of threads greater than `n` try to access a lock, the excess threads will be placed in a *wait queue*. *Wait queues access* may occur in FIFO fashion, round robin, shortest-job-first, priority, etc, with the particular mode of access determined by the programmer. The various access policies require additional knowledge about threads. FIFO and round robin imply knowledge of *thread access ranges of locks*. Shortest-job-first implies knowledge of *thread execution times within locks* while the priority access policy implies knowledge of *thread priorities*.

The second type of information sharing, namely message passing, is common in distributed environments, as well as client-server and peer-to-peer applications. Here, two threads communicate by passing messages back and forth. Messages may be sent to a particular thread or they may be broadcast to more than one thread [Goma 00]. Messages may be sent either *synchronously* or *asynchronously*. In asynchronous - or loosely coupled interaction - the sender does not wait for a reply from the receiver. Hence, the sender is free to resume its own processing once it has sent the message. Synchronous - or tightly coupled interaction - is of two types: with reply and without reply. In the former, the sender blocks while waiting for a reply from the receiver. The reply is generated after the receiver has completed the processing necessary as indicated by the message. Hence, the sender can only resume its own processing after the receiver completes its processing of the message. In synchronous communication without reply, the sender sends a message and waits until the receiver has accepted the message.

The sender is released once the message is received by the receiver. Here, the sender need not wait for complete processing of the message; reception by the receiver is sufficient [Goma 00].

2.2 Concurrency Fault Taxonomy

A number of faults are specific to the two aforementioned means of sharing information between concurrent threads. Faults associated with shared memory communication include data races, deadlocks, livelocks and starvation. Faults associated with message passing are further subdivided into synchronous communication problems, as well as asynchronous faults.

Data races are situations where *unsynchronized thread access ranges* result in unpredictable program states and behavior [Chen 06]. These types of faults are due to accesses to *unprotected resources* (e.g., a shared memory location). Threads may access a shared location as either *readers* or *writers*. Problems then arise due to the order of *execution of threads within the shared resource* [Chen 06]. While unsynchronized access to shared resources is often due to errors on the part of the designer, it may also be on purpose to satisfy performance constraints. In general, three conditions must be met before a data race occurs: 1. Two or more threads access the same memory location concurrently, 2. At least one thread accesses the memory location non-atomically for writing, 3. Access to the memory location is unsynchronized (or is inadequately synchronized). When these three conditions are met, writer and reader threads may execute concurrently within the shared memory, resulting in inconsistent data [Chen 06].

Deadlocks occur when a thread is unable to continue its execution because it is blocked waiting for a lock that is held indefinitely by another thread [Goma 00]. Consider the Resource Allocation Graph

(RAG) - which depicts the allocation of resources to threads [Baco 97] – in Figure 1. Thread T1 locks mutex M1 (as indicated by the solid arrow) and requests access to M2 (as indicated by the dotted arrow). This request places T1 in M2's wait queue because M2 is currently held by T2, which is also requesting access to M1. Neither T1 nor T2 can proceed because each is waiting for the mutex held by the other.

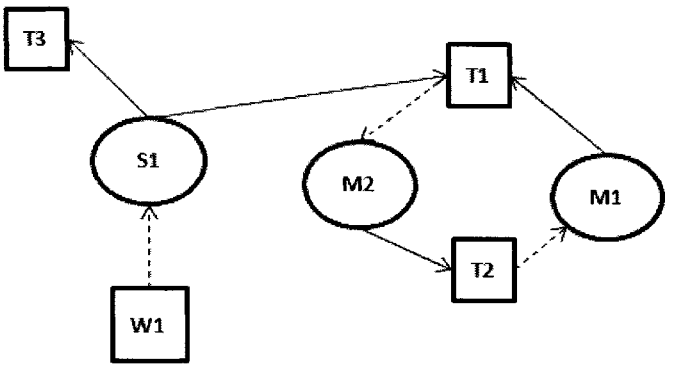

Figure 1: Deadlocks as illustrated on a RAG

In general, four conditions must be true before a deadlock occurs: 1. Threads share or access information that is placed under a lock, 2. Threads acquire a lock while waiting for more locks, 3. Locks are non preemptible, 4. There exists a circular chain of requests and locks (the circularity condition), as identified in a RAG [Baco 97].

Livelocks occur when threads are unable to complete their executions because they are busy responding to each other [Burn 93]. This is often likened to a situation in which two people are approaching from different ends of a hallway. The hallway is wide enough to let two people pass. Person A is approaching from the north and is walking on his or her right side. Person B is approaching from the

south and is walking on his or her left side. When they meet, A moves to his/her left to let B pass, while B simultaneously moves to his/her right to let A pass. This then prompts A to move back to the right and B back to the left, and so forth. Livelocks can also occur when threads are waiting on a condition that will never become true [Baco 97]. Consider for example two threads, T1 and T2. Each has an associated flag, namely flag [T1] and flag [T2], respectively. The following code fragments will likely result in a livelock [Tai 94]:

```
//in T1
while (true){
     flag[T1] = true;
     while (flag[T2]==true){}
     flag[T1]  = false;
}
```

```
//in T2
while (true){
     flag[T2] = true;
     while (flag[T1]==true){}
     flag[T2] = false;
}
```

Thread starvation has generally been defined in two contexts: the operating system context and the concurrency context. From an operating system point of view, thread starvation refers to the inability of threads to access enough CPU cycles to complete their execution [Sinc 02]. In other words, if a thread is not allocated enough CPU cycles, it is not given the chance to complete its execution, hence it starves. This type of starvation is often referred to as CPU starvation [Oaks 04]. It is argued that scheduling algorithms largely influence the potential for CPU starvation [Sinc 02]. In the context of concurrency, starvation is related to locks, and is therefore coined lock starvation [Oaks 04]. Much like CPU starvation, where threads wait for CPU access indefinitely, lock starvation occurs when some threads wait for lock access indefinitely [Oaks 04], making them similar to a deadlock situation. However, unlike deadlocks, only some—not all—threads accessing a lock are at a standstill [Down 05]. Consider, for example, a solution to the reader/writer problem. Readers are allowed to access the criti-

cal section upon arrival, whereas writers are placed in a wait queue if the critical section is locked by at least one thread. Hence, writers can only proceed when there are no readers accessing the critical section. As seen in Figure 1, a situation arises where a writer (W1) arrives when readers (T1 and T3) are executing in the critical section, and hence W1 is placed in the wait queue. Before all readers exit the critical section, more readers arrive and are granted access to the critical section. As long as new readers arrive before the ones in the critical section complete their execution, the writer will never be allowed to access the critical section. Unlike a deadlock situation, some threads are executing (T3), while others (W1) are denied access to the critical section [Down 05].

Thread communication in the form of message passing is vital to the correct execution of concurrent systems. Recall that messages may be sent synchronously or asynchronously. In synchronous message sending, faults can arise in the form of ensured delivery of the message. A synchronous message that is not received by the receiver (either because the receiver has terminated [Tane 08] or because the message is lost [Tane 08]) would result in infinite blocking by the sender. For asynchronous message passing, several problems may arise. When a message cannot be delivered, how does the sender learn about the error, so a resend is issued? Asynchronous message passing usually requires the use of a buffer to store sent messages on the receiver side until the receiver is able to process them [Goma 00]. Several problems may arise here. What happens when the receiver buffer is full and more messages are sent by the sender? A number of strategies exist: 1. The sender can delay its message until the buffer is free. 2. The message could be relayed back to the sender indicating that it cannot be processed because the buffer is full. 3. The sender can block until the receiver processes messages and frees up some of the queue [Tane 08], or 4. The message can be ignored until the queue is freed up by the receiver [Jia 05].

The last approach leads to loss of messages which can deem the system unreliable. With the third approach, when a sender blocks when the message queue is full, deadlocks may occur when another thread floods the receiver buffer with messages. For example, assume threads T1 and T2 communicate together asynchronously, with q1 and q2 the message queues for holding messages sent to T1 and T2 respectively. Assume T1 fills up q2 with sent messages and blocks. Assume that another interface floods T2 with messages that it needs to send to T1. Since T1 is already blocked, it will not be able to process them, letting T2 fill up q1 and blocking when the message queue is full. Because messages are assigned priorities, faults related to priority inversion can also occur, whereby a receiver processes a lower priority message leaving a higher priority one waiting [Dava 92]. Another common problem is how a receiver handles multiple copies of the same message [Jia 05]. Does it treat each request as a different message, processing each repeatedly, or does it execute the message just once and ignore other copies? In the former case, problems arise if only one execution of the message was desired (the multiple copies could be due to a communication error whereby the sender did not receive the response) [Jia 05].

Of all the different faults listed, our approach aims only at detecting data races, deadlocks and lock starvation (hence referred to simply as starvation). CPU starvation lies outside the scope of this work since we are interested in problems that arise independently of the underlying operating system. It is important to note that many real-time operating systems (such as QNX and RT-Linux [Bask 05]) have mechanisms for dealing with deadlocks (such as the priority ceiling protocol). These ensure that a deadlock will not occur. Since we aim at detecting faults in real-time systems, deadlock detection may therefore appear as being unnecessary. However, industrial systems often use their own proprietary

systems; ones that are developed specifically for their applications (such as Google's Android). These operating systems often do not ensure the absence of deadlocks. The problem we address is therefore relevant to those systems that do not enforce mechanisms for dealing with deadlocks.

Table 1: Summary of concepts needed

	Concept	**Properties**						
General Concepts	Lock	acquire	release	capacity	wait queue	wait queue access	thread access ranges of locks	thread execution time within locks
	Thread	priority						
	Thread type	reader	writer					
	Synchronous Message Passing	occurrence						
	Asynchronous Message Passing	occurrence						
	Unprotected resource	Thread access ranges of resource			Thread execution time within resource			
	Fault	**General Concepts Needed**						
Faults	Deadlock	Lock		Thread				
	Starvation	Lock		Thread				
	Data Race	Unprotected resource		Thread	Thread type			

Table 1 summarizes the concepts needed (as had appeared in italics in this section) to detect faults (namely deadlocks, starvation and data races) in our approach. From the first half of the table, the general concepts of lock, thread, thread type, unprotected resource and synchronous/asynchronous message passing are listed, along with the properties needed for each. While thread type can be considered

a property of thread, it is depicted as a different concept to emphasize the idea that the type of thread (whether reader or writer) is not needed for all fault situations. Rather, as shown in the second half of the table, it is only needed for data race detection. For synchronous and asynchronous message passing, occurrence appears as a property because we only need to know whether or not they occur. In the second half of the table, the three faults are listed along with the general concepts that are needed for each. Hence, for deadlock detection, we need the concepts of lock and thread, both of which have properties (appearing in the first half of the table). Thread, for example, has a priority property.

The remaining types of faults discussed in this section cannot be detected at such a high level. For shared memory communication, we do not detect the presence of livelocks as they involve the detection of the progress of threads. Thread progress defines how much of the threads' total execution has completed. Hence, when threads are executing, yet their overall progress does not change, the threads are recognized as being livelocked. This is how most operating systems detect livelocks. However, this information is not available at the modeling level. Only the target application and operating system can recognize livelocks as they can monitor the progress of threads. Faults associated with message passing not only often appear in distributed environments (which is beyond our scope), but are also difficult to recognize at the modeling level. For example, in synchronous message passing, ensured message delivery problems arise during execution. Theoretically, at the modeling level, the message is assumed to be delivered. Upon execution, various other conditions may arise. In general, such problems are often dealt with by various underlying communication protocols. Similarly, communication protocols and the underlying real time operating systems can deal with faults associated with asynchronous messaging.

2.3 Concurrency in UML

Standard UML 2.2 identifies a number of aspects related to concurrency [UML 2]. These are spread across class, sequence, communication and activity diagrams. In the first three types of diagrams, active objects are defined as having their own thread of control. In that sense, each such object can be considered a concurrent thread. UML depicts active objects with two vertical lines on the side of a class as shown in Figure 2.

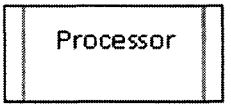

Figure 2: Active class depiction in UML 2.2

Sequence diagrams also introduce the concept of concurrency with the par fragment. Each section within the par fragment executes concurrently, as depicted in Figure 3, where `cookFood()` and `rotateFood()` execute concurrently.

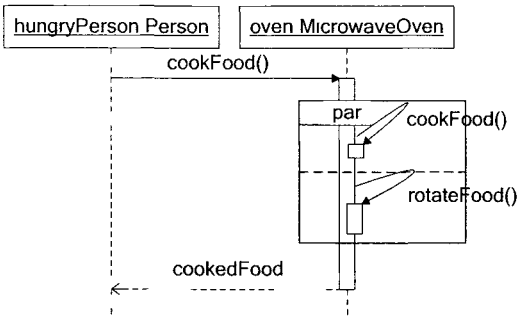

Figure 3: Sequence diagram with par [Bell 04]

Concurrent executions also appear in standard UML activity diagrams and are denoted by forks and joins. In Figure 4, `FillOrder` and `SendInvoice` occur in parallel.

UML further introduces the concepts of synchronous message passing and asynchronous message passing. These are depicted on sequence diagrams with different arrowheads. Synchronous messages have filled arrowheads. The message `cookfood()` in Figure 3 is an example of a synchronous message. Asynchronous messages are depicted with different arrowheads.

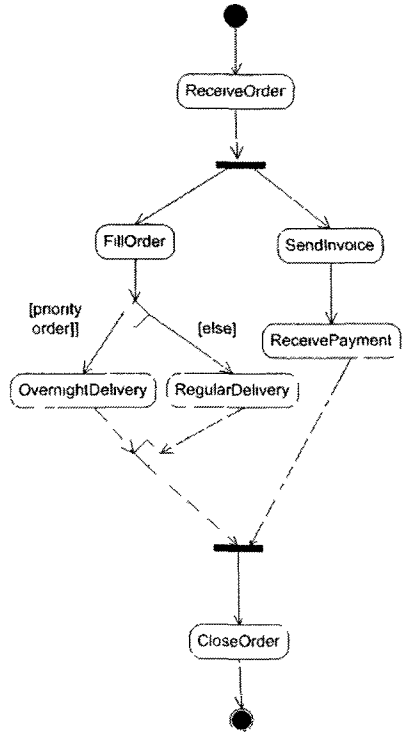

Figure 4: Activity diagram example [Fowl 04]

Beyond these concepts, standard UML cannot depict various other concurrency aspects. For example, there is no standard way to depict locks, along with their various attributes, such as capacity and

queuing policy. This type of information would be inserted as comments in the UML design. The MARTE profile, however, aims at addressing precisely these issues. We delve into the MARTE profile in Section 3.

2.4 Genetic Algorithms

GAs are a means of solving complex optimization problems that are often NP hard [Hou 94] in limited amounts of time [Hong 00, Hou 94, Gold 89]. Optimization problems are those that try to reach the best solution given measurements of the goodness of solutions [Gold 89]. GAs have increasingly been used in the field of search based software engineering [Harm 09]. This may be attributed to the ease with which GAs can be adapted to a given problem [Harm 09].

GAs are based on concepts adopted from genetic and evolutionary theories. They are comprised of several components: a representation of the solution, referred to as the *chromosome*, fitness of each chromosome, referred to as the *fitness* or *objective function*, the genetic operations of *crossover* and *mutation* which generate new offspring, and selection operations which choose offspring fit for survival.

The chromosome models the problem solution. Each element within the chromosome is known as a *gene*. The collection of chromosomes used by the GA is dubbed *population*. Figure 5 illustrates these concepts in terms of representation of the red, green and blue (RGB) makeup of a population of three pixels on a screen. The chromosome in the figure is composed of three genes. A gene represents the red, green, or blue component of a pixel on a screen, respectively. Hence, the chromosome depicts one pixel's RGB makeup. The population portrays the makeup of three pixels on the screen.

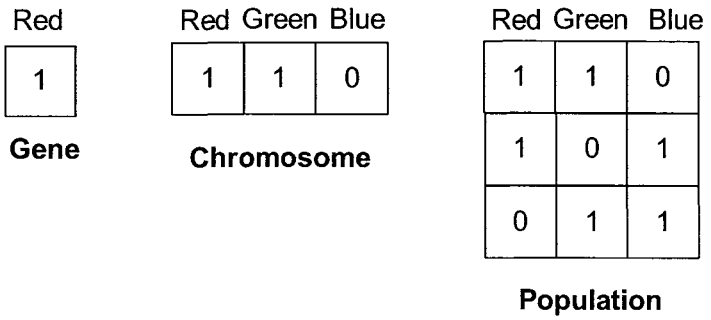

Figure 5: GA chromosome terminology

The quality of a chromosome is its fitness. Fitness defines which chromosomes are closer to the optimal solution. If the optimal solution for the population of Figure 5 is a pixel with only a red component (i.e. a chromosome with RGB values 1, 0, 0), the first two chromosomes of the population would be deemed fitter than the last one. The initial two chromosomes each have one more color component in addition to the red component, while the last chromosome has no red component.

Both crossover and mutation operators are needed to explore the problem search space. Crossover operators generate offspring from two parents based on the merits of each parent, as demonstrated in Figure 6 through single point crossover. Taking the G component as a division point common to both parents, the parents alternate genes with respect to the division point in creating the children. Parent 1 contributes the RB components of Child 1, allowing Parent 2 to contribute the G component. Similarly, Parent 2 contributes the RB components of Child 2, while Parent 1 contributes its G component. Hence, GAs use the notion of survival of the fittest by passing superior traits from one generation to the next.

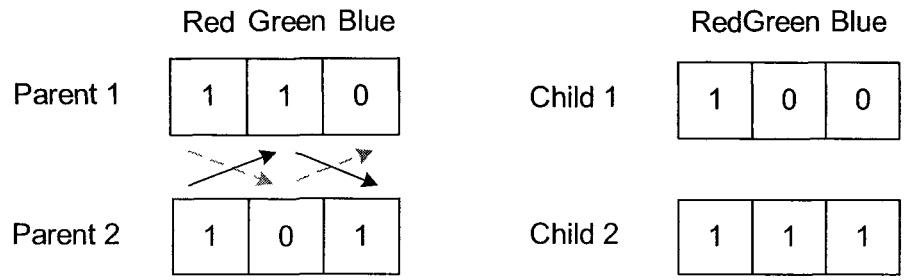

Figure 6: Illustration of crossover concept

Mutation operators mutate, or alter a single chromosome as Figure 7 shows. Mutation introduces new genetic information and aids the GA in avoiding getting stuck in local optima. In the figure, the Red gene is mutated, resulting in a chromosome with RGB values 0, 1, 0 respectively.

The process of selection determines which individuals among the original population, mutated and child chromosomes will survive, hence retaining a constant population size.

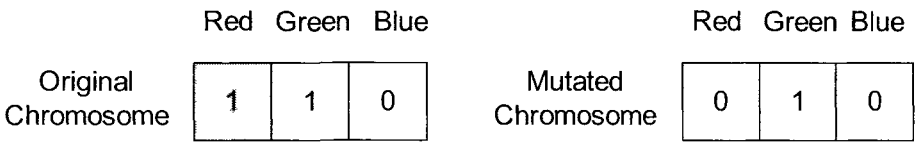

Figure 7: Illustration of mutation concept

As shown in Figure 8, the Genetic Algorithm first randomly creates an initial population of individuals. Working with the population, the genetic algorithm then selects and performs various crossover and mutation operations, creating new chromosomes. The fitness of the new chromosomes is compared to others in the population. Fitter individuals are retained while less fit ones are removed. The

process of crossover, mutation, fitness comparison and replacement (known as a *generation*) continues until the termination criterion, such as a predefined number of generations, is reached [Haup 98].

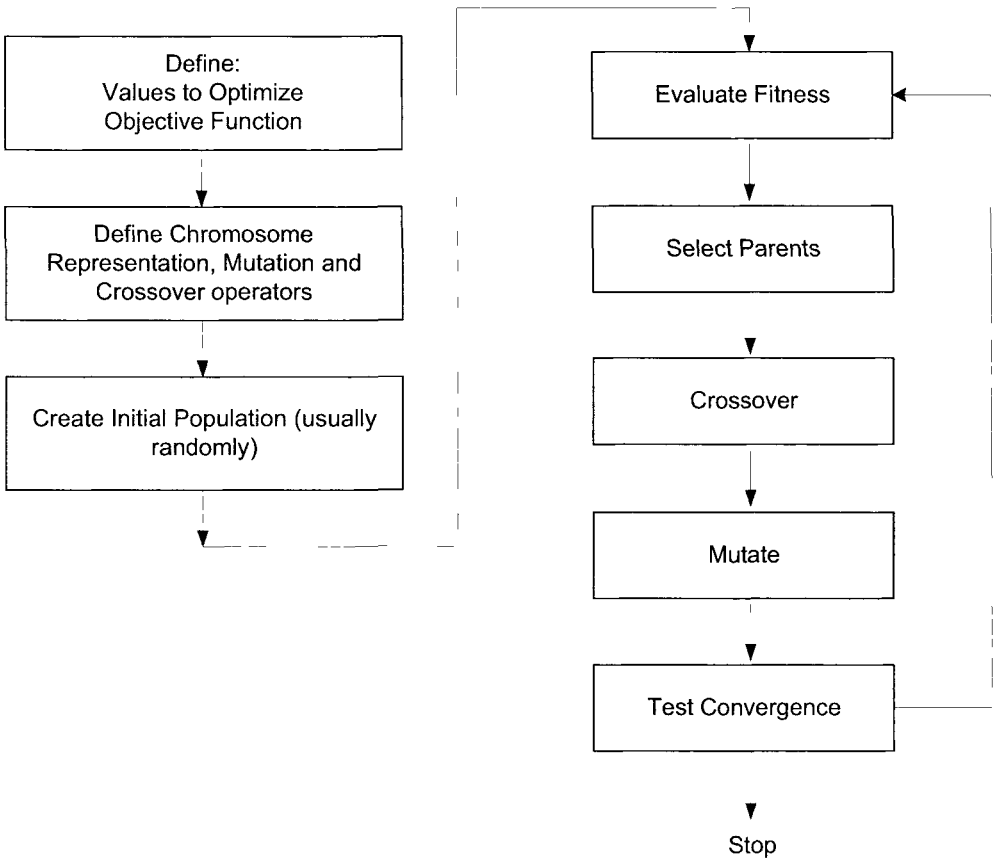

Figure 8: Flow chart of genetic algorithm (Adapted from [Haup 98])

A variety of replacement methodologies are defined for GAs, such as simple, steady state and incremental. Each specifies how much of the population should be replaced with each run or generation of the algorithm. The simple genetic algorithm creates an entirely new population of chromosomes with each generation of the algorithm. The steady state algorithm, on the other hand, uses overlapping popu-

lations, leaving it up to the user to determine the number of chromosomes to replace in each generation. Each generation, the steady state GA produces offspring, storing them in a temporary location. These are then added to the population and the worst individuals are removed such that the population size remains constant. In incremental genetic algorithms, only one or two offspring chromosomes are generated. These are integrated into the population in one of the following ways: replacing the parent, replacing a random individual in the population, or replacing an individual that is similar to the offspring.

3 MARTE PROFILE

Our aim is to use MARTE to support the identification of design issues pertaining to synchronization in real time systems. These issues revolve mainly around various locking mechanisms (such as semaphores), and whether they are appropriately used or not. These design issues underlie various faults. In this thesis, we focus on three types of faults, namely deadlocks, data races and starvation. We detail here how the information required to detect these faults, namely those in italics in Section 2, are supported in MARTE.

MARTE is geared towards both the real-time and embedded system domains. The profile is roughly divided into two sub-divisions: the MARTE design model and the MARTE analysis model. The former models various features of real-time and embedded systems while the latter allows the annotation of models for system analysis purposes. Both sub-divisions are based on a common foundation, the MARTE foundation, which defines time concepts and use of concurrent resources. The real time aspects of systems using the MARTE profile are meant to be part of a larger model. In other words, the profile is not meant to fully model a system. Instead, it is used to model only the real time aspects of a system, with non-critical elements being sufficiently modeled with UML [Jens 09]. To facilitate this, the MARTE profile is modular in structure, allowing users to choose the appropriate subsets needed for their applications [MART 09]. We next describe the aspects of the profile that are relevant to our work. We illustrate these concepts on an online airline reservation system, where a user, represented by a thread named user1, can reserve an airline seat online.

We begin with the foundations of MARTE, namely the Generic Resource Modeling (GRM) package. The GRM package models and manages resources at a high level [Koud 08]. It is in this package (specifically in `GRM::ResourceTypes`) where we find the `<<Acquire>>` and `<<Release>>` stereotypes. These stereotypes model the acquisition and release of resources, respectively. Resources cannot be accessed or released until both actions have been executed successfully. In our approach, locks are modeled as resources. Within the GRM package, the Software Resource Modeling (SRM) sub-package presents mechanisms for designing multithreading applications. SRM is further subdivided into four packages: `SW_ResourceCore` (which contains all the basic resource concepts), `SW_Concurrency` (which contains concurrent execution concepts), `SW_Interaction` (which deals with communication and synchronization resources) and `SW_Brokering` (which deals with resource management). In the `SW_Concurrency` package, concurrently executing entities competing for resources are depicted with the `<<SwConcurrentResource>>` stereotype. As aforementioned in Section 2.3, concurrency is also depicted in standard UML, but `<<SwConcurrentResource>>` enhances concurrent execution modeling due to its associated attributes, such as `priorityElements`, which is used to determine the priority of the associated thread. In our example, user1 would be designated as an `<<SwConcurrentResource>>`, with its priority indicated using `priorityElements`, as shown in Figure 9 on a sequence diagram. Here, user1 has medium priority (the highest priority being 32).

During system execution, resources may be shared by various concurrent actions and hence should ideally be protected. Shared resources in general are identified as `<<SharedDataComResource>>` in the `SRM::SW_Interaction` package; protected resources (i.e. locks) in particular are coined

<<SwMutualExclusionResource>> in the same package. Attributes associated with the latter stereotype include accessTokenElements, which defines lock capacity and waitingQueuePolicy, which defines the access control policy for elements waiting in the wait queue (e.g., FIFO, LIFO, Priority). Seats in the airline reservation system, as shown in Figure 9, are stereotyped as <<SwMutualExclusionResource>>, whereby only one user at a time can access the seat for booking. A relationship of <<Acquire>> and <<Release>> is present between the seats and user1. Each seat additionally defines its number of accessTokenElements as one, with a priority waitingQueuePolicy (an airline agent has higher priority than online users when reserving seats).

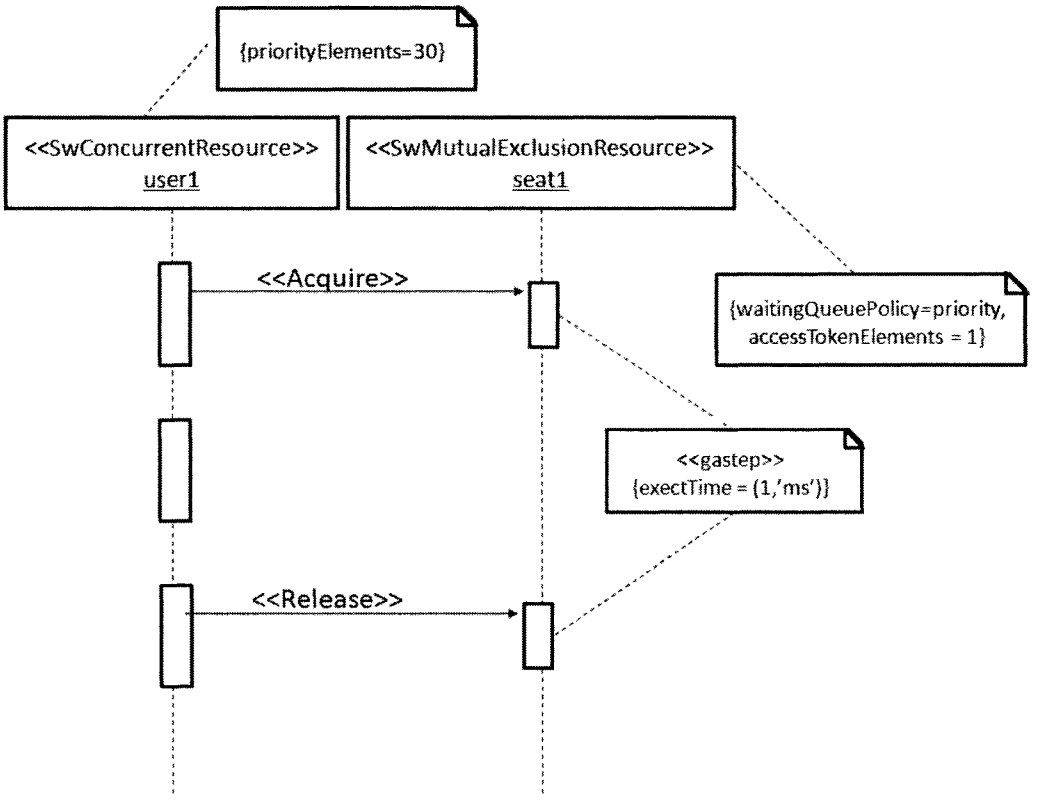

Figure 9: Airline reservation example

The High-Level Application Modeling (HLAM) sub-package introduces `<<RtService>>`, a specialized service with specific real-time constraints. It contains several attributes. A particular attribute, `concPolicy`, can be used to determine the type of concurrency policy used for the real-time service. Defined types include `reader` and `writer`.

The concept of time is vital in real time systems. Real time systems by definition are reactive systems whose correct application depends not only on the outputs, but also on the time delay during which these outputs are computed [Gama 10]. The issue of how to deal with timing information within such systems is often determined by the design of the system. In general, the representation of time can be classified into three categories: asynchronous, pre-estimated and synchronous [Gama 10]. Asynchronous time is dependent on the underlying operating system on which the implemented real time system will eventually run. In this model of time, the execution times of threads are not known in advance, hence deadlines are defined for these threads [Gama 10]. In the pre-estimated notion of time, execution times are known in advance, as is the case with timed automata, for example [Gama 10]. In the synchronous model, much of the temporal quantitative aspects of the system are abstracted away. Instead, it is assumed that the system will operate on the inputs immediately upon their initiation, and will produce the desired outputs within the required timing requirements. The advantage of such a view is that it allows one to focus only on the functional aspects of the system [Gama 10].

Since we aim at targeting real time systems in general, we adopt the asynchronous (also called physical or real-time [MART 09]) view of time. MARTE supports this view of time in several aspects. Because time is considered a performance and schedulability issue, MARTE provides several stereotypes (as

shown below) that allow for specification of various timing constraints. The Time sub-package also provides some aspects of asynchronous time, as shown below.

The Generic Quantitative Analysis Modeling (GQAM) sub-package contains aspects pertaining to the non-functional properties of real time systems. It defines stereotype `<<saStep>>` (that extends stereotype `<<gaStep>>`) which is used when decisions about the allocation of system resources is made. Its tags include `priority` (the priority of the action on the host processor), `interOccTime` (interval between multiple initiations of the action), and `execTime` (the execution time of the action). Execution times can be specified as maximum and minimum time ranges. For the airline system in Figure 9, the summation of `execTimes` designates the amount of time taken to book a seat.

The Time sub-package models various time accesses and structures, such as clock times, continuous times, etc. [Koud 08]. The `Time::TimedConstraints` package introduces the `<<TimedConstraint>>` stereotype which models a constraint that is imposed on the occurrence of a particular event.

This overview of MARTE illustrates that the input needed for the detection of faults, i.e., the concepts related to deadlock, data races and starvation presented in Table 1, can be retrieved from a UML/MARTE design model. The mapping between those concepts and the profile is summarized in Table 2. The type of message passing, whether synchronous or asynchronous, is derived from the UML representation rather than the MARTE profile representation. It is important to note that this mapping, illustrating how MARTE stereotypes and tags can be used to model real-time concepts, is consistent with how others (such as [Koud 08, Jens 09, Faug 07]) have used the MARTE profile. The

concepts used may vary over time because MARTE is an evolving profile. As such, it is constantly being updated by OMG to meet varying user demands. Table 2 also shows the type of fault each concept is used in.

Table 2: Concept to MARTE mapping

Concept	MARTE Stereotype/Tag	MARTE sub-profile	Type of Fault
Thread	<<SwConcurrentResource>>	SRM::SW_Concurrency	All
Lock	<<SwMututalExclusionResource>>	SRM::SW_Interaction	Deadlock, starvation
Lock acquire	<<Acquire>>	GRM::ResourceTypes	Deadlock, starvation
Lock release	<<Release>>	GRM::ResourceTypes	Deadlock, starvation
Wait queue access policy	<<SwMutualExclusionResource>>/ waitingQueuePolicy	SRM::SW_Interaction	Deadlock, starvation
Thread priority	<<SwConcurrentResource>>/ priorityElements	SRM::SW_Concurrency	Deadlock, starvation
Thread execution times within locks	<<gaStep>>/execTime	GQAM::GQAM_Workload	Deadlock, starvation
Lock capacity	<<SwMutualExclusionResource>>/ accessTokenElements	SRM::SW_Interaction	Deadlock, starvation
Thread access range of locks	<<gaStep>>/interOccTime | <<gaStep>>/execTime	GQAM::GQAM_Workload	Deadlock, starvation
Unprotected resource	<<SharedDataComResource>>	SRM::SW_Interaction	Data race
Reader thread	<<RtService>>/concPolicy = reader	HLAM	Data race
Writer thread	<<RtService>>/concPolicy = writer	HLAM	Data race
Thread exec. time in shared resource	<<gaStep>>/execTime	GQAM:: GQAM_Workload	Data race
Thread access range of shared resource	<<gaStep>>/interOccTime| <<gaStep>>/execTime	GQAM:: GQAM_Workload	Data race
Time constraints	<<TimedConstraint>>	Time::TimedConstraints	All

4 RELATED WORK

Our approach spans several fields of research, namely verification of concurrent systems (in terms of deadlocks, starvation and data races), search-based software verification, as well as the use of UML profiles for concurrency. Indeed, our verification approach can be considered a combination of three aspects: (1) It is based on using design information from UML models, (2) it focuses on non-functional, concurrency aspects, and (3) it makes use of search techniques to identify our targeted faults. Model checking has predominantly been used to address design-based, non-functional verification but has not (exclusively) relied on information captured by UML models. In addition, other works are related to ours in that they cover one or more of the three aspects mentioned above. For the sake of completeness, we also discuss other works that use the MARTE profile. We present these works in the following order: model checking (Section 4.1), search-based, non-functional verification (Section 4.2), and uses of MARTE (Section 4.3). We end this section with a brief summary of how our approach differs from other works.

4.1 Related Work: Model Checking

Essentially, model checking has the same general aim as our approach though it is based on different requirements: using system models to automatically detect whether a given system meets its specifications in terms of safety (properties that should be restricted to a set of known behaviors, such that nothing "bad" happens in the system [Kupf 01]), concurrency, or other important properties [Clar 00]. We do not aim at outperforming model checkers. Instead, we aim at extending the use of model-based verification to UML-based development in a practical fashion. Model checking properties are normally

expressed in a form of temporal logic [Clar 00]. Hence, they are not easily adopted by the many users unfamiliar with and reluctant to use temporal logic. The approach we propose is meant to be used in the context of the OMG's MDA, hence our reliance on the UML standard and the MARTE profile for modeling real-time, concurrency information. One possible method of achieving this is by combining model checking and our UML based input through transformations. In other words, the UML/MARTE model input our approach relies on can be translated into a form of temporal logic (e.g., CTL, LTL), which can then be used by various model checkers. However, there is no obvious translation from UML/MARTE to temporal logic. The information required is not entirely present in the sequence diagrams we use, as they might be problem specific and require information about the states of the system. For example, to specify - in temporal logic - that no deadlock should be present in the famous dining philosophers problem (see Section 7.1.1 for further details), one possible formulation would be: $\Box(\neg(l_0 \wedge l_1 \wedge l_2) \wedge \neg(r_0 \wedge r_1 \wedge r_2))$. Assuming there are three philosophers, the temporal logic formulation states that it is always the case that philosophers one, two and three are not holding their left forks $\neg(l_0, l_1$, and $l_2)$ and philosophers one, two and three are not holding their right forks $\neg$ $(r_0, r_1$, and $r_2)$. Hence, one needs to model the various states of the system, i.e., the fact that a philosopher holds a left or right fork, which is information not readily available in the UML/MARTE model. We must therefore rely on a different means, namely heuristic search algorithms to detect concurrency faults. This can be considered a drawback in so far as the approach not guaranteeing detection of faults if they occur. But the advantages of using our approach are two-fold: 1) *Familiarity with UML*: Using extended UML diagrams to detect concurrency faults is easier for designers already working with UML. On the one hand, one may consider that diagrams required by our approach are more detailed than

those required when the system is initially designed. On the other hand, details on timing of events, estimated task execution times, and so on, would anyway be identified when designing real-time, concurrent systems [Goma 00], and adding such details to UML diagrams would be natural in a MDA process. Furthermore, adding information to pre-existing diagrams for verification purposes is probably easier than working with a different, unfamiliar model. 2) *Design model use*: Existing design models can be used again for verification purposes, rather than developing different models for verification. Our approach is thus an alternative to model checking; one that can more easily be adopted in circumstances where UML is already prevalent.

Some model checkers, such as the Java Path Finder [Brat 00], aim at detecting data races, while others are geared more towards deadlocks and starvation detection. Model checking techniques can further be categorized according to the source of input information as well as various search techniques used. In terms of the source of input information, some model checkers use models of the system under test while others use the system's source code: UPPAAL [Behr 04] uses a network of timed-automata, and properties to verify are expressed via UPPAAL's query language, which uses a version of Computation Tree Logic (CTL) [Behr 04]; SPIN [Holz 97] uses automata (expressed with the Process Meta Language (Promela)) and properties to be verified are expressed in Linear Temporal Logic (LTL) [Holz 97]. In [Gagn 08, Knap 07, Schn 99] transformations from UML to other intermediate languages are used, before being inputted to model checkers. Properties to be checked by the model checkers are specified in temporal logic. For example, the work in [Gagn 08] transforms UML state, class and communication diagrams into Maude specifications which are then fed to a model checker, where

properties to be verified are defined in LTL. Other model checkers, such as Verisoft [Gode 04], rely on source code analysis to search for error states.

Model checkers use various search techniques: exhaustive search (SPIN [Holz 97]); graph exploration algorithms - such as depth first, breadth first and A* search techniques - constructing only relevant parts of the search space (HSF-SPIN [Edel 01], DELFIN+ [Grad 06]); heuristic searches such as GAs (Verisoft [Gode 04]) or Ant Colony optimization [Alba 07].

Direct, quantitative comparisons with the various model checking techniques we have encountered were not possible as some works, such as [Edel 01], did not provide enough details in the case studies to enable meaningful comparisons. The only work that we came across where results were clearly reported was that described in [Gode 04]. For others, such as [Grad 06], the tools used were not readily available so we could not run them for our case studies on the same hardware. More importantly, our aim is not to provide a technique that is better at detecting concurrency faults than model checking, but rather an alternative that is more practical in the context of UML MDA development. So, comparisons with the above techniques, though interesting, are not an absolute necessity to demonstrate the value of our work.

4.2 Related Work: Search-Based, Non-Functional Verification

Search-based engineering is a term referring to the use of various meta-heuristic techniques to solve a multitude of software engineering problems. The essence of all such problems is optimization. Because these problems are normally very complex in nature and cannot be addressed by exact operations re-

search techniques (such as linear programming), meta-heuristic techniques (such as genetic algorithms, tabu search and simulated annealing) have abounded.

Current trends in the field of search-based software engineering are wide and varied. They cover almost all aspects of the development life cycle: requirements engineering, planning, testing, maintenance and quality assessment [Harm 07]. In [Harm 09], search-based software engineering is categorized into seven main divisions: Requirements/Specifications, Design Tools and Techniques, Software / Program Verification and Model Checking, Testing and Debugging, Distribution / Maintenance and Enhancement, Metrics, and Management. The approach we propose falls under the Software / Program Verification and Model Checking category. Hence it is a form of verification or testing. Using the taxonomy presented in [Afza 09], our approach is further refined as a search-based, non-functional testing approach, which can further be refined by goal into primarily five categories: usability, safety, execution time, buffer overflow and quality of service. Safety testing searches for inputs that violate a safety property and is probably the closest category to our work. Safety properties are ones that should be restricted to a set of known behaviors, such that nothing "bad" happens in the system [Kupf 01]. GAs and simulated annealing have both been used to generate inputs and sequences of inputs that aim at violating a safety property [Afza 09]. There is no conclusive way to determine the most suitable optimization technique for a given problem [Harm 09]. However, due to their ease of adaptation, GAs have often been used [Harm 09]. The initial setup of the GA can be rather cumbersome, requiring the specification of various parameters (such as population size and selection operator). There does exist a large body of knowledge aiding in that aspect, determining the best parameters to use, based on experimentation. (Later in this thesis we tailor a genetic algorithm and rely on this body of knowledge to set

parameters.) In [Trac 99], software fault tree analysis is used: a safety property is assumed to be violated at a certain statement within the system's code, then working backwards, the set of inputs that lead to this violation are determined via a meta-heuristic. Somewhat similarly in [Abde 01], the system under test is executed and observed as to whether or not a safety property is violated. This is done in terms of stepwise construction of test scenarios whereby each step explores the continuation of the previous step where the property is violated [Abde 01].

Also in terms of verifying concurrent systems, design-based verification works aim at uncovering deadlocks and starvation as well as data races. One approach for deadlock and starvation detection, presented in [Kim 05], is to build a model - in the form of a UML state machine - that captures both the aspects of the system's behavior that one wants to verify, as well as the underlying programming language concurrency mechanisms (the authors specifically target Java). The Symbolic Analysis Laboratory model checker derives test sequences from the model, which are then executed with a deterministic, run-to-completion testing tool [Kim 05]. A similar work is presented by Lei, Wang and Li [Lei 08]. Here, the authors use a model-based approach for the detection of data races. Data races are identified by checking the state transitions of shared resources at runtime. The corresponding test scenarios leading to the race are then identified using UML activity diagrams extended with data operation tags. This extension is necessary as UML activity diagrams provide no means to model data sharing. Hence, the authors extend them with stereotypes to depict data sharing. The extended UML diagrams can then serve as an oracle for verifying execution traces [Lei 08]. They also serve to ensure that both code and design are consistent. Both works differ from our approach as our goal is to reuse exist-

ing UML design models rather than to create UML models specifically for testing language specific implementations.

Baker et al. describe an approach and tool for the detection of faults in UML sequence diagrams [Bake 05]. Their approach examines sequence diagram messages to determine the presence of blocking conditions (threads waiting for a particular message yet receive another), non-local choices and ordering (behavior of one thread depends on the run-time behavior of another thread not observable by the initial thread) and false-underspecification (events on a lifeline are unordered but an ordering actually exists when considering the complete specification). Hence, the authors target semantic faults that may be present in sequence diagrams [Bake 05]. Our approach, however, targets non-functional faults. The two problems are complementary: a sequence diagram may be free of false-underspecification, for example, yet it could still exhibit deadlocks, starvation or data race faults.

Other works also aim at verifying concurrent systems in the context of detecting data races. Some of them [Flan 00, Flan 02, Abad 06, Kahl 07] do so using the code of the system under test. The work by Kahlon et al. in [Kahl 07] begins by statically detecting the presence of shared variables in the code, before proceeding to output warnings about the presence of data races. Chugh et al. [Chug 08] also use a form of static analysis. They use program code to develop a data flow analysis for the system under test. They combine this with a race detection engine to obtain a data flow analysis that is suitable for concurrent threads. Both approaches necessitate putting off the detection of data races until the system under test is implemented. This has the disadvantage that any data races that are found due to design faults may be very costly to fix. Furthermore, data races due to dynamically allocated shared resources

might go undetected. Other works, such as Savage et al. [Sava 97], tackle this point. They also use system code in their Eraser tool, but do so dynamically (at run time). In so doing, they ensure that dynamically allocated shared variables involved in data races are also detected. There are limitations to their technique, however, the most important of which is that they are limited to examining paths that are triggered by their test cases. If the test cases chosen are not sufficient to visit a particular path where data races occur, the data race will remain undetected.

4.3 Related Work: Uses of MARTE

In the context of using UML profiles for concurrency, a number of works use the MARTE profile [Mrad 08, Dema 08, Peni 10, Vida 09, Pera 08]. In [Mrad 08], the profile is used to create an approach for real-time embedded system modeling along with transformations to execute those models. In [Vida 09], the authors develop a methodology for designing embedded systems. Their methodology develops systems in three modeling levels: abstract, execution and detailed. They use the MARTE profile to enhance their models [Vida 09]. In [Dema 08], the authors aim at probing the capabilities of MARTE by applying it to a case study, while in [Peni 10], the aim is using MARTE to generate executable specification in SystemC. In [Pera 08], the authors target the automotive domain. Their approach models such systems based on separation of concerns between the software model and the execution platform. Their approach is also used for schedulability analysis to determine whether or not threads are schedulable [Pera 08].

Other works use other UML profiles for verification. Cambroner, Valero and Diaz use the Schedulability, Performance and Time (SPT) profile to ensure freedom from deadlocks, mutual exclusion and tim-

ing constraints [Camb 10]. The authors transform sequence diagrams extended with the SPT profile into timed automata, which are then input to UPPAAL. System specifications that are to be verified are input via UPPAAL's query language, which uses a version of CTL [Camb 10]. The Performance by Unified Model Analysis (PUMA) project, also uses the SPT profile, but is geared towards performance modeling [PUMA 02]. The project aims at estimating various performance analyses, such as the number of events per a system's response, estimating resource loads and bottlenecks, estimating response times and throughputs [PUMA 02].

4.4 Related Work: Differences of Proposed Approach

None of the above works aim at developing a scalable, automated approach that can be easily integrated into an MDA development approach, as we set out to do. The advantages of our model analysis approach include design model reuse whereby existing design models can be reused for verification purposes. Furthermore, familiarity of designers with UML brings another advantage: Using extended UML diagrams to detect concurrency faults is easier for designers already working with UML. While model checking is perhaps the closest to our aim, it does so in a different context by employing different types of models as well as temporal logic. Works using transformations from UML to other intermediate languages before using model checkers also need to specify the properties to be checked in temporal logic. The transformations may also be a potential, practical drawback as the original model undergoes multiple transformations: from UML to an intermediate language and from that intermediate language to some form of automaton. Such transformations tend to add overhead and complexity. They may also introduce scalability issues. Our approach has the added advantage that the verification process is easily integrated into the UML modeling environment, for example as a plug-in.

In the next sections, we present how we achieve our aim of developing a verification technique focused on concurrency, using only UML designs as inputs, and applying search-based techniques to find faults.

5 APPROACH

Recall that the aim of our approach is the detection of concurrency faults - namely deadlocks, starvation and data races - based on design models expressed in UML extended with MARTE. This is achieved using an annotated front end model of the system under test coupled with a backend based on tailored GAs.

In Section 3, we have described the front end, showing how to define aspects needed to detect these three faults using MARTE on sequence diagrams. In this way, the first of our three aims (demonstrating the feasibility of extracting all relevant concurrency information from UML/MARTE design diagrams) is achieved. Information is then extracted from the sequence diagrams and fed to the appropriate GA, depending on the type of fault the designer is interested in uncovering. This is done to fulfill our second aim, showing the effectiveness of the proposed search based technique in detecting concurrency faults. Although extraction from sequence diagrams is currently manually performed, we present a proof of concept in Appendix A, showing how to automate extraction.

The remainder of this section is divided as follows: Section 5.1 presents the associated genetic algorithm for deadlock detection, while sections 5.2 and 5.3 present the GAs for starvation and data races, respectively.

5.1 Deadlock Detection

In uncovering deadlocks, we aim at trying to find a sequence of thread execution interleavings that will result in the worst possible scenario of thread executions—namely deadlocks. Hence, the values to be

optimized to try to reach a deadlock situation are the particular access times of threads to locks. For this, we use a GA. We next introduce the various constituting components of our GA. It is important to note that these components are applicable to any concurrent system. In other words, the GA components are only defined once. What varies from system to system are the input values.

5.1.1 Chromosome Representation

Recall that a chromosome is composed of genes and models a solution to the optimization problem. The collection of chromosomes used by the GA is dubbed population. We want to optimize the particular access times of threads to locks. These are the values that will be altered by the GA to try to reach a deadlock situation. In other words, the chromosome only contains information that will be manipulated by the GA during its search. The access times must reflect schedulable scenarios. In other words, we need to ensure that all execution sequences represented by chromosomes are schedulable [Goma 00]. This entails meeting system specifications of periods, minimum arrival times, etc. Thus, we need to encode threads, locks and access ranges, which are available in the input model: `<<SwConcurrentResource>>`, `<<SwMutualExclusionResource>>`, and `<<gaStep>> / interOccTime` or `<<gaStep>> / execTime`, respectively (Table 2). Thread/Lock associations are determined from the MARTE input using the `<<Acquire>>` and `<<Release>>` stereotypes. Any time constraint (`<<TimedConstraint>>`) is also accounted for in the chromosome; that is chromosomes are created so that constraints are satisfied.

Note that when the initial population is first randomly generated, this is done within the allowable system specifications. As a result of mutation and crossover these specifications may no longer be met by chromosomes, which is why we need to introduce a repair operation (Sections 5.1.2 and 5.1.3).

A gene can be depicted as a 3-tuple (T, L, a), where T is a thread, L is a lock, and a is a specific time unit when T accesses L. We refer to this value as *access time*. Note that access time is distinct from, but related to *thread access range of locks* (Table 2). Access time is a specific time unit chosen within the acceptable thread access range of locks. Overall, a tuple represents the execution of a thread when trying to access a lock. Tuples are defined for a user specified time interval during which the test engineer wants to study the system's behavior. Ideally, to be able to detect deadlocks, the time interval should start with the start of the system under test and end with it. However, most concurrent systems are embedded in real-time applications that are constantly running (e.g., control systems). Furthermore, such verification would be rather costly to perform for most systems. Alternatively, a reduced time interval can be used, following some user-defined heuristic based on the available resources for verification. The heuristic depends on the amount of time that can be spent on each GA run. This is determined by time availability and practical considerations. Designers can choose an initial time interval (e.g., [0-100]), use it to run a few generations, then decide based on these runs an appropriate time interval that fits their time budgets. The time interval controls how many times each thread can access each lock. An increase in the time interval can lead to the increase in the number of times each thread can access a lock, depending on the thread access range of locks. It can also lead to an increase in the number of threads accessing locks. In both scenarios, more tuples/genes are added to the chromosome, thereby increasing its size. An increase in size ultimately increases verification time. Hence, a balance

should be achieved when determining the time interval: it should be long enough for deadlock to occur, but not too long such that it unnecessarily increases verification time. Furthermore, without specifying a time interval, we cannot assume a fixed size for chromosomes, thus making crossover operations much more complex [Jaco 01].

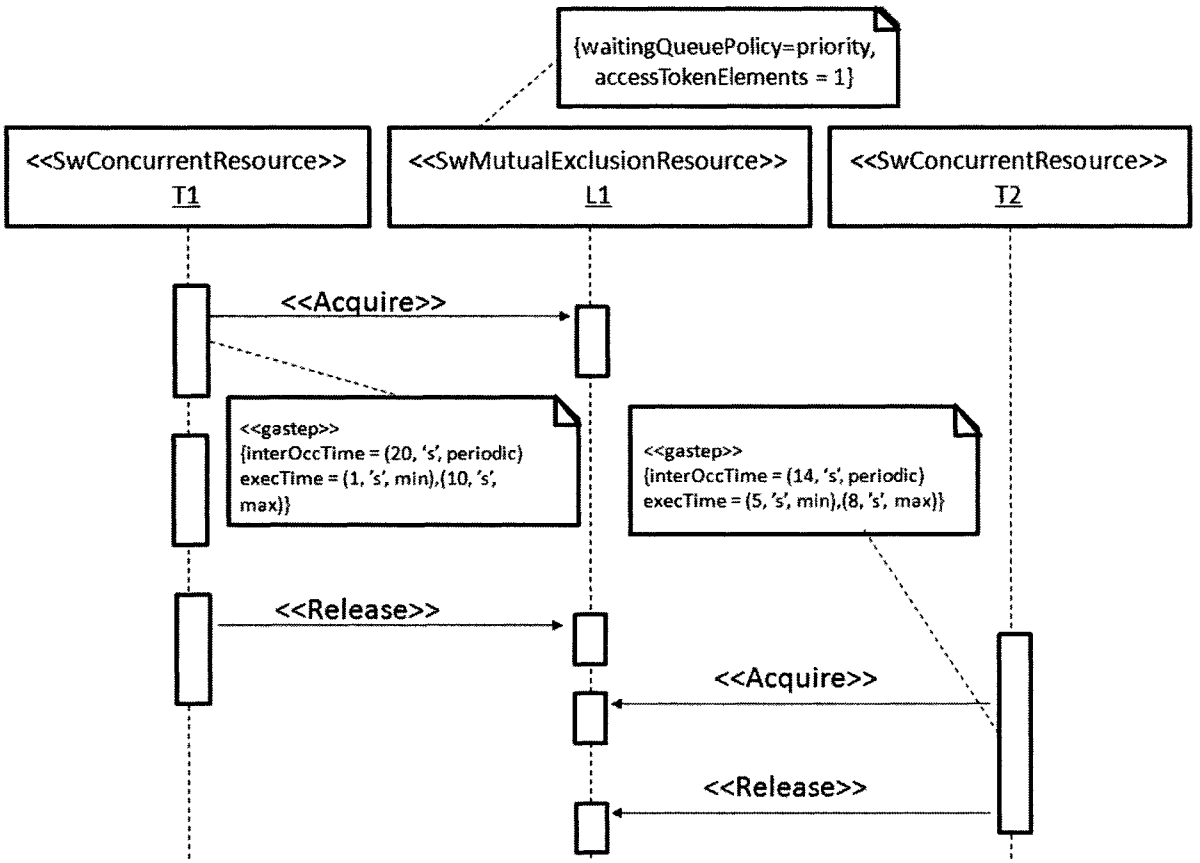

Figure 10: Extracting information for chromosomes

Because a chromosome models a solution to the optimization problem, it needs to be large enough to model all schedulable scenarios during the time interval. Hence, the chromosome size (its number of genes) is equal to the total number of times all threads attempt to access all locks in the given time in-

terval. A thread can appear more than once in the chromosome if it accesses a lock multiple times. A special value of -1 is used to depict lock access times that lie outside this interval: (T, L, -1) represents a lock access that does not occur.

Consider, for example, the sequence diagram of Figure 10. Here, two threads access a lock. From the <<Acquire>> and <<Release>> stereotypes, we can formulate part of the chromosome to express a solution to the problem: (T1, L1, -) (T2, L1, -). Because T1 accesses the resource periodically every 20 seconds and its execution before it accesses the resource takes between 1 and 10 seconds, the range of times it can therefore access the resource are between time units [1-10], [21-30], [41-50], etc. Likewise, T2 can access L1 between [5-8], [19-22], [33-35], etc. Let us assume that the chosen time interval is [1-30]. This means that both T1 and T2 can access L1 at most twice. Thus, the size of the chromosome is four: two genes depicting T1's access and two genes depicting T2's access. Access times for all genes are randomly chosen such that they fall within the acceptable access ranges: (T1, L1, 3) (T1, L1, 27) (T2, L1, 7) (T2, L1, 19).

Three constraints - which we refer to collectively as timing constraints - must be met for the formation of valid chromosomes and to simplify the crossover operation discussed below. 1.) All genes within the chromosome are ordered according to increasing thread identifiers, then lock identifiers, then increasing access times. 2.) Lock access times must fall within the specified time interval or are set to -1. 3.) Consecutive genes for the same thread and lock identifiers must have access time differences equal to at least the minimum and at most the maximum access range of the associated thread and lock, if start and end times are defined as ranges (Section 3).

Consider, for example, a set of three threads T1 (thread access range of lock is [23-25] time units), T2 (thread access range of lock is [15-22]) and T3 (thread access range of lock is [25-35]) each accessing a lock L1. In a time interval of [0-30] time units, the chromosome length would be three since each of the threads can access L1 at most once during this time interval. The following is then a valid chromosome: (T1,L1,24) (T2,L1,20) (T3,L1,-1) where T1 accesses L1 at time unit 24, hence its access time is 24, T2's access time is 20 and T3 does not access the lock before time 30.

5.1.2 Crossover Operator

Crossover is the means by which desirable traits are passed on from parents to their offspring [Haup 98]. We use a one-point crossover operator where two parents are randomly split at the same location into two parts which are then alternated to produce two children.

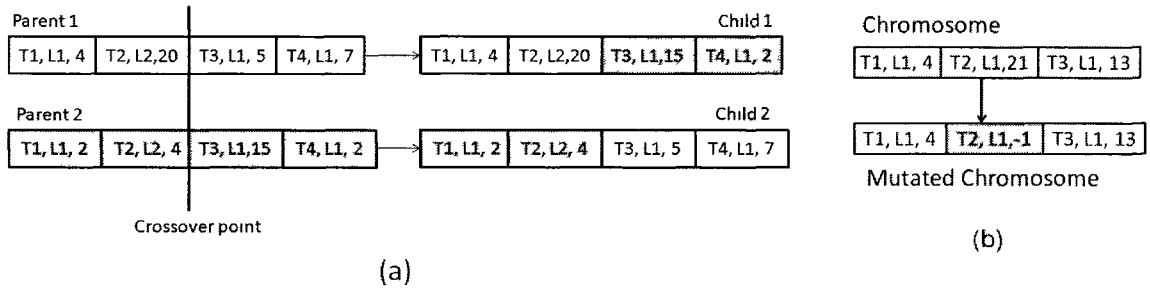

Figure 11: (a) Crossover and (b) mutation examples

For example in Figure 11.a, the two parents on the left produce the offspring on the right. If, after crossover, any two consecutive genes of the same thread and lock no longer meet their lock access time requirements (constraint 3 is violated), the second gene's access time is randomly changed such that constraint 3 is met. Hence, a value from the set of possible access

times is randomly chosen (using a uniform probability distribution) to replace the second gene's access time. This is repeated until all occurrences of this situation satisfy constraint 3.

5.1.3 Mutation Operator

Mutation introduces new genetic information, hence further exploring the search space, while aiding the GA in avoiding getting caught in local optima [Haup 98]. Mutation proceeds as follows: each gene in the chromosome is mutated based on a mutation probability and the resulting chromosome is evaluated for its new fitness. Each gene has an equal mutation probability. Our mutation operator mutates a gene by altering its access time. The rationale is to move access times along the specified time interval, with the aim of finding the optimal times at which these access times will be more likely to cause deadlocks. When a gene is chosen for mutation, a new timing value is randomly chosen (using uniform distribution) from the range of possible access range values. If the value chosen lies outside the time interval, the timing information is set to -1 to satisfy constraint 2. Similar to the crossover operator, if, after mutation, two consecutive genes no longer meet their lock access time requirements, the affected genes are altered such that the requirements are met. For example, assume threads T1, T2 and T3 attempt to access lock L1 with access times [4-7], [20-30], [12-15], respectively, and the time interval is [0-25]. The original chromosome of Figure 11.b is then valid. When the second gene of the chromosome is chosen for mutation (second chromosome in Figure 11.b), a new value (say, 27) is chosen from its access time range [20-30]. Because this falls outside the time interval specified, the mutated gene is set to -1.

5.1.4 Objective Function

The objective function calculates the fitness of chromosomes. In our methodology, we define an objective function that is based on how close a given chromosome is to representing a deadlock. Yet, a deadlock can only be determined after threads acquire their needed locks. We thus need to schedule thread executions according to chromosome data for the time interval defined, and examine the state of threads and locks at the end of the time interval. Hence, our approach incorporates a scheduler. The scheduler uses information available from the MARTE inputs as well as individual chromosomes to map thread accesses of resources. For example, assume the fitness value of the following chromosome is to be determined in a time interval of [1-5]: (T1, L1, 2) (T2, L1, 4). Also assume that, according to the MARTE input, T1 executes for 1 time unit in L1 and T2 executes for 1 time unit. The scheduler is then able to determine the states of the threads for every time unit of the interval. At time unit 1, no threads access the lock. At time unit 2, T1 gains access to L2 and executes. At time unit 3, T1 has completed its execution and the lock is again empty. At time unit 4, T2 gains access to the lock and executes within it for 1 time unit, before releasing it at time unit 5, the end of the time interval.

Since a deadlock appears when the involved threads cannot proceed with their executions, any deadlock that occurs early during the time interval will propagate to the end of the interval; the objective function is therefore used at the end of the time interval. We define the following objective function for deadlock detection:

$$f(c)=\begin{cases} \#LockExecs + threadsWaiting & \text{if } threadsWaiting < 2 \\ \#LockExecs + threadsWaiting + lockCapacities & \text{if } threadsWaiting \geq 2 \end{cases} \quad (1)$$

`threadsWaiting` is the total number of threads waiting on any lock. By definition, a thread waiting on a lock is blocked and its execution cannot resume until it gains access to the lock. This variable is in the range [0-`#T`], where `#T` is the total number of threads in the system. `#LockExecs` is the total number of threads executing within all locks. It is the summation of the slots in all locks that are occupied. This variable is in the range [0-`lockCapacities`], where `lockCapacities` is the summation of all lock capacities. `threadsWaiting` and `#LocksExecs` are obtained after scheduling and are calculated at the end of the time interval, whereas `lockCapacities` comes from the UML/MARTE input (Table 2). This fitness function also gives higher fitness to fitter individuals since high values are obtained in situations where more threads are executing and waiting on more locks. For example, recall the example of Figure 1, reproduced in Figure 12 below for convenience. There are three threads waiting for access locks (T1, T2, W1), hence threadsWaiting>=2. Assuming that S1 has capacity 5, M1 and M2 capacities 1 each, then f(c) = 4 + 3 + 7 = 14.

Some of the input information required for the detection of faults as retrieved from the MARTE profile is used to evaluate the fitness function, namely the lock capacities. Other information obtained from the MARTE input is used by the scheduler. The scheduler uses thread priorities in conjunction with the wait queue access policy to determine the order of thread accesses to locks. The scheduler also uses thread execution times within locks to determine how long each thread executes within an associated lock, as well as lock capacity to determine how many threads can concurrently access the lock. The scheduler also uses the chromosome to determine when a thread actually requires access of a lock. The scheduler also makes use of other information, such as type of communication (synchronous or asynchronous, as depicted in Figure 10, for example) in determining a schedule. Such information does not

affect the applicability of our approach. Instead, the type of communication will affect the scheduler as it will use the type of communication to determine the various states of thread accesses to shared resources.

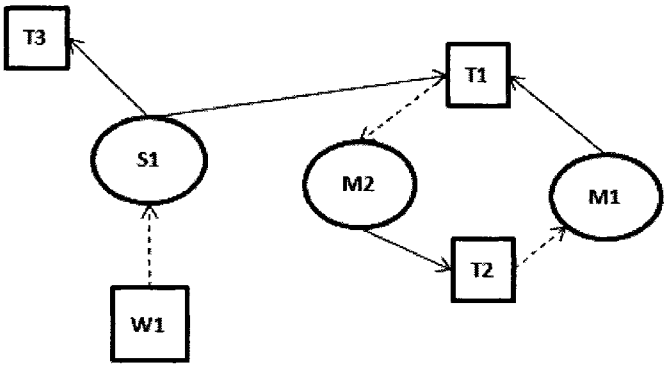

Figure 12: Deadlock as illustrated in a RAG

It is important to note that when there is a deadlock situation, the fitness function does not guarantee that the fitness value will be maximized. For example, near deadlock situations (where `threadsWaiting>2`) may overshadow a deadlock situation (where `threadsWaiting=2`.) Consider Figure 13, a modified version of Figure 12. Here, f(c) = 7 + 2 + 7 = 16. The situation of Figure 13 has higher fitness than Figure 12, yet the latter is an instance of a deadlock while the former is not. In Figure 12, a cycle is present: T1, M2, T2, M1, T1. In Figure 13, no such cycle is present. To ensure that a deadlock is detected when there is one, a RAG is built after scheduling, i.e., once we know the allocation of resources to threads according to chromosome data. A RAG is built during deadlock detection only for situations where `threadsWaiting`≥ 2. Hence, at the end of the time interval, after allocation of resources to threads according to chromosome data, if the sum of the number of threads in all wait queues is greater than or equal to two, a RAG is built and evaluated for cycles. Cycles in a RAG indi-

cate deadlocks. Identifying cycles in such a graph is a well-known problem of linear time complexity[1]. Once a deadlock is detected from the RAG, the GA terminates and the chromosome yielding the deadlock is returned.

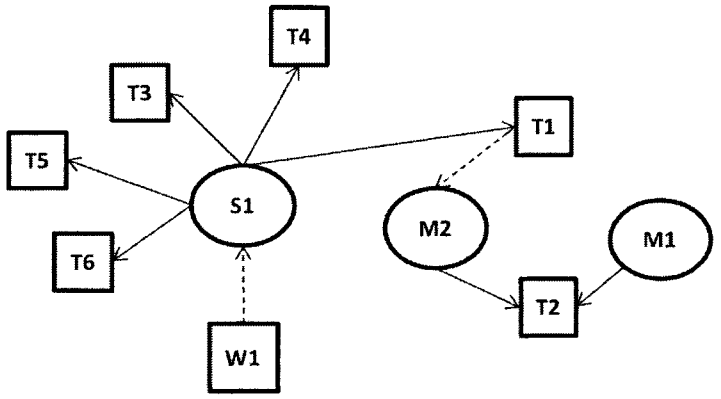

Figure 13: Deadlock fitness function example

The fitness function in (1) possesses a number of qualities: 1.) Because deadlocks involve at least two waiting threads, the fitness of scenarios where at least two threads are waiting on locks is always greater than the fitness of scenarios where zero or one thread is waiting; 2.) The fitness function is driven by the number of locks locked, i.e., an additional thread executing in a lock increases the fitness; 3.) The fitness function is driven by the number of threads waiting on locks, i.e., an additional thread waiting for access to a lock increases the fitness. Property 1 ensures that situations where no deadlock is possible are penalized, whereas properties 2 and 3 guide the search towards situations where deadlocks are possible and increasingly likely. Details of how this fitness function satisfies the aforementioned properties are provided in Appendix B.

[1] Tarjan's algorithm has linear running time based on the sum of the number of edges and nodes in the graph.

The timing interval here is based on the longest thread execution time in all locks (`lt`) and the maximum lock access time range of all threads (`ll`). Our heuristic is to guarantee, using these two variables, that all thread accesses to locks will occur at least twice, giving deadlocks a chance to occur. Therefore, the time interval equals: `[0-(lt+ll)*2]`.

5.2 Starvation Detection

Recall that in the context of concurrency, starvation occurs when threads cannot gain access to a lock because their executions are delayed by other threads. By definition, a thread waiting for access to a lock cannot proceed in execution until it acquires that lock. Hence, a thread waiting on a lock cannot be simultaneously in the wait queue of another lock. This implies that if starvation occurs for a particular thread, it will do so in the context of a single lock. We allow designers to track starvation for a target thread and target lock, and do this in turn for every thread and lock that is deemed critical. Therefore, the values to be optimized during starvation detection are the access times of threads to locks, such that the target thread waiting to access the target lock starves. We next introduce the various constituting components of our GA, with this objective in mind.

5.2.1 Chromosome Representation

The same 3-tuple chromosome representation used for deadlocks is also used for starvation. Hence, a starvation chromosome is depicted by genes as (T, L, a), where T is a thread, L is a lock, and a is T's access time of L. A tuple represents the execution of a thread when trying to access a lock.

5.2.2 Crossover Operator

The same one point crossover used in deadlock detection is also used for starvation detection. Hence, two parents are randomly split at the same location into two parts which are then alternated to produce two children.

5.2.3 Mutation Operator

The same mutation operator used in deadlock detection is also used for starvation, whereby each gene in the chromosome is mutated based on a mutation probability and the resulting chromosome is evaluated for its new fitness.

It is interesting to note that the chromosome representation and the mutation and cross-over operators are common to starvation and deadlock detection. What differs among them is the fitness function. This illustrates that the whole approach is easy to adapt from one type of concurrency problem to another, with adaptations performed on the objective function.

5.2.4 Objective Function

Recall that starvation detection requires the designer to select both a target thread and target lock. The chances of lock starvation increase when the number of threads accessing a particular target lock at the same time as the target thread increases. In other words, the more threads that try to access the target lock at the same time as the target thread, the greater the chance of starvation. We use this premise to develop an appropriate fitness function. Because threads wait for access to a resource in a wait queue,

we also need to examine the wait queue of the target lock over the time interval: e.g., if the wait queue of the target lock becomes empty, then the target thread accesses the lock and there is no starvation.

We next define a number of properties that we deem the fitness function should possess: 1.) Because starvation involves at least the target thread waiting in the target lock's wait queue, the fitness of scenarios where the target thread is waiting for the target lock should always be greater than the situations where: a.) no thread is waiting for access to the target lock or b.) threads are waiting for access to the target lock, but the target thread is not one of them; 2.) If the target thread gains access to the target lock once, later accesses in subsequent time units should still be checked for starvation. Hence, each access of the target thread should be treated as a separate instance of possible starvation. Property 1 ensures that situations where no starvation occurs are penalized, whereas property 2 ensures that multiple target thread accesses will be treated separately.

Based on the premise and properties aforementioned, we examine the fitness function `f(c)` below of a chromosome *c*, given a target lock and a target thread in the system under test. Details of how this fitness function satisfies the aforementioned properties are provided in Appendix C. The fitness function is weighted such that the longer the target thread spends waiting on the target lock, the greater its fitness.

$$f(c) = \sum_{t=startTime}^{endTime} \begin{cases} 1 & \text{if i} = 0 \text{ and } A \notin execThreads_t \text{ and } A \in waitingThreads_t \\ i & \text{if } A \notin execThreads_t \text{ and } A \in waitingThreads_t \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The variable *A* represents the target thread. Variables `startTime` and `endTime` denote the time interval start and end times, respectively. The set of threads executing within the target lock at time unit `i` is denoted $\texttt{execThreads}_i$ and $\texttt{waitingThreads}_i$ is the set of threads waiting for access to the target lock at time `i`. The sets $\texttt{execThreads}_i$ and $\texttt{waitingThreads}_i$ are obtained after scheduling threads and are calculated for every time unit of the time interval. This means that before a fitness value is associated with a chromosome, the execution of threads, as specified by their access times to locks in the chromosome must be scheduled by a scheduler.

Similarly to deadlock detection, some of the input information required for the detection of starvation as retrieved from the MARTE profile is used by the scheduler. The scheduler uses thread priorities in conjunction with the wait queue access policy to determine the order of thread accesses to locks. The scheduler also uses thread execution times within locks to determine how long each thread executes within an associated lock, as well as lock capacity to determine how many threads can access a lock concurrently. The scheduler also uses the chromosome to determine when a thread actually requires access of a lock.

Our fitness function is defined to give higher fitness values to situations where the target thread is waiting longer on the target lock, as well as later in the time interval; larger values are therefore indicative of fitter individuals. The fitness function assigns a fitness value to a chromosome based on whether the target thread is waiting on the target lock during each time unit of the time interval. If the target thread is waiting, the time unit is added (or 1 in the case of time unit 0). If, during `i`, the target thread executes within the target lock, or is not waiting on the target lock, the fitness value is not incremented.

The fitness is weighted in the sense that the longer the target waits along the time interval, the greater its chance of being starved.

It is important to note that the fitness function focuses on one target thread for a target lock at a time. This is because the search can only focus on one thread and one lock at a time when searching for starvation scenarios. The user can, however, investigate various threads and locks in turn, following an order that can be determined by their order of criticality, for example.

Consider an example with the following chromosome: (T1, L1, 0) (T2, L1, 0) over the time interval [0-3]. T2 and L1 are the target thread and lock, respectively. L1 has capacity 1. Further assume that the execution time of T1 in L1 is 2 and the execution time of T2 in L1 is 4 time units. T1 has a high priority and T2 has a low priority. At time unit 0, both threads attempt to access the target lock. Because T1 has higher priority, it gains access to the lock, leaving T2 waiting in the wait queue. At time unit 1, T1 continues to execute in L1, with T2 still waiting in the queue. At time unit 2, T1 releases L1 and T2 is granted access to the lock, leaving the wait queue empty. Thus, at time unit 2, the target thread accesses the target lock. The fitness values for each time unit are as follows:

Time unit 0: $\text{fitnessValue}_0 = 1$

Time unit 1: $\text{fitnessValue}_1 = 1 + 1(\text{fitnessValue}_0) = 2$

Time unit 2: $\text{fitnessValue}_2 = 0 + 2\ (\text{fitnessValue}_1) = 2$

Time unit 3: $\text{fitnessValue}_3 = 0 + 2\ (\text{fitnessValue}_2) = 2$

$f((T1, L1, 0)\ (T2, L1, 0)) = \text{fitnessValue}_3 = 2$

If, for the same example, both threads tried to access the lock at time unit 3 instead of time unit 0, the fitness values would be as follows:

Time unit 0: $fitnessValue_0 = 0$

Time unit 1: $fitnessValue_1 = 0 + 0(fitnessValue_0) = 0$

Time unit 2: $fitnessValue_2 = 0 + 0(fitnessValue_1) = 0$

Time unit 3: $fitnessValue_3 = 3 + 0\ (fitnessValue_2) = 3$

$f((T1, L1, 3)\ (T2, L1, 3)) = fitnessValue_3 = 3$

It is important to note that according to the defined fitness function, the only situations that would result in starvation situations are ones where the thread is waiting on the target lock at the end of the time interval (i.e., $A \in waitingThreads_{endTime}$). This is the termination criterion used to determine the presence of starvation and it may result in false positives. False positives are a concern only in starvation detection due to the termination criterion used. They arise when the target thread remains waiting on the target lock at the end of the time interval, but if the time interval were increased, the target thread would eventually execute within the target lock. This is the case with the second chromosome (f((T1, L1, 3) (T2, L1, 3))) from the example above. If the time interval were increased to [0-5], T2 would gain access to the lock. Even when false positives occur, they can still unveil problems in the system under study. For some systems, performance constraints may necessitate a maximum wait time for threads on resources. If this maximum wait time elapses, performance constraints are violated. When positive results are reported, it is up to the user to determine whether to increase the time interval and re-run the starvation detection, or whether performance constraints are violated due to the maximum wait time. In the latter case, the user would effectively treat the positive result as an instance of starva-

tion. In the former case, if the positive result is still reported, then this is an indication that there is starvation. If - by increasing time - the positive result is no longer reported, then the user can consider the initial result as a false positive.

For the fitness function to be effective, the testing range over which it is defined (i.e., `startTime` and `endTime`) must be adequate: it should be long enough for starvation to occur, but not too long such that it unnecessarily increases testing time. Our heuristic here depends on the amount of time that can be spent on each GA run. This is determined by time availability and practical considerations. Designers can choose an initial time interval (e.g., [0-100]), use it to run a few generations, then decide based on these runs an appropriate time interval that fits their time budget. Our experiments suggest there is a natural logarithmic relationship between the time interval and the amount of time spent in execution of the GA. Designers can simplify calculations by erring on the side of caution and assuming that the relationship is linear.

5.3 Data Race Detection

Recall that data races are types of faults that arise from unsynchronized thread access to the same memory location. In particular, data races occur when at least two threads share a resource and at least one is a writer thread. Problems then arise due to the order of execution of events. The values to be optimized for data race detection are the access times of threads to locks, such that an overlap of at least one writer thread with any other thread occurs within the shared resource. We next introduce the various constituting components of our GA, with this objective in mind.

5.3.1 Chromosome Representation

Much like deadlocks and starvation, the values to be optimized during data race detection are the access times of threads to a resource, such that the number of threads accessing a resource simultaneously is maximized. These access times are the values that will be altered by the GA to try to reach a data race situation. As is the case with deadlocks and starvation, the access times here must also reflect schedulable scenarios, which entails meeting system specifications of periods, minimum arrival times, and so on.

Recall from Table 2 that for data races, we need to encode threads (<<SwConcurrentResource>>), resources (<<SharedDataComResource>>), read and write operations (<<RtService>>/ concPolicy = read, <<RtService>>/concPolicy = write) and access and execution times (<<gaStep>> / interOccTime or <<gaStep>> / execTime), which are available in the input model. Any time constraint (<<TimedConstraint>>) is also accounted for in the chromosome.

Since, by definition, a data race involves multiple accesses to the same shared memory location, we consider only one resource at a time. Hence, the gene does not need to contain encoding of the resource, and can be depicted as a 2-tuple (T, a), where *T* is a thread and *a* is *T*'s access time of the resource[2]. As in the previous tuple representations for deadlocks and starvation, data race tuples are defined for a user specified time interval during which the designer wants to study the system's behavior. A special value of -1 is used to depict access times that lie outside this interval. It is important to note that information about the type of access (i.e. reader or writer) is not encoded in the gene. Rather, it is

[2] For reusability, the chromosome is still a 3-tuple, (T, L, a), but L is fixed.

considered as a property of the thread access, along with the valid access time ranges. This is useful for threads which are both readers and writers (i.e. sometimes access the resource as a reader, sometimes as a writer).

The overall chromosome size is determined by the total number of times all threads attempt to access the resource in the given time interval. A thread can appear more than once in the chromosome if it accesses the resource multiple times. The same timing constraints on valid chromosomes from Section 5.1.1 must also be met.

5.3.2 Crossover Operator

The same single point crossover is used for data races as is used in deadlock and starvation detection. Crossover here, however, proceeds on the two-tuple representation of the genes of the chromosome, rather than the previous three-tuple representation.

For reader/writer threads (ones that access the shared resource as both a reader and writer), the order of reads and writes can change during crossover. When this happens, it will still be within the allowed timing constraints. For example, assume thread T1 accesses a shared resource as a reader between the interval [1-5], then again as a reader between [10-15], then as a writer between [14-20]. A chromosome indicating the thread accessing the shared resource as a reader, reader, then writer is: (T1, 2) (T1, 15) (T1, 20). After crossover, this order may change, as indicated by the following chromosome: (T1, 2) (T1, 15) (T1, 14). Because of constraint 1 (Section 5.1.1), the genes would be re-ordered to (T1, 2) (T1, 14) (T1, 15), with the thread now acting as a reader, writer, then reader, respectively.

5.3.3 Mutation Operator

Mutation too proceeds as before, operating on the altered two-tuple chromosome. For reader/writer threads, the order of reads and writes can change during mutation in a similar fashion as for crossover. For the previous example, with the original chromosome (T1, 2) (T1, 15) (T1, 20), mutating the last gene so that its access time is 14 instead of 20 will result in the change of the original read/writer order.

5.3.4 Objective Function

We define the following fitness function for data race detection:

$$f(c) = \min_{i=startTime\ to\ endTime} \begin{cases} |W_i - N(W_i)| & if \#W_i \geq 1, \#T \geq 2 \\ endTime & otherwise \end{cases} \quad (3)$$

StartTime and *endTime* are the starting and ending times of the time interval. W_i is the time unit i during which a writer thread accesses the shared resource. $N(W_i)$ is the time unit i of the nearest executing thread to W_i within the resource. W_i and $N(W_i)$ are in the range [*startTime-endTime*]. $\#W_i$ is the total number of writer threads during time unit i and #T is the total number of threads that access the resource during the time unit i. $N(W_i)$, W_i, #T and $\#W_i$ are obtained after scheduling.

The fitness function uses input information regarding type of thread (reader or writer) from the MARTE profile. The scheduler uses execution times of threads within shared resources along with the chromosome to determine when a thread actually executes within a shared resource.

The fitness function of equation (3) is a minimizing function; hence, it gives lower values to fitter individuals. Essentially, the fitness function minimizes the difference of resource access times between writer threads and any other thread (reader or writer). The smaller the difference, the closer the overlapping execution of a writer thread with another thread. A fitness value of zero indicates the presence of a data race, whereby the writer thread is executing within the resource at the same time unit as another thread, hence a data race. This is one of the properties of the function that guides the search towards situations where data races are possible and increasingly likely. The fitness function also ensures that scenarios where data races are possible (two threads executing and at least one is a writer) are always rewarded over situations where no data races are possible (when zero or one thread is executing, regardless of its type). Hence, it is never the case that *c1* is a chromosome that results in a data race and *c2* is a chromosome that does not yield a data race, but f(*c1*)>f(*c2*). Details of how the fitness function adheres to the aforementioned properties is given in Appendix D.

Our heuristic for the identification of the time interval is the same as the one presented for deadlocks: all threads can completely access the resource at least twice. The time interval is: [0-*(lt+lr)*2*], where (*lt*) is the longest thread execution time in the resource and (*lr*) is the maximum resource access time range of all threads.

Let us consider an example. Assume we are calculating the fitness of the following chromosome: ((T1, 324), (T2, 327), (T3, -1)) where T1 is a writer thread and all other threads are readers. The time interval is assumed to be [0-350]. When scheduled, the chromosome can be depicted as shown in Figure 14. It is assumed that all threads are accessing resource R1. For the majority of the time interval, the re-

source is not utilized. It is only used by T1 at time unit 324 (for only one time unit) and by T2 at time unit 327 (also for one time unit). Using equation (3) on Figure 14, we examine the time units for resource R1 and calculate the fitness:

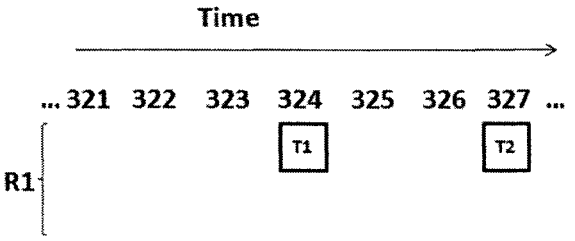

Figure 14: Scheduling of data race fitness example

At time units less than 321, as well as time units 321, 322, and 323: $\#W_i = 0$, min = 350

At time unit 324: $\#W_i = 1$, $W_i = 324$, $N(W_i) = 327$, absolute difference = 3, min = 3

At time unit 325, 326, and 327: $\#W_i = 0$, min = 350

then, f(c) = 3.

5.4 Summary of Approach

Recall that the information presented in Table 2 represents the inputs required for fault detection as extracted from the MARTE sequence diagrams. This information is used by the appropriate GA, depending on the type of fault being addressed. Some inputs are used to generate chromosomes, which are then evaluated by the various fitness functions which internally make use of the scheduler. Other inputs are used by the fitness functions to aid in their calculations. Yet other inputs are used by the scheduler.

Table 3 presents a mapping of which MARTE inputs are used by chromosomes, fitness functions and the scheduler.

Table 3: MARTE input mapping to type of GA component used

Concept	MARTE Stereotype/Tag	Fault	Component
Thread	<<SwConcurrentResource>>	Deadlock, Starvation, Data race	chromosome
Lock	<<SwMututalExclusionResource>>	Deadlock, Starvation	chromosome
Lock acquire	<<Acquire>>	Deadlock, Starvation	chromosome
Lock release	<<Release>>	Deadlock, Starvation	chromosome
Wait queue access policy	<<SwMutualExclusionResource>>/ waitingQueuePolicy	Deadlock, Starvation	scheduler
Thread priority	<<SwConcurrentResource>>/ priorityElements	Deadlock, Starvation	scheduler
Thread execution times with-in locks	<<gaStep>>/execTime	Deadlock, Starvation	scheduler
Lock capacity	<<SwMutualExclusionResource>>/ accessTokenElements	Deadlock, Starvation	scheduler
		Deadlock	fitness
Thread access range of locks	<<gaStep>>/interOccTime | <<gaStep>>/execTime	Deadlock, Starvation	chromosome
Unprotected resource	<<SharedDataComResource>>	Data race	chromosome
Reader thread	<<RtService>>/concPolicy = reader	Data race	fitness
Writer thread	<<RtService>>/concPolicy = writer	Data race	fitness
Thread exec. time in shared resource	<<gaStep>>/execTime	Data race	fitness
Thread access range of shared resource	<<gaStep>>/interOccTime| <<gaStep>>/execTime	Data race	chromosome
Time constraints	<<TimedConstraint>>	Deadlock, Starvation, Data race	Chromosome

6 TOOL SUPPORT

We have built a prototype tool, Concurrency Fault Detector (CFD), supporting our method. CFD is an automated system that identifies concurrency faults in concurrent applications modeled with the UML/MARTE notation. It can help identify deadlock, starvation and data race faults.

6.1 Tool Description

CFD involves a sequence of steps. Users first input information: (1) UML/MARTE sequence diagrams for the analyzed system, (2) the execution time interval during which the system is to be analyzed, and (3) the type of concurrency fault targeted: deadlock, starvation or data race. For starvation, the target thread and target lock are also inputted, while only the target shared resource is needed for data race detection. CFD then manually extracts the required information from the inputted UML/MARTE model (i.e., from its sequence diagrams). This step would not be difficult to automate as described in Appendix A.

CFD is decomposed into three portions: a scheduler, a genetic algorithm, and a RAG evaluator. Depending on the type of concurrency fault targeted, the appropriate objective function is used in the GA, as described in Sections 5.1.4, 5.2.4 and 5.3.4. When a system is designed, assumptions are made about the architecture it will be run on. These deployment assumptions are incorporated in CFD in the form of the scheduler. It is important to note that the approach we propose is not affected by the scheduling technique chosen and CFD can, in the future, provide a choice of several schedulers. In the context of our case studies, CFD's scheduler emulates single processor execution and is POSIX compliant.

Deadlock detection is performed using a RAG whenever a chromosome results in at least two threads waiting on locks. If a cycle is found, CFD outputs the details of the chromosome causing it as well as that chromosome's fitness value. The chromosome provides users with a particular scenario that leads to the deadlock found so it can be recreated. Users are also provided with a textual list of threads executing in all locks, as well as a listing of all threads waiting in all lock wait queues. This helps users determine the threads and locks involved in the deadlock. Furthermore, the RAG corresponding to the deadlock is output. This too helps users identify the threads and locks involved in the deadlock. A sample output for a detected deadlock is:

```
Deadlock Detected!
RAG Cycle (Threads and Locks involved in deadlock): [L1, L2, T4, L3, T5, L4, T1,
L5, T3, T2]
chromosome: (T1, L1, 31)(T1, L5, 32)(T2, L1, -1)(T2, L1, 3)(T2, L1, 38)(T2, L2, -
1)(T2, L2, 2)(T2, L2, 37)(T3, L2, 5)(T3, L3, 4)(T4, L3, 10)(T4, L4, 9)(T5, L4,
17)(T5, L5, 16)
fitness value: 15.0
```

From the example, the deadlock involves the threads and locks iterated in the RAG cycle. Hence, the threads and locks involved in the deadlock are T1, T2, T3, T4, T5 and L1, L2, L3, L4, L5, respectively. The deadlock occurs (as outputted by the chromosome) when T1 accesses L1 at time unit 31; T1 accesses L5 at time unit 32; T2 accesses L1 at time unit 3; etc. These accesses are translated to the fitness of the chromosome, namely 15.

Currently, the output information provided is in textual format. However, displaying the information back to users in the form of a UML diagram can be done in a reverse process of what is presented in Appendix A.

If no deadlock is found, CFD terminates, showing both the fitness value and output details of the highest fitness chromosome found (in text form). Even when no deadlock is detected, the highest fitness chromosome output can still be of use as it is likely very close to an actual deadlock scenario. Hence, it may uncover potential deadlocks that can arise due to actual execution times of threads in locks slightly differing from the estimates used as inputs in the model.

Starvation is detected when the target thread is waiting on the target lock at the end of the time interval. CFD uses this as a termination criterion for starvation detection. When starvation is detected, CFD terminates showing the highest fitness chromosome found, as well as its fitness value, and executing and waiting threads of the target lock. It also shows the times when the target thread executed within the target lock. Times when the target thread requested access of the target lock are given in the chromosome. When the target thread remains waiting at the end of the time interval, we may be in presence of a starvation case or a false positive. Recall that false positives arise when the target thread remains waiting on the target lock at the end of the time interval, but if the time interval were increased, the target thread would eventually execute within the target lock. The distinction between starvation and false positive is difficult to decide. One practical approach is for the user to use the output of CFD to determine how long the thread has been waiting and whether this is beyond a practical maximum above which this can be considered a starvation case for all practical purposes. If it is not, users can increase the time interval and re-run CFD. For example, assume that T1 and L1 are the target threads and locks respectively, and a run of CFD produces the following for a time interval of [0-400]:

```
Starvation Detected!
Target lock: L1
Target thread: T1
```

```
chromosome: (T1, L1, 22)(T1, L1, 129)(T1, L1, 236)(T1, L1, 343)(T1, L40, 23)(T1,
L40, 130)(T1, L40, 237)(T1, L40, 344)
Times T1 accessed L1: 22
Fitness value: 271
```

T1 accesses L1 at time unit 22, but it is unable to do so from time unit 129—the next time it attempts to access L1 (second gene)—until the end of the time interval (400). The designer determines that the wait time for T1 has not exceeded the maximum wait time of 750 time units, so the time interval is increased to 900 and CFD is run again, yielding the same results. The designer will then consider this an instance of starvation and will alter the design.

When no starvation is detected, CFD terminates, outputting details of the executing and waiting threads of the target lock for the fittest chromosome found. Much like deadlocks, these can turn out to be useful if execution time estimates turn out to not be accurate. For example, assume three threads access a target lock with execution times within the target lock being two time units for each thread. Let us further assume that with thread three being the target thread during a time interval of [0-6], no starvation is detected, with the outputted chromosome being: (T1, L1, 1) (T2, L1, 3) (T3, L1,5). Once the system is built, let us assume that the lock execution time estimates were off, with each thread executing for three time units (instead of two) within the target lock. Since the previously outputted chromosome represented the nearest scenario found to starvation, it can be used at this stage to test the implemented system. With the new execution times, it turns out that starvation does indeed occur during the specified time interval.

In the GA for data races, if a data race is detected, CFD outputs the chromosome resulting in the data race, its fitness value, as well as the time unit at which the data race occurs and a textual depiction of the threads executing within the resource at that time. A sample output would be:

```
Data race Detected!
chromosome: (T1, -1)(T1, 325)(T1, 723)(T2, -1)(T2, 325)(T2, 726)(T3, -1)(T3,
330)(T3, 730)
fitness: 0.0
Executing Threads at time 325 that cause a data race: T1  T2
```

If no such sequence is found, CFD terminates outputting the chromosome with the lowest fitness value (since it is a minimization function) as well as the fitness value.

Since collecting input data can be automated from a UML case tool, and all the other phases are automated, CFD is meant to be used interactively: the user is expected to fix the design of the system when CFD terminates with a detected deadlock situation, starvation or data race. This is the main reason why we developed a strategy that only reports one fault scenario at a time, i.e., per run of CFD, allowing designers to fix the system's design before running the modified design again on CFD. Such a stepwise refinement process, however, requires the problem detection to be efficient enough and to scale up, even on large UML/MARTE models, which we show to be the case in Section 7.6.

CFD is used to investigate whether scenarios can be generated where deadlock, starvation or data race faults occur. If no such scenario is found, this does not guarantee that none exist, as GAs are based on heuristics. However, one can still feel more confident that such a case is unlikely (i.e., rare in the search space). One can feel even more confident by running CFD several times. The number of times differs for each system, depending on the available time budget as well as the acceptable probability of not detecting a fault. If designers are bound by time budgets, they may choose to run CFD only a few times if each run takes long to execute. However, as a tradeoff, they must also consider the resulting probability of not detecting a fault in those runs. For n runs, the probability of not detecting a fault - assuming each run is independent and no bias is introduced - is: (1-detection rate)^n. For example, as-

suming a detection rate of 10%, running CFD twice will result in an 81% chance of faults going undetected. However, ideally this probability should be made very small by running CFD a sufficient number of times, within practical time budgets. In practice, one would make an estimate (possibly pessimistic to be conservative) for a run to detect faults, and then compute the number of runs required to achieve a selected, very large probability of detection over all runs.

6.2 GA Parameters

Though various parameters of the GA must be specified, we can fortunately rely on a substantial literature reporting empirical results and making recommendations. Parameters include the type of GA used, termination criterion, population size, mutation and crossover rates and selection operator. All parameter values are based on findings reported in the literature, as detailed below. In addition, we have fine tuned population size and the termination criterion based on some experimentations.

The type of GA we use is a steady state GA, with a worst replacement scheme and 50% replacement, as suggested in [Haup 98]. The population size we apply is 200. This is higher than the size suggested in [Haup 98], but through our experimentation, we found this works more effectively for larger search spaces. The selection operator is rank selector, whereby chromosomes with higher fitness are more likely to be chosen than ones with lower fitness [Koza 92]. Mutation and crossover rates are $1.75/\gamma\sqrt{l}$ (where γ denotes the population size and l is the length of the chromosome) and 0.8, respectively. Both are based on the findings in [Back 92] and [Haup 98], respectively. The termination criterion we apply is number of generations. In particular, the GA terminates after 1000 generations if no deadlock, starvation or data race is found. According to [Safe 04], the value of the termination criterion requires

some knowledge of the application to determine an appropriate search length. Hence, there is no set value or guideline for this criterion. Through experimentation on our case studies, we found 1000 generations to be adequate for our various search spaces. In the future, further studies of convergence are needed to verify this value.

These parameter values have worked exceedingly well in all our case studies when considering both the detection rate and execution time to find a concurrency fault. The same parameter values can be used for other system designs, though further empirical investigation is required to ensure the generality of these parameter values in our application context. In the worst case, if one wants to be on the safe side and ensure fully optimal results, the parameters can be fine tuned once for each new system design: when the system design being checked is first analyzed. For further design modifications of the same system, the parameters need not be fine tuned.

6.3 Tool Limitations

CFD is a prototype tool. It was developed as a proof of concept and to facilitate the running of test cases. As such, it has a number of limitations. These limitations, however do not affect the applicability of our approach. CFD does not currently perform any direct querying of the UML/MARTE metamodel to gather real-time information. Instead, this information has to be manually extracted and provided to CFD as an input text file. However, as seen from Appendix A, automating this can be easily accomplished. Furthermore, our tool is not integrated with any UML/MARTE case tool.

Implementation wise, the code running CFD could be optimized. For example, when checking for $N(W_i)$ (the nearest thread to the current writer thread in data race detection – Section 5.3.4), a loop

checks for the nearest thread during time units less than *i*, the current time unit. It also checks for time units greater than *i*, then compares the two to determine the nearest thread. This can be optimized by keeping track of the nearest previous and following threads instead of having to determine them every time through the loop. As a result, test case execution times can be lowered by increasing execution efficiency. Furthermore, CFD does not take advantage of parallelism in GAs, again affecting test case execution times.

Our tool does not currently support various UML sequence diagram constructs, such as alternatives (e.g., alt combined fragment) and loops. However, supporting these would not affect the form of the various GA components, from chromosomes, operators and fitness functions. For example, alternatives specify various paths that an execution can take. If a thread access to a lock appears in an alt combined fragment, the access may not be triggered if the alternative path is taken. This would not, however, affect the size of the chromosome by introducing a new gene. This is because the chromosome size is determined based on all thread accesses to locks. Hence, for the alternative path example, the size of the chromosome would include the execution in the alt fragment. If that path is taken, the gene would have a non -1 access time. If the path is not taken, the gene would have its access time set to -1. The case would be similar for lock accesses that occur in a loop combined fragment, whereby the number of times the thread accesses the lock is determined by the number of times the loop is executed. The size of the chromosome would account for the maximum number of times the loop can be executed, assigning genes for each loop execution. The number of times the loop is executed is reflected by the number of non -1 genes assigned for the loop execution in the chromosome. At the chromosome level, this all still translates to access times of threads to locks, or shared memory locations. If additional

constraints are needed on the chromosome due to the UML constructs (such as loop fragments), these will need to be enforced by the tool. For example, assume three threads T1, T2 and T3 access a shared memory location. T1 and T2 are writer threads and T3 is a reader. T3's access of the shared memory location is specified in a loop which executes at most twice. If either T1 or T2 updates the shared memory location within 5 ms after T3 first reads from it, T3 will perform a second read. Otherwise, T3 executes the loop just once, accessing the resource just once. This type of constraint (which we refer to as a UML control flow constraint) will be enforced by the tool on the chromosome in much the same way as timing constraints (see Section 5.1.1) are imposed. Hence, whenever a chromosome is created, mutated or has crossover performed, it must be validated for both timing constraints and UML control flow constraints. In this case, alternatives, loop fragments, and optional fragments are all considered a form of constraint which will be enforced on the chromosome by the tool.

7 CASE STUDIES

In achieving our last aim, namely evaluating the effectiveness of our method at detecting concurrency faults based on UML/MARTE design models, we used our tool to run a number of case studies. In so doing, we also examine our tool's run-time efficiency, though we realize that a great deal of improvement can be obtained by using more powerful hardware and distributed GAs [Dori 93]. We also study the scalability of our approach with respect to fault detection rates.

The case studies cover a variety of different classes of problems. It can be argued that a careful design inspection might highlight the possibility of concurrency faults in these case studies. However, such inspections are unlikely to be effective in detecting concurrency problems in large industrial systems, hence the need for our approach. Our aim is not to replace, but to support and enhance such careful design inspections by reporting on particular scenarios that can be used by system designers to prioritize faults in the system.

We first describe the case studies used for deadlock detection, followed by those used for starvation detection, then data race detection. Next we describe our experimental design before discussing results and scalability.

7.1 Deadlock Case Studies

We present three deadlock case studies: the dining philosophers, bank transfer and cruise control problems. The dining philosophers problem is the epitome of deadlock identification and is often used in the literature as a case study. The bank problem represents a class of problems with large, complex

search spaces. These are precisely the types of problems GAs are designed to handle well. To determine the overhead associated with using a GA on a small search space as well as on large, simple search spaces, we introduce the cruise control problem.

7.1.1 The Dining Philosophers Problem (Phil)

The renowned n-dining philosophers problem has commonly been used to demonstrate deadlock detection [Gode 04, Grad 06]. It is an interesting problem as it provides complex resource sharing as well as a large search space. The problem is summarized as follows: n philosophers (we use n=40) are sitting at a round table either eating or thinking. Every two philosophers share one fork, yet only one can access a shared fork at a time. The forks are set so that each philosopher has one on their right and one on their left. Philosophers must access both left and right forks when they are hungry in order to eat. Otherwise, when they are thinking, they do not need either fork. Design solutions to the dining philosophers problem differ in how the philosophers access the forks. The design solution we apply is this: when philosophers are hungry and attempt to eat, they pick up their left forks first followed by the right forks. When finished eating, forks are released in the same order. This solution design can be deadlocked if all philosophers attempt to eat at the same time and all pick up their left forks.

To detect a deadlock, we need to search the set of possible sequences of thread (philosopher) accesses to locks (forks) for at least one that yields a deadlock. This set of possible sequences is called the search space. In other words, the search space is the size of all possible combinations of access times in a chromosome for all philosophers. The size of the search space is design solution specific. What is independent of the design solution, and based just on the problem is the state space. The state space is

the combination of legal states that all philosophers can be in. Each philosopher can be in one of four states: thinking, holding right fork, holding left fork or eating. The total combination of states is therefore 4^{40}. However, for two adjacent philosophers, it is illegal that they both hold the same fork, making the state space size approximately[3] $1.2 * 10^{19}$. The search space, on the other hand, is affected by the number of philosophers and forks, the designated time interval, as well as the thinking and eating time ranges for each philosopher. For the particular 40 philosopher design solution we use, each philosopher has 100 choices (the thinking time – see later in this section) for accessing their left fork and only one choice for accessing their right fork (because pickups of forks occur consecutively). During the chosen time interval of 400 (see below), each philosopher can access their forks at least four times. Hence, per philosopher, the total number of combinations is: $(100*1)^4$. But there are 40 philosophers, so there are 100^{4*40} combinations which is approximately $1.0 *10^{320}$.

Search spaces are characterized by their complexity. All points in the search space represent valid solutions to the problem at hand. The best, or optimum solutions are the points in the search space that result in concurrency problems. These are called global optima, whereas local optima are ones where all surrounding points have worse fitness, but the point itself is not an instance of a concurrency fault. The more local optima in the search space, the more complex it is. To the best of our knowledge, no metric exists that can measure complexity. This is because complexity of a search space involves the number of solutions present within the search space, along with the overall shape of the search space and number of optimum solutions. One can, however, get a feel of the complexity of a search space through random search. A random search simply randomly selects individuals within the search space and

[3] The state space for large numbers of philosophers corresponds to $3^{\#phil}$, where #phil is the number of philosophers.

checks to see if they are global optima. If random search often succeeds in finding such individuals, this is a strong indicator that the proportion of optima to the size of the search space is large, hence suggesting that the search space is not complex. If, on the other hand, the proportion of solutions to the size of the search space is small, this means that the search space is complex. Hill climbing can also be used as a means of detecting the structure of the search space [Harm 09].

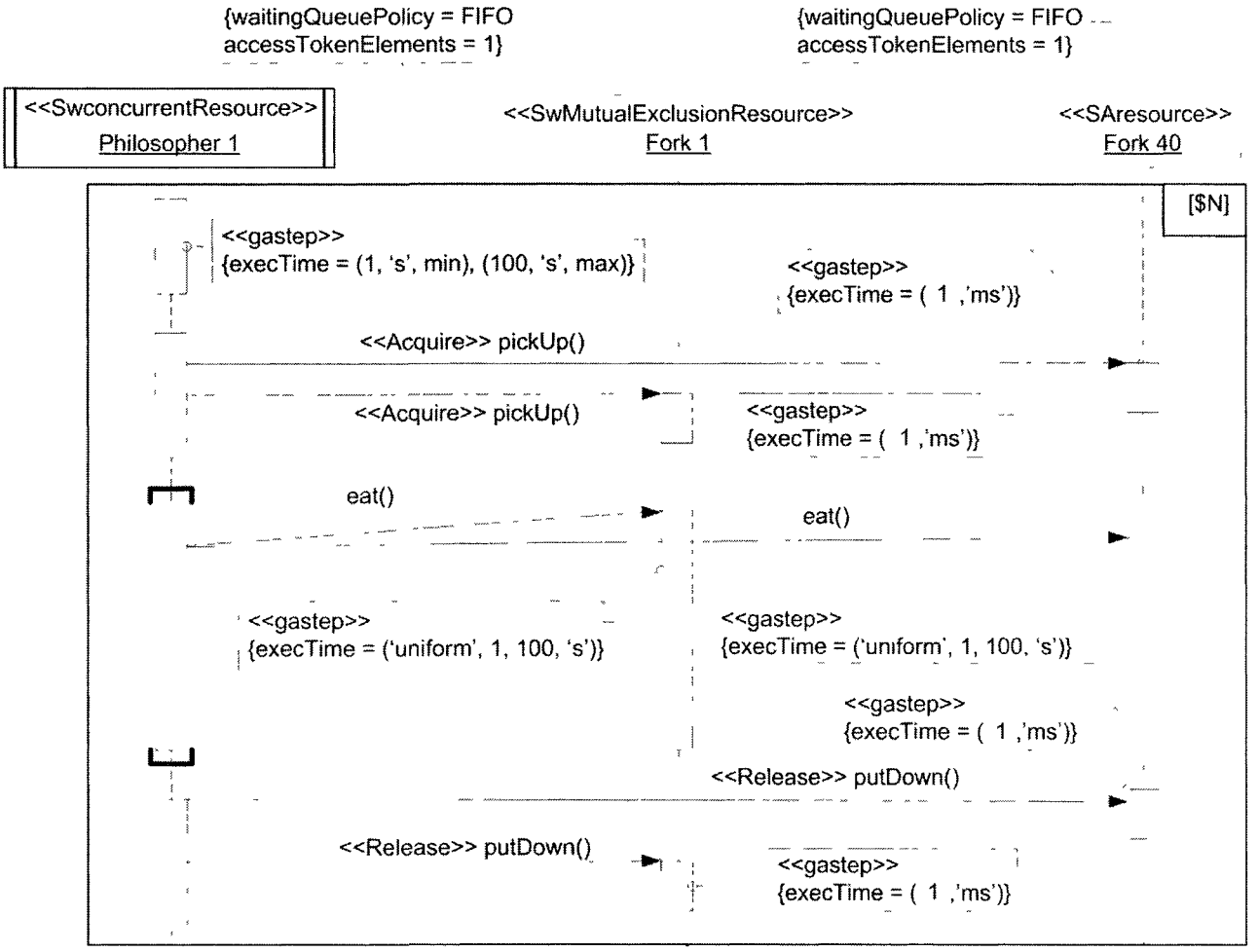

Figure 15: Dining philosophers sequence diagram

Concurrency aspects for the philosopher problem are depicted in Figure 15 for philosopher 1 and his/her two forks, as a UML/MARTE sequence diagram. This is an excerpt of the complete sequence diagram, which would show the interactions of all philosophers and all forks. Because so many interactions occur in the dining philosophers problem, in practice, the interactions would be portrayed in a set of sequence diagrams. One single sequence diagram or a set of sequence diagrams would not affect the applicability of our approach.

In Figure 15, Philosopher 1 is depicted as a concurrently executing thread via `<<SwConcurrentResource>>`. It acquires two locks, Fork 40 and Fork 1, designated by `<<SwMutualExclusionResource>>`. Fork 40 (left fork) and Fork 1 (right fork) each allow only one thread to execute at any given point in time, as indicated by `accessTokenElements`. Each lock's waiting threads are accessing the lock on a first come first served basis as specified by `waitingQueuePolicy`. The philosopher's thinking time (thread access range of locks) is represented by `execTime` in the first `<<gastep>>`. Access time has a minimum and maximum time range. Execution times are discrete uniform distributions between 1 and 100 seconds. The execution duration of Philosopher 1 (thread) in each of the locks is defined by `execTime` on each lock's `<<gastep>>`, i.e., between 1 and 100 time units.

For deadlock detection, recall that CFD requires two inputs. The first input is the complete set of sequence diagrams. The second input is the time interval. We use our heuristic from Section 5.1.4: the longest thread execution time is 100 (maximum eating time), and the longest lock access time (i.e., thinking time) is 100, hence the time interval we use is 400.

Recall from the discussion on search space that during the chosen time interval of 400, each philosopher can access their forks at least four times. For 40 philosophers, there are thus 40 * 4 genes for the 40 philosophers accessing one of their forks, plus 40 * 4 genes for the philosophers accessing their other forks making the length of the chromosome equal to 320.

7.1.2 The Bank Transfer Problem (Bank)

The bank fund transfer problem is based on a simple banking functionality: fund transfer between accounts. It simulates multiple threads transferring funds among multiple, different accounts. The problem is based on the bank problem presented in [Yang 06]. Ten threads, representing 10 account holders, repeatedly transfer funds between any two randomly selected accounts out of 50 available accounts. It is an interesting problem as it models repeated resource sharing. Like the dining philosophers, a number of design solutions can be applied to this problem. The one we use involves a number of steps: When transferring from account A to account B, account A is first locked, then checks on the balance of A is performed. If A can transfer the amount, account B is locked and the user is prompted to verify the transfer of the amount, before the transfer is completed. Account B is released first, followed by account A. Transfers where the source and destination accounts are the same (i.e., A=B) are not allowed. In this design solution, a deadlock can occur if one thread is transferring from account A to account B, while another is simultaneously transferring from B to A. The state space of this problem depends on the number of threads and the number of accounts, specifically $(n^2-n)^t$, where n is the num-

ber of accounts and t is the number of threads[4]. For 10 threads and 50 accounts, the state space is approximately 7.7 * 10^{33}. For the particular design solution we use, each thread has 50 choices for source and 49 for destination, where the same source and destination are not included. This repeats for the specified time interval of 404 (see later in this section). Because each transfer can take 200ms (see later in this section), each thread can choose source and destination 3 times during the time interval. Thus, there are 50*49*3 choices per thread for source and destination. Because the choices for source and destination occur 3 times per thread, there are 200*3 combinations for how long the transfer takes. Combining all yields: 50*49*3*200*3 per thread. For 10 threads, the search space is approximately 2.7 * 10^{66}. Again, the necessary information used by the GA, including data to determine the time interval, can be retrieved from the UML/MARTE model: see Figure 16.

In Figure 16, Thread 1 is a concurrently executing thread: stereotype `<<SwConcurrentResource>>`. Each account is associated with a lock and each thread randomly selects the two accounts (Account `$i` and Account `$j`, stereotyped `<<SwMutualExlcusionResource>>`) that will be involved in the transaction. The selection of accounts (the first action stereotyped `<<gastep>>`) takes 1 ms. The thread then acquires the lock associated with the transfer source (Account `$i`). This begins within a timing range of [1-200] ms. Once that is done, an `initiateTransfer()` is applied, executing for 1 ms, after which the transfer destination lock is acquired (Account `$j`). The actual fund transfer takes 1 ms before the transfer destination lock is released, followed by the source lock. Hence, overall, the

[4] The state space for one thread is the cross product of the accounts minus the number of accounts (to remove the cases where the source and destination are the same account). The total state space is then the cross product of the thread state spaces

source account is locked for a total of 2ms. The destination account is locked for only 1 ms, and this locking occurs after 1 ms of execution within the transfer source's lock.

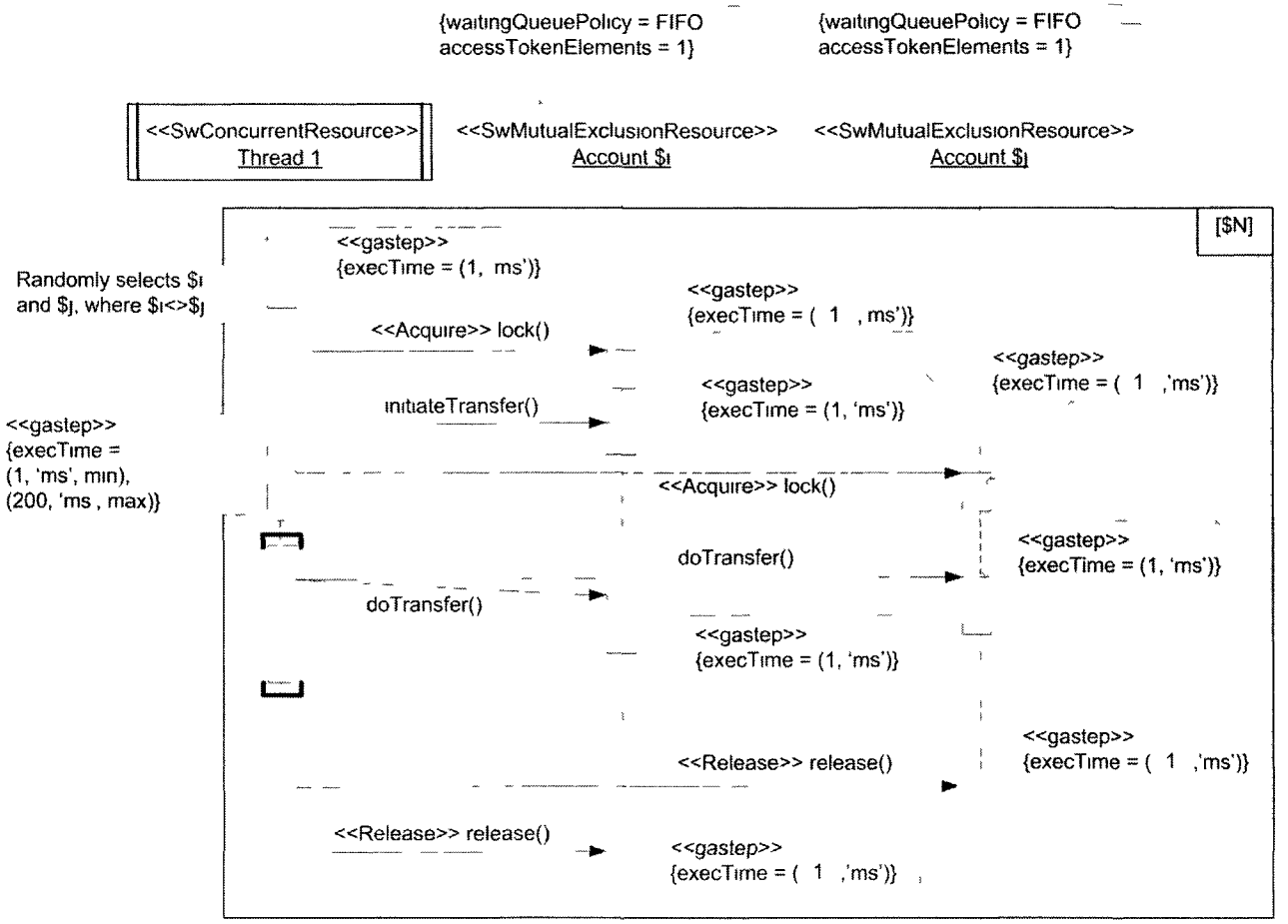

Figure 16: Bank transfer sequence diagram

For deadlock detection, the time interval used is based on our heuristic: the longest thread execution time is 2 ms (maximum transfer time), and the longest lock access time is 200 ms, hence the time interval is (200 + 2) * 2 = 404.

Recall from the discussion on search space that during the chosen time interval of 404, each thread can access accounts 3 times. For 10 threads, there are thus 10 * 3 genes for the threads accessing the source account, plus 10 * 3 genes for the threads accessing the destination account making the length of the chromosome equal to 60.

7.1.3 The Cruise Control Problem (Cruise)

The cruise control problem emulates a car simulator along with its cruising controller. The system is divided into a number of classes: `CarSimulator` simulates the car engine, runs a thread while the car is started, and simulates car speed changes based on the throttle and brake settings as well as the controlled speed by the cruising system when it is enabled; `CruiseControl` is a container for both car simulator and cruise controller of the car. This is the entry class that receives commands and dispatches them to the car and the controller; `SpeedControl` is a thread that runs in the `Controller` to adjust car speed whenever cruising is enabled.

When cruising is enabled, the current car speed is recorded and maintained for the duration of cruising time. When resuming cruising, the latest recorded speed is used as the speed to maintain during cruising; `Controller` simulates the cruise control of the car, disabling, enabling or resuming cruising according to the commands received by `CruiseControl`. It creates a new `SpeedControl` thread when cruising is enabled. A simplified version of the class diagram is presented in Figure 17

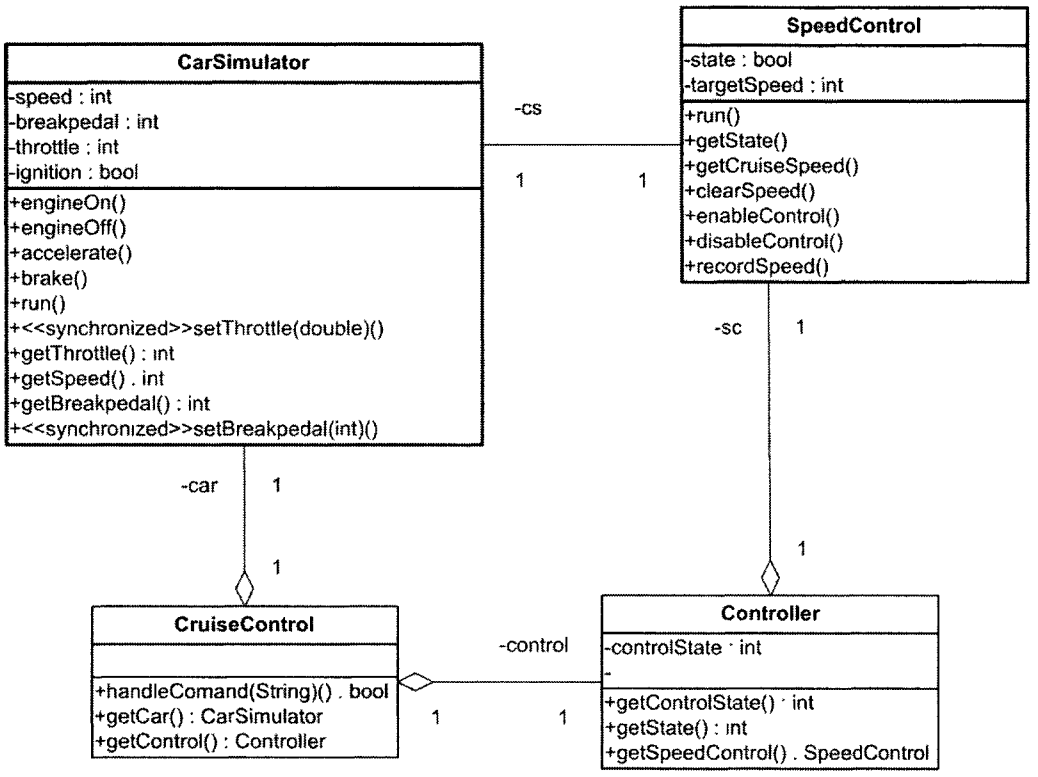

Figure 17: Cruise control class diagram

Unlike the other case studies, many different scenarios of execution, involving acceleration, cruising and breaking, are available here for verification. We limit our study to the following scenario for a number of threads and locks, as shown in Figure 18:

```
engine on
repeat
    accelerate
    cruise control on
    brake
```

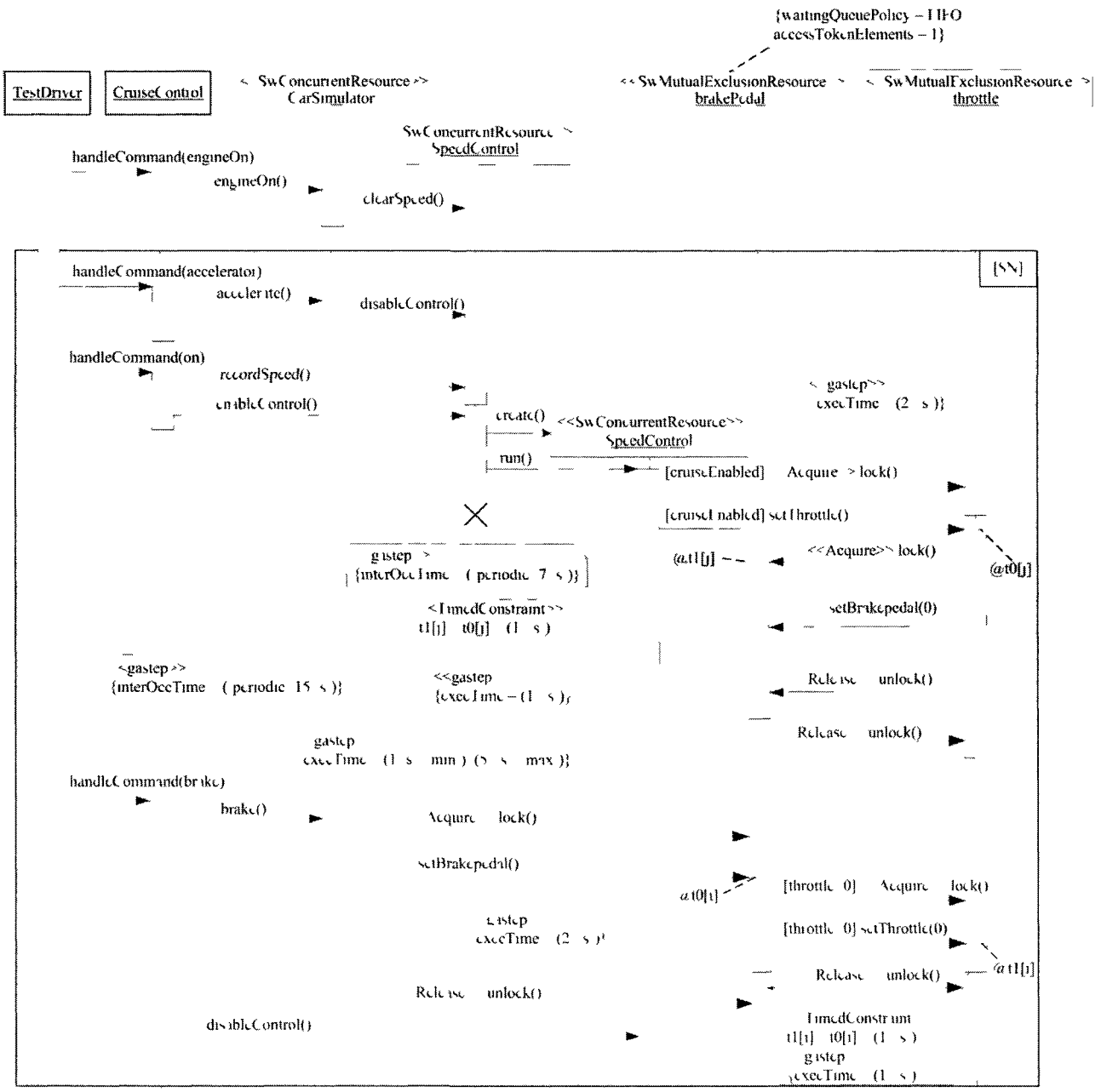

Figure 18 Cruise control sequence diagram

Acceleration takes between 1 and 7 seconds. Cruise control is turned on periodically every 7 seconds. Braking takes between 1 and 5 seconds, with the cycle repeating every 15 seconds. A deadlock can occur during breaking when the `CarSimulator` is executing the brake command and the `SpeedControl` is in its periodic run. When the `CarSimulator` executes a brake command, as seen in Figure 18, it first locks `brakepedal` then `throttle`. The opposite occurs when `SpeedControl` is in its periodic run: it first locks `throttle` then `brakepedal`. A deadlock shows up only when the timing is just right so that `CarSimulator` locks `brakepedal` and `SpeedControl` locks `throttle` at precisely the same time.

For deadlock detection, the time interval used is based on our heuristic: the longest thread execution time is 15 sec, and the longest lock access time is 2 sec, hence the time interval is (15 + 2) * 2 = 34.

`SpeedControl` locks `throttle` periodically every 7 seconds. Hence, in the time interval, at most 5 accesses can occur here. Similarly, `CarSimulator` locks `brakepedal` and repeats every 15 seconds. Hence in 34 seconds, at most 3 accesses can occur. Because each thread accesses two locks, the length of the chromosome is: (5 + 3) * 2 = 16.

It is important to note that, in practice when using a modeling tool, the sequence diagram of Figure 18 would be much simpler and feature a different layout. Information about various stereotypes would be embedded within the tool used to create the diagram, rather than appearing as comments. Information appears in comments here only to facilitate understanding.

7.2 Starvation Case Studies

For starvation, we introduce three more case studies: 1. the starve problem involves a large simple, search space. We created this problem to create an environment where verification of starvation would prove more challenging, especially when considering that the modified cruise version (see third case study) is really too simple; 2. A different version of the dining philosophers problem; 3. A different version of the cruise control problem. The bank transfer problem previously used for deadlock detection would not be interesting as the designated target thread might never access the designated target lock (because of the random nature of thread accesses to accounts in the design of this system).

7.2.1 Modified Dining Philosophers Problem (ModPhil)

Notice that for the dining philosophers problem as defined for deadlock detection above, only two threads access each lock. This would make for a rather simple test for starvation. To test for starvation, the problem is therefore altered slightly as shown in Figure 19: 80 philosophers (squares) are sitting in two concentric circles, with 40 forks (circles) shared between them, such that every two philosophers share the same two forks. Hence, each fork is accessed by four philosophers. The aim again is that philosophers use their two right and left forks when they are hungry and want to eat. In our modified philosopher design solution, philosophers are assigned one of two priorities, low or high. Philosopher 1 is assigned low priority while others are randomly assigned either a higher priority than philosopher 1, or equal priority. A philosopher can starve in this solution if the surrounding three philosophers constant-

ly pick up the shared fork before the philosopher has the chance to do so. The state space[5] here is approximately 9.8 * 10^{37}. The actual search space is calculated much like in Phil. Each philosopher has 100 choices (the thinking time) for accessing their left fork and only one choice for accessing their right fork. During the chosen time interval of 400 (see below), each philosopher can access their forks at least four times. Hence, per philosopher, the total number of combinations is: $(100*1)^4$. But there are 80 philosophers, so there are 100^{4*80} combinations, so the search space size amounts to 1.0 * 10^{640}.

To test for starvation Fork 1 and Philosopher 1 are the designated target lock and thread, respectively. The time interval used is 400 as for deadlock detection (recall from Section 5.2.4 that there is no associated heuristic: it is left up to users to determine an appropriate interval).

The chromosome size here is larger than for Phil. Although the same time interval is used, there are now 80 philosophers, each accessing two locks four times each. Thus, the chromosome size is: 80 * 4 * 2 = 640.

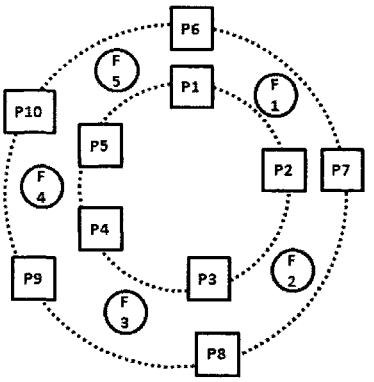

Figure 19: Modified dining philosophers for starvation

[5] The state space here is an approximation: $3^{\#phil} - 3^{\#phil-1}$. The first term is like the one for deadlocks. However, there are now more illegal situations (where two philosophers are holding the same fork) for an additional #phil-1 philosophers.

7.2.2 The Starvation Problem (Starve)

The Starve problem is an artificial problem created to provide a more challenging environment for starvation detection. Six threads access a common lock with properties defined in Table 4.

T6 is the target thread. It has the lowest priority, while T1 has the highest priority. The time interval chosen is [0-24]. Here, the search space is 4.8 X 10^8. There are 4.8 X 10^8 different ways the threads can combine to execute within the time interval: For T1, it can access the lock between time units [1-2], [8-9], [15-16] and [22-23]. T2 can access the lock between time units [0-2], [9-11] and [18-20]. T3 can access between [3-9], [10-16], [17-23] and [24]. T4 can access between [4-9], [11-16], [18-23]; T5 between [2-6] and [22-24]. T6 can only access the lock at time unit 3. There are two choices for each thread access range of locks for T1, three choices for T2, 7 for T3, 6 for T4, 5 for T5 and only one for T6. Hence, the total number of combinations is: 16 (2*2*2*2 for T1) * 27 (3*3*3 for T2) * 343 (7*7*7*1 for T3) * 216 (6*6*6 for T4) * 15 (5*3 for T5) * 1 (for T3) = 4.8 X 10^8. Here, the target thread will starve if every time slot after time unit 3 is occupied by another thread. This happens in 704,295 combinations[6]. Hence, only 0.14% of the population results in starvation.

The size of the chromosome here can be calculated as follows: In the interval of 24, T1 can access the lock $\left\lceil \frac{24}{7} \right\rceil$ times, T2 $\left\lceil \frac{24}{9} \right\rceil$ times, T3 and T4 $\left\lceil \frac{24}{7} \right\rceil$ times each, T5 $\left\lceil \frac{24}{20} \right\rceil$ times and T6 $\left\lceil \frac{24}{24} \right\rceil$ times.

Summing these yields 18, which is the length of the chromosome

[6] Because it could not be done analytically, a computer program was developed to calculate the number of combinations that would lead to starvation.

Table 4: Thread properties for the starvation example

Thread	Priority	Lock access range (units of time)	Repetition time (units of time)	Execution time (units of time)
T1	32	1-2	7	1
T2	31	0-2	9	2
T3	30	3-9	7	1
T4	29	4-9	7	2
T5	28	2-6	20	2
T6	27	3	24	2

7.2.3 Modified Cruise Control Problem (ModCruise)

The same cruise control problem, while simple, is used for starvation. However, to test for starvation, the `CarSimulator` thread is designated the target thread, with low priority, and the target lock is `brakepedal`. The time interval used here is 24. With this time interval, the length of the chromosome is: $\left\lceil \frac{24}{7} \right\rceil$ * 2 (for `SpeedControl`'s access of two locks) + $\left\lceil \frac{24}{15} \right\rceil$ * 2 (for `CarSimulator`'s access of two locks) = 12.

7.3 Data Race Case Study – Therac-25 (Therac)

The case study we use for data race detection was inspired from the Therac-25 machine - a computer controlled radiation therapy machine that was responsible for overdosing six patients [Leve 95]. Investigations into the causes behind the overdoses revealed faults due to race conditions, whereby the high power electronic beam was activated (instead of the low power one), without the beam spreader plate rotated into place [Leve 95]. We use an altered version of the original design to provide a larger search space, thus representing a larger scale of problems.

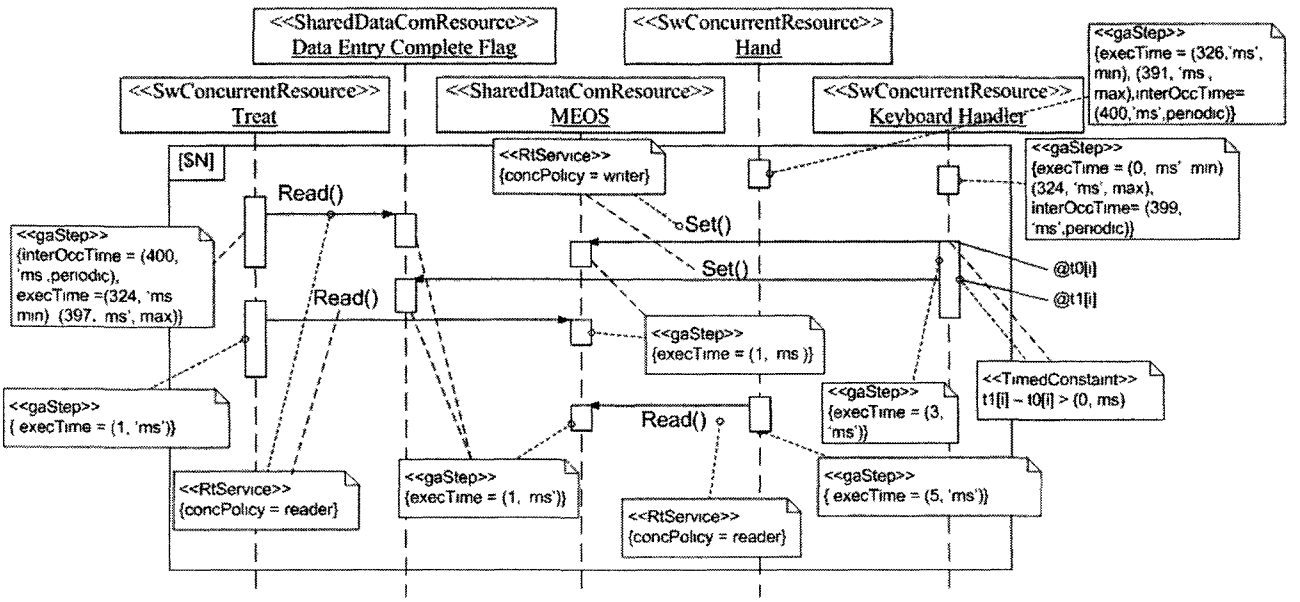

Figure 20: Therac sequence diagram

Figure 20 shows the UML/MARTE sequence diagram of the shared resources in Therac. The treatment monitor task, `Treat`, controls the phases of radiation treatment. It uses the `Tphase` control variable to determine which phase of the treatment is to be executed next. In the first phase, a check is performed to see whether the required radiation levels have been input. This check is performed on a variable named data entry complete flag (DECF), which is set by the keyboard handler task, where the operator enters radiation level information. DECF is set whenever the cursor is moved to the command line. Information about the radiation level specified by the operator is encoded into a two byte variable named MEOS (Mode/Energy offset). The higher byte of MEOS is used by `Treat` to set various parameters. The lower byte is used by `Hand`, which rotates the turntable according to the inputted energy and mode.

In the figure, two resources, MEOS and DECF, are shared as indicated by <<SharedDataComResource>>. The former is accessed by the three available threads designated with the <<SwConcurrentResource>> stereotype. The latter resource is only accessed by the Treat and KeyboardHandler threads. Treat periodically reads DECF between [324-397] and repeating at 400 unit intervals. KeyboardHandler is a writer thread on the same resource. The same thread also accesses MEOS as a writer thread. However, the write access to MEOS occurs before the write access to DECF; there is at least a one second interval. The Hand thread accesses MEOS periodically every 400 milliseconds.

The search space differs for the two shared resources used. For MEOS, it is based on the access time intervals of the Treat (executing between [325-398] ms and repeating every 400 ms), Hand (executing between [327-392], repeating every 400 ms) and KeyboardHandler (executing between [1-325], repeating every 399 ms) threads as well as the timing interval. For a timing interval of 802 time units (based on our heuristic, Section 5.3.4), the search space is approximately $9.8 * 10^{12}$: Within the timing interval, Treat executes between [325-398] and [725-798]; Hand between [327-391] and [727-791]; KeyboardHandler from [1-325], [400-724] and [799-802]. The search space is the cross product of the various repeating thread choices (74*74*65*65*325*325*4). Of these, $4.1 * 10^8$ yield a data race[7] (i.e. 0.004%). For DECF, Treat accesses the shared resource between [324-397], repeating at 400 ms intervals while KeyboardHandler's access occurs between [1-325], repeating at 400 ms intervals, making the search space $2.3 * 10^9$ (74*74*325*325*4 from the cross product of the various repeating

[7] The number of sequences leading to data races was calculated using a program for all search spaces.

threads) with 360,750 resulting in a data race (i.e. 0.02%). In terms of complexity, results presented in Section 7.5 indicate that the MEOS search space is complex, with many local optima. For DECF, the search space appears to have few local optima, thus simplifying the search.

The chromosome lengths for MEOS and DECF also differ. For MEOS, the length of the chromosome is defined by: $\left\lceil \frac{802}{399} \right\rceil$ for `KeyboardHandler` plus $\left\lceil \frac{802}{400} \right\rceil$ for `Treat` and `Hand`. The chromosome length is thus 9. For DECF, the chromosome length is defined by $\left\lceil \frac{802}{399} \right\rceil$ for `KeyboardHandler`, plus $\left\lceil \frac{802}{400} \right\rceil$ for `Treat`. The chromosome length here is thus 6.

To further enhance our case study, we altered the access ranges of threads in both resources to create two more different search spaces where detecting data races is significantly more difficult. For the altered MEOS resource, or simply MEOS2, with a timing interval of 2500 time units, the access ranges are: [1050-2049] repeating every 2000 ms for `Treat`, [1051-1060] repeating every 2000 ms for Hand, and [1-1050] repeating every 2049 ms for `KeyboardHandler`. The search space here is approximately 4.7 * 10^9 (1000*10*1050*451 from the cross product of the various repeating threads) with 4510 yielding a data race. Hence, 9.5 * 10^{-5}% of the search space results in a data race. For the altered DECF, or simply DECF2, with a timing interval of 800, access times are: [350-650] repeating every 500 ms for `Treat` and [2-350] repeating every 700 ms for `KeyboardHandler`. The search space here is 1.0 * 10^7 (301*349*99 from the cross product of the repeating threads) with 99 resulting in a data race. This translates into 9.9 * 10^{-4}% of the search space leading to a data race. In terms of complexity,

results (Section 7.5) indicate that for the MEOS2 resource, the search space appears complex, with many local optima. For DECF2, results suggest its search space has few local optima. With these altered search spaces, we can then better assess how the performance of search techniques is affected by the difficulty of the search. The chromosome lengths for the altered resources are also different. MEOS2 has a chromosome length of 6 ($\left\lceil \frac{2500}{2000} \right\rceil + \left\lceil \frac{2500}{2000} \right\rceil + \left\lceil \frac{2500}{2049} \right\rceil$), while DECF2 has a chromosome length of 4 ($\left\lceil \frac{800}{500} \right\rceil + \left\lceil \frac{800}{399} \right\rceil$).

7.4 Design of Case Studies

We begin by briefly describing the techniques used for the various fault detections, then relating how all case studies were set up to ensure that all techniques are comparable.

7.4.1 Description

We use three different techniques to detect deadlocks, starvations and data races: random generation, hill climbing and our GA approach. Both random and hill climbing are simpler techniques that are often suggested as benchmarks to justify the need for a GA search [Ali 09]. Hill climbing is also often used as a quick and efficient way of determining the structure and complexity of the landscape [Harm 09].

In random generation, a point in the search space (generated in such a way that it satisfies the same timing constraints imposed on the genes when using GAs) is randomly chosen and checked for a fault.

Running a random search involves running a pre-determined, usually large number of points in the search space.

For hill climbing, we use stochastic hill climbing [Rose 99]. One random point is generated, then a neighboring point is generated by mutating the current point. Hence, a neighbor differs from the current point by only one gene. If the new point is better than the current point, it replaces it and a new neighboring point is generated. If no point is better than the current point, execution stops. This continues (starting with a new random point selected) until a maximum number of points, or sequences (see below in Section 7.4.2) are generated [Rose 99]. For example, consider three threads, T1, T2 and T3, accessing a shared resource during the access time intervals [1-2], [3-5] and [6-7], respectively. If the current point is (T1, 2) (T2, 3) (T3, 6), one valid neighbor would be: (T1, 1) (T2, 3) (T3, 6) which differs in only one access time value from the current point.

All case studies were run on the same computer: an Intel Core 2 2.0 GHz processor with 2.0 GB of RAM.

7.4.2 Fairness of Comparisons

Because each of the three techniques proceeds differently, we wanted to ensure fair comparisons. We do this in two ways: 1.) By comparing numbers of sequences and 2.) By comparing time budgets. In comparing numbers of sequences and time budgets, as described below in Sections 7.4.2.1 and 7.4.2.2, we ensure that comparable results are produced. As GAs are heuristic optimization techniques, variance occurs in the results they produce. To account for the variability in results, we initially ran each case study 25 times. Twenty five runs is a reasonable number for the case studies provided: they ex-

ecute within reasonable time, and they turned out to be sufficient to find the cases of deadlocks, starvation and data races at least once in all case studies. Random generation and hill climbing were also run 25 times. To ensure that the differences in detection rates between our GA technique and the other techniques were not due to sampling, we conducted a Fisher Exact hypothesis test to assess differences in proportions. This is a test commonly used to analyze statistical significance for small data sets. The output of the hypothesis test is a probability value, or p-value, which gives an indication of the probability of obtaining test statistics as extreme as the ones observed in the data set. The lower the value, the less likely this is the case. The Fisher Exact hypothesis test we used uses two-tailed p-values[8]. Widespread practice considers p-values below 0.05 to be significant. A number of p-values suggested that the differences between the detection rates are not significant. We thus increased the number of runs to 100 in phase two. In phase two, all case studies, except for Phil, were run 100 times. Phil takes a much longer time to run and results are sufficiently clear and statistically significant with 25 runs.

7.4.2.1 Comparing Numbers of Sequences

Here, we analyzed the number of chromosomes, or sequences generated by the GA and generated the same number for other techniques to ensure a fair comparison. All sequences for both random and hill climbing are generated in such a way that they satisfy the same timing constraints imposed on the genes when using a GA. For random and hill climbing, each run generated the same number of sequences as the number created and evaluated on average by a GA run. For example, with a timing interval of 400, a GA run generates on average 5,922 sequences per run for Phil during deadlock detec-

[8] P-Values calculated with GraphPad Software, available online from http://www.graphpad.com/quickcalcs/contingency2.cfm.

tion. Hence, 5,922 sequences were generated per run for both random search and hill climbing. To further ensure fairness of comparison, for random, hill climbing and the GA, when a deadlock or starvation is detected, execution stops for that run and a new run is executed.

7.4.2.2 Comparing Time Budgets

Here, we noted execution times of our GA technique and allowed both hill climbing and random detection to run for the same time budget. This is another means of comparison. So, if, for example, our GA approach detected 20 data races in 20 minutes, we ran both hill climbing and random generation for a 20 minute time slot, noting the number of data races detected by each. Here, we only used execution times for phase two, as both phase one and phase two execution times were roughly proportional.

7.5 Results

Results for the detection of deadlocks, starvation and data races for phase one, for random generation (RA), our approach (GA) and hill climbing search (HC) are presented in Table 5. Many p-values (with values greater than 0.05) indicate that the differences between the approaches are not significant and may be due to sampling. To determine whether these differences are truly insignificant, we ran the case studies with 100 runs for each of the different techniques. Results for phase two are presented in Table 6. Detection rates are presented in both tables. Detection rates are the percentage of faults detected in the number of runs. It is important to note that the detection rates across phases one and two are consistent. For example, the detection rate in HC for the three starvation case studies are: 84% for ModPhil, 88% for Starve and 92% for ModCruise. In phase two, when the number of runs is quadrupled, the detection rates are fairly close at 100%, 86% and 100%, respectively. Furthermore, execution times

appear proportional: For RA's starvation case studies, for example, execution times in phase one were roughly 0.5 seconds, 3.5 seconds and 132 milliseconds across the three case studies. Quadrupling the amount of time resulted in roughly quadruple the execution time at 1.5 seconds, 14 seconds and 432 milliseconds. For these reasons, we use only phase two results in the remainder of our analysis for sequence comparisons.

Looking collectively at the results, one can draw some interesting conclusions. In nine of the case studies, out of ten, the GA has the highest detection rate across all techniques. Even in the case where it does not have the highest detection rate (DECF2), it's detection rate is rather high (91%) However, when timing budgets are imposed on designers, our GA approach does not always offer the highest number of fault detections. Most of the time, RA or HC are capable of detecting more faults within the same amount of time as the GA. For practical purposes, designers only need one instance of a fault to show that the design may trigger that fault. Hence, only one instance of any fault is sufficient. Furthermore, when the search space size and complexity is not known, as is often the case in practice, GAs are the best of the three techniques to use as they provide good detection rates.

In terms of variety on reported faults, sometimes, as in the case of Starve, several of the reported faults were identical. At other times, as in the case for Phil, the reported faults were different.

7.5.1 Deadlock

For deadlock detection, most of the three techniques are capable of uncovering a fault for both comparison techniques, but with very different rates.

7.5.1.1 Deadlock: Comparing Numbers of Sequences

For Cruise, the case study with a very small search space, random search is expected to be nearly as effective as the GA. Indeed, the GA mostly detects a fault in the random initialization of the population and both techniques detect a fault in 100% of the cases. However, hill climbing does not perform well as shown in Table 6: 70% of the time, it is unable to detect a deadlock, and the difference with GA is statistically significant. This is because its performance depends on the initial randomly generated sequence for each run. Recall that, to ensure fair comparison, the number of sequences generated by the GA was analyzed and the same number of sequences was generated for hill climbing. If the initial hill climbing chromosome is very different from the optimal chromosome, it will take many mutations to reach the optimum. Hence, hill climbing may exhaust its allocated number of sequences. In Bank, which has a larger search space, the GA outperforms hill climbing and random search. For Phil - which has by far the largest search space - the GA significantly outperforms again both random search and hill climbing, but to a much larger extent. It is interesting to note that other than the search space size, the conditions for a deadlock are more stringent in Phil than Bank: All 40 philosophers must be accessing all left forks simultaneously, versus any two accounts with inverted source and destinations would suffice to create a deadlock. This makes the search harder in Phil than in Bank and, expectedly, execution time is substantially increased in Phil. The search space in Phil appears to be very complex, as indicated by the very low detection rate of RA. The GA, however, is much less affected by the complexity of the search space whereby the detection rates for these two case studies are much closer (81% vs. 88%) to each other than for random search (0% vs. 70%). Hill climbing's overall detection rate does not vary much in Phil and tends to be poor overall. From the above results we can conclude that

the GA effectively performs as well or better than the two other alternatives and that the differences are driven by the size and complexity of the search space.

Table 5: Phase one results

		Phil	Bank	Cruise	ModPhil	Starve	ModCruise	MEOS	DECF	MEOS2	DECF2
	Fault type	**Deadlock**			**Starvation**			**Data Race**			
	Search space	$1.0 * 10^{320}$	$2.7 * 10^{66}$	245	$1.0 * 10^{640}$	$4.8 * 10^{8}$	175	$9.8 * 10^{12}$	$2.3 * 10^{9}$	$4.7 * 10^{9}$	$1.0 * 10^{7}$
RA	#Faults/#Runs	0/25	19/25	22/25	25/25	4/25	22/25	1/25	5/25	0/25	1/25
	Total Runtime (hr:min:sec:ms)	0:11:40:697	0:02:06:190	0:00:00:133	0:00:00:560	0:00:03:560	0:00:00:132	0:00:33::581	0:00:22:124	0:02:14:419	0:00:55:418
	Detection rate	0%	76%	88%	100%	16%	88%	4%	20%	0%	4%
	p-value	< 0.0001	0.4635	0.2347	1.0	< 0.0001	0.2347	0.0232	< 0.0001	0.4898	< 0.0001
GA	#Faults/ #Runs	22/25	22/25	25/25	25/25	24/25	25/25	8/25	24/25	2/25	21/25
	Total Runtime (hr:min:sec:ms)	2:34:34:482	0:08:31:009	0:00:00:222	0:00:01:266	0:00:08:102	0:00:00:166	0:00:47:155	0:00:30:040	0:02:50:349	0:01:23:360
	Detection rate	88%	88%	100%	100%	96%	100%	32%	96%	8%	84%
HC	#Faults/ #Runs	7/25	6/25	7/25	21/25	22/25	23/25	1/25	25/25	1/25	25/25
	Total Runtime (hr:min:sec:ms)	0:48:05:368	0:05:45:499	0:00:00:175	0:00:09:123	0:00:02:341	0:00:00:143	0:00:27:480	0:0:06:462	0:02:19:032	0:00:07:047
	Detection rate	28%	24%	28%	84%	88%	92%	4%	100%	4%	100%
	p-value	< 0.0001	< 0.0001	< 0.0001	0.1099	0.6092	0.4898	0.0232	1.0	1.0	0.1099

Table 6: Phase two results

		Phil	Bank	Cruise	ModPhil	Starve	ModCruise	MEOS	DECF	MEOS2	DECF2
	Fault type	**Deadlock**			**Starvation**			**Data Race**			
	Search space	$1.0 * 10^{320}$	$2.7 * 10^{66}$	245	$1.0 * 10^{640}$	$4.8 * 10^{8}$	175	$9.8 * 10^{12}$	$2.3 * 10^{9}$	$4.7 * 10^{9}$	$1.0 * 10^{7}$
RA	#Faults/#Runs	0/25	70/100	100/100	100/100	17/100	100/100	3/100	17/100	0/100	1/100
	Total Runtime (hr:min:sec:ms)	0:11:40:697	0:04:05:568	0:00:00:489	0:00:01:546	0:00:13:924	0:00:00:432	0:02:12:521	0:01:26:112	0:07:08:343	0:04:18:563
	Detection rate	0%	70%	100%	100%	17%	100%	3%	17%	0%	1%
	p-value	< 0.0001	0.0996	1.0	1.0	< 0.0001	1.0	< 0.0001	< 0.0001	0.0140	< 0.0001
GA	#Faults/ #Runs	22/25	81/100	100/100	100/100	98/100	100/100	33/100	97/100	7/100	91/100
	Total Runtime (hr:min:sec:ms)	2:34:34:482	0:15:20:322	0:00:00:552	0:00:02:558	0:00:35:292	0:00:00:534	0:03:12:951	0:02:03:445	0:11:20:209	0:06:00:360
	Detection rate	88%	81%	100%	100%	98%	100%	33%	97%	7%	91%
HC	#Faults/ #Runs	7/25	30/100	30/100	100/100	86/100	100/100	6/100	97/100	1/100	100/100
	Total Runtime (hr:min:sec:ms)	0:48:05:368	0:10:45:499	0:00:06:238	0:01:52:168	0:05:06:768	0:00:00:516	0:04:31:023	0:00:46:158	0:09:16:713	0:00:37:959
	Detection rate	28%	30%	30%	100%	86%	100%	6%	97%	1%	100%
	p-value	< 0.0001	< 0.0001	< 0.0001	1.0	0.0029	1.0	< 0.0001	1.0	0.0649	0.0032

Table 7: Results using time budget

		Phil	Bank	Cruise	ModPhil	Starve	ModCruise	MEOS	DECF	MEOS2	DECF2
	Fault type	**Deadlock**			**Starvation**			**Data Race**			
	Search space	$1.0 * 10^{320}$	$2.7 * 10^{66}$	245	$1.0 * 10^{640}$	$4.8 * 10^{8}$	175	$9.8 * 10^{12}$	$2.3 * 10^{9}$	$4.7 * 10^{9}$	$1.0 * 10^{7}$
	Runtime (hr:min:sec:ms)	2:34:34:482	0:15:20:322	0:00:00:552	0:00:02:558	0:00:35:292	0:00:00:534	0:03:12:951	0:02:03:445	0:11:20:209	0:06:00:360
RA	#Faults	0	239	207	179	55	280	11	49	0	10
GA	#Faults	22	81	100	100	98	100	33	97	7	91
HC	#Faults	23	45	2	5	10	195	10	469	1	1186

7.5.1.2 Deadlock: Comparing Time Budgets

The results for time budgeting in Table 7 closely mimic those of sequencing. For example, RA detects the most deadlocks for Bank and Cruise in sequencing. By increasing the execution time for time budgeting, the number of deadlocks detected also increases, with Bank and Cruise still detecting the greatest number of faults.

Given the same time budget, random detection is capable of finding nearly twice as many deadlocks for Cruise as the GA. Again, this was expected for such a small search space. Hill climbing continued to perform poorly, detecting only two deadlocks, versus the GA's 100 detections. In Bank, random detection again flourishes finding nearly three times as many deadlocks as the GA, while hill climbing only detects half as many as the GA. The only case where hill climbing flourishes is in Phil. Here, it performs as well as the GA, with random detection clearly lagging behind.

7.5.2 Starvation

For starvation, all three techniques are capable of uncovering a fault for both comparison techniques. Their rates, however, vary.

7.5.2.1 Starvation: Comparing Numbers of Sequences

It is not surprising that the three techniques produce perfect detection rates for ModPhil and ModCruise, as seen in Table 6 suggesting that these problems do not really need a GA to obtain starvation sequences. Indeed, it appears that the search space in these case studies is not complex (see Section 7.1.1), as indicated by RA's perfect detection rate. Though the search space is large in ModPhil, there

are many sequences leading to starvation, as the target lock is only accessed by four threads. As a result, the GA is capable of detecting a fault in its initial random generation of the population. In ModCruise, with the smallest search space, only two threads access the target lock. Here too, the number of sequences leading to starvation is substantial, which is why both random and hill climbing fare well. In Starve, because so few sequences lead to starvation, random search cannot detect a problem 83% of the time whereas this is the case only 2% of the time with GA. Based on 100 runs, p-values reveal that the GA significantly outperforms hill climbing as well. So, like for deadlocks, we see that the relative performance of our GA depends on the size and complexity of the search space. However, even when the search space does not appear to be complex (as in ModPhil and ModCruise), the GA performs at least as well as the two other alternatives.

For both ModCruise and ModPhil, false positives (Section 5.2.4) are not a concern: while the GA reports that the target threads for both case studies access the target locks at least once, subsequent accesses are denied. Further tests conducted by doubling the time interval produced the same results, thus indicating the absence of false positives. For Starve, increasing the time interval to [1-100] results in no faults being reported, hence indicating that all previous results were false positives. However, since Starve is an artificial problem, we can equally assume that performance constraints are violated and treat the false positive as an actual starvation case. For this reason, it is important to note that detection rates for Starve include false positives, not the results obtained by increasing the time interval.

7.5.2.2 Starvation: Comparing Time Budgets

When comparing time budgets, Table 7 shows that ModCruise results in an influx of detections for random search (280 detections) and hill climbing (195 detections). This is expected due to the size of the search space coupled with the fact that each technique is now allowed a larger time slice of execution than during sequence comparisons. Sequence comparisons resulted in perfect detection rates, so increases in execution time result in increased detections. In ModPhil, the number of random detections increases to 179 detections from the previous 100, while the number of hill climbing detections decreases down to 5 detections from the previous 100 during sequencing. The decrease in hill climbing detections is due to a decrease in execution time. Hill climbing was capable of detecting 100 faults in under two minutes during sequence comparisons, yet the execution time frame was decreased to a little over two seconds for time budget comparisons. In Starve, the case study with few sequences leading to starvation, neither hill climbing nor random generation fare as well as the GA. Random generation detects 55 faults. This is roughly half the 98 faults detected by the GA. Hill climbing fares even worse with only 10 starvation faults detected versus 98 with the GA when both are run for the same amount of time.

7.5.3 Data Race

Here again, random generation, hill climbing and the GA approach can detect faults in most of the case studies, but with different probabilities.

7.5.3.1 Data Race: Comparing Numbers of Sequences

In Table 6, for MEOS, hill climbing does not fare very well. It appears to be oftentimes caught in local optima: 94% of the time, it is unable to detect a data race. This, coupled with RA's low detection rate, empirically suggests that the search space for MEOS is complex, as hill climbing is known to perform well in search spaces characterized by few local optima. We observe that our GA does better: 33% detection rate for MEOS, with p-values indicating differences with hill climbing are statistically significant. This confirms that where the search space is large and complex, GAs are known to yield much better results than the two other techniques [Mahf 95]. Complexity is not an issue in random search because information about the landscape of the search space is not used during the search. However, random search performs poorly in MEOS due to the small percentage of sequences leading to a data race: only 0.004% of the search space yields a data race.

On the other hand, in the case of the simpler DECF search space, hill climbing does exceedingly well, detecting data races in almost all runs. This, along with RA's low detection rate, suggests that the search space has few local optima. Random search performs better than it did for MEOS, with a 17% detection rate versus its previous 3% in MEOS. This increase is due to the higher percentage of sequences leading to a data race (0.02% versus the previous 0.004%). With both techniques capable of detecting data races relatively well, a GA is therefore of no benefit in this case, although it too performs well.

For MEOS2 and DECF2, results suggest their search spaces are of similar complexity as MEOS and DECF, respectively, but with smaller sizes and lower percentages of sequences leading to a data race. For MEOS2, the search space is large and complex, with $9.5 * 10^{-5}$% (see Section 7.3) of sequences

leading to a data race, which is why both random and hill climbing perform very poorly. The GA, while performing worse than for MEOS, still manages to detect data races seven times as much as hill climbing, with p-values indicating differences being slightly significant. In DECF2, random performs worse because of the lower percentage of data race sequences ($9.9 * 10^{-4}\%$). The GA too performs worse than for DECF.

In all MEOS cases, the GA far outperforms both random and hill climbing techniques. This confirms that it fares much better in large, complex search spaces, and is therefore a better option in many practical cases where such characteristics are likely to be present. For both cases of DECF, where the search space is smaller and less complex, the GA detection rate is somewhat comparable to hill climbing, which is designed for such search spaces.

In large, complex search spaces, where few sequences yield data races, the GA yields significantly higher detection probabilities than other techniques. Because these probabilities for a run can still remain low, the GA must be run as many times as possible, given time budgets, to obtain the highest possible overall probability of detecting data races. Using the most complex case (MEOS2) as an example, with a 7% probability of data race detection, 100 GA runs results in a probability of less than 0.07% (0.93^{100}) not to detect a data race in at least one run, with less than eleven minutes and a half of execution time. Such execution times can of course be brought down significantly with faster hardware and parallel computing. Even when the complexity of the search space is not known and it is not clear what percentage of this space results in a data race, using the GA will in the worst case yield comparable detection rates to hill climbing.

7.5.3.2 Data Race: Comparing Time Budgets

As expected, the execution time of the GA is longer than the other techniques. However, it is interesting to note that given the same time budget, the GA still far outperforms the other two techniques in terms of number of detections except in two cases: DECF and DECF2. In both these case studies, hill climbing is capable of detecting many more data races than the GA. This is in line with the findings in the previous section where hill climbing appears well suited for these two particular case studies. For the remainder of the case studies, the GA outperforms the other two techniques.

7.5.4 Execution Time Analysis

Though we see from Table 6 that the GA systematically detects deadlocks, starvation and data races in all case studies, and with a probability that is at least as good as with random search and hill climbing, for large search spaces, it takes substantially more time for the GA to detect faults than for the other techniques. Execution times in both starvation and data race detection, as well as deadlock detection are driven by a number of factors. In the former case, the number of threads accessing the target lock or shared resource contributes to the overall execution time: the greater the number of threads, the longer it takes to generate sequences where all threads are accessing the target lock or shared resource. The total number of threads executing in a system has a similar effect on execution time in the case of deadlock detection. Here, the greater the total number of threads, the longer it takes to generate sequences where threads produce a deadlock. Execution times for the GA are still reasonable, considering the sizes of the search spaces (even with the most complex search space for deadlock Phil). As seen from Table 7, allowing the other techniques to run as long as the GA does not always result in better performance over the GA. Assuming designers would use a verification plug-in based on their

MARTE designs, using our approach would take, in the worse cases (like Phil) and based on low-end hardware, a couple of hours or so. Such performance could, as discussed next, easily be improved by using clusters or distributed computers. Given that such analyses are not frequently required, execution times for running the GA to find concurrency problems are likely to be practical. Furthermore, despite the increased execution times, since in practice the search space size and complexity is not known, using GAs ensures to obtain the best detection rate and is therefore the best option.

7.6 Scalability

Scalability is a quality that is often touted as being highly desirable in the field of computing. However, scalability is often hard to define [Bond 00]. Bondi defines primarily four dimensions of scalability: load, space, space-time and structural. Systems with load scalability are ones that have "the ability to function gracefully, i.e., without undue delay and without unproductive resource consumption or resource contention at light, moderate, or heavy loads while making good use of available resources" [Bond 00]. Space scalability refers to systems' memory requirements. So long as a system's memory requirements do not grow to intolerable levels, the system is defined as having space scalability. In terms of space-time, a scalable system has the ability to grow, or expand, as the size of the problem increases. Structural scalability is used to refer to system enhancement by adding additional functionality with little effort. In this context, a system is scalable if it can be easily enlarged.

Recall from Section 5 that for starvation, data race and deadlock detection, the chromosome representation and the mutation and cross-over operators are common with the main difference between them being the fitness function. This illustrates that the whole approach is easy to adapt from one type of

concurrency problem to another, with adaptations performed mainly on the fitness function. Hence, we deem our approach structurally scalable whereby additional functionality can be added with little effort. To determine the remaining facets of scalability of our approach and tool, we ran the Phil problem with varying sizes of philosophers and forks. We chose this problem in particular because it represents a highly complex situation where deadlocks are difficult to detect. The number of philosophers used ranges from 5 to 60. This range is enough to establish a pattern in terms of fault detection rates and execution times, as shown below. Table 8 shows our results in terms of detection rate and execution time when run on our CFD tool.

Table 8: Results with varying numbers of philosophers

Number of Philosophers	**#Faults / #Runs**	**Execution time (hr:min:sec:ms)**
5	23/25	00:01:14:157
10	25/25	00:00:13:145
15	22/25	00:16:05:498
20	21/25	00:21:50:543
25	21/25	01:13:44:087
30	18/25	01:58:43:955
35	17/25	04:42:55:234
40	22/25	02:34:34:482
45	19/25	08:43:30:209
50	20/25	07:25:57:740
55	17/25	10:21:14:432
60	19/25	13:38:01:508

None of the variations of the problem ran out of memory. They were all able to fully execute. From this, it can be argued that space scalability is achieved whereby 2.0 GB of RAM were enough to run all variations of the problem without the system crashing. As for the detection rate, the lowest detection rate is 17/25 (68%) with the highest being 25/25 (100%). Figure 21.a graphically illustrates the detec-

tion rate of Table 8. It shows the result of a regression analysis between the number of deadlocks and philosophers: there is a significant, negative linear relationship ($R^2 = 0.5$) showing an average of one less deadlock detection (out of 25 runs), or a decrease of 4% in deadlock detection rate, per additional ten philosophers. The decrease is linear and takes an additional 10 threads accessing a specific lock for a 4% decrease in detection rate. Our approach is thus scalable in terms of space-time: it is readily able to handle growing search spaces without very high variance in detection rate. It is clear, though, that for large search spaces, more runs will be necessary to ensure high probabilities of detection.

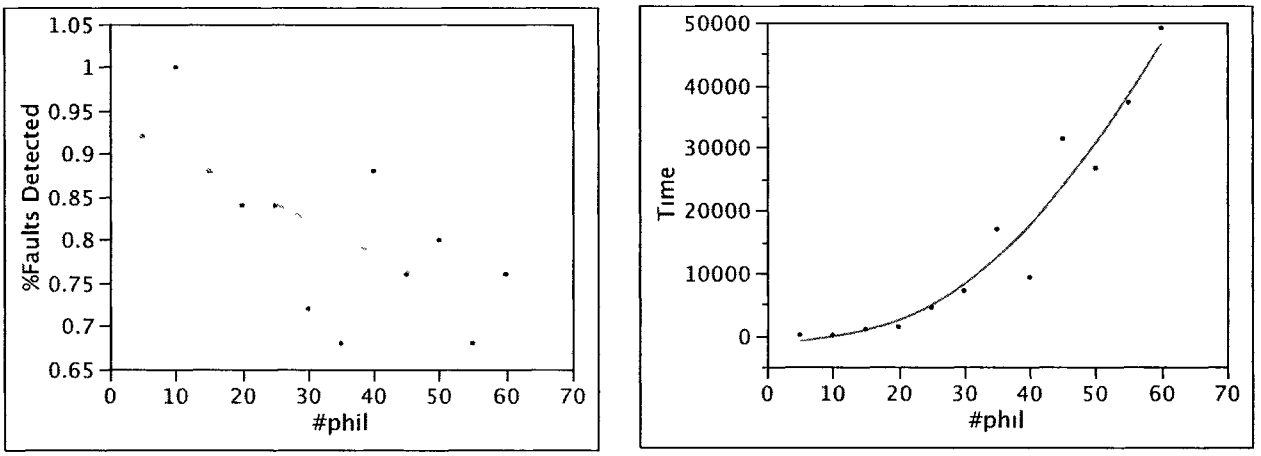

Figure 21: Scalability in terms of (a) fault detection and (b) execution time in seconds

As expected, and as seen from Table 8 and the scatter plot of Figure 21.b, execution time increased with an increase in the number of philosophers, indicating that our tool does not have load scalability. The scatter plot of Figure 21.b shows an overall exponential increase in execution time as the number of philosophers increases. Given the low-end hardware on which this was run, the fact that for a highly complex search problem (60 Philosophers) 25 runs take 13 hours is still manageable. However, not much else can be concluded from this since there is much room for improvement by using more po-

werful hardware and especially distributed GAs [Dori 93]. For systems with large numbers of threads, such as 50 and 60 philosophers in our study, given the availability of faster and cheaper hardware, parallel GAs can be exploited at little additional cost [Dori 93]. The nature of genetic algorithms lends itself easily to distribution simply by allowing populations to evolve in parallel [Dori 93]. The gains of distribution strategies in terms of execution time are promising, as shown in [Wang 98]. For the traveling salesman problem, execution times dropped by more than 99.5% for some variations of distributed GAs (from 394 seconds down to 1.76 seconds) without a decrease in effectiveness [Wang 98]. The use of distributed GAs will, however, be investigated and reported in future work as it requires large scale studies, such as the ones in [Wang 98], assessing and comparing various strategies for distribution. For example, depending on the type of hardware machines available for distribution, two approaches of parallel GAs can be used: the island model and the fine-grained, or neighborhood model. In the former case, isolated subpopulations evolve on separate machines, while periodically migrating their best chromosomes with neighboring subpopulations. In the neighborhood model, a single population evolves, but selection and crossover is based on neighboring chromosomes placed on a planar grid. The island model runs well on Multiple Instruction Multiple Data (MIMD) machines, while the neighborhood model runs well on Single Instruction Multiple Data (SIMD) machines [Dori 93].

8 CONCLUSION AND FUTURE WORKS

Concurrency abounds in many software systems. Such systems usually involve threads that access shared resources, and complex thread communications. If not handled properly, such accesses can lead to many problems, such as starvation, deadlock and data race situations, which may hinder system execution. It is important to detect such problems as early as possible and therefore find practical, scalable ways to do so. We have presented a method based on the analysis of design representations in UML completed with the MARTE profile, both of which are international standards for the object-oriented modeling of concurrent, real-time applications that are widely supported by commercial and open source tools. We demonstrated the feasibility of using concurrency information from UML/MARTE diagrams to detect concurrency problems based on search algorithms, in this case a tailored genetic algorithm. Our automated verification method can then be applied in the context of model-driven, UML-based development and thus reduces the need for complex tooling and training, and additional modeling to what is already required for UML-based development.

Being geared towards systems characterized by large numbers or threads and complex synchronization among them, our method treats the detection of concurrency problems as a search and optimization problem. Because of their well-documented track record for global search in complex landscapes, genetic algorithms are then used to search through the space defined by access times. The best resulting sequence, measured by the fitness function, is then outputted. Further examining the length of the outputted sequence (e.g., if it is the shortest one leading to a failure), as well as the extent to which se-

quences can help with debugging is reserved for future work. Also in future works, use of a genetic algorithm can be compared to other heuristic techniques, such as simulated annealing.

Our approach provides a general framework that can be adapted to different types of concurrency problems: originally developed for deadlock detection, adapting it to the problems of starvation and data race detection mainly required the definition of a new fitness function in the genetic algorithm.

We demonstrated the effectiveness of our approach (in terms of concurrency fault detection) through ten case studies run on our tool support. These showed very promising results in terms of detecting deadlocks, starvation and data race problems. Even in the presence of large search spaces, our tool's running time is of the order of a few hours on low-end hardware; these few hours achieve very high chances of detection. However, concerning limited time budgets, our tool did not always detect the most number of faults when compared to other techniques.

Regarding scalability, results are rather encouraging with respect to fault detection rates as it takes many more threads to obtain a significant decrease in rates: 10 threads for a 4% decrease on the most complex search space. However, execution time increases exponentially for this same, complex search problem reaching up to 13 hours for 60 threads and 25 GA runs when run on a low-end PC. Since our tool is more of a prototype, there is much room for improvement in running time. Furthermore, execution time could be easily improved on more powerful hardware or by parallelizing the execution of the genetic algorithms [Dori 93]. Optimizing execution time through sophisticated use of distribution and computer clusters was not our focus here. Our results are therefore not representative of what could be optimally obtained in terms of execution time. In future works, ways of assessing and comparing strat-

egies for parallelizing the GA can be explored in aims of improving execution time. Also in future works, we can further assess the performance of our approach and tool on systems with varying structures to establish the particular characteristics of systems our approach is well suited for. For example, the structure of the dining philosophers problem is highly symmetrical and results in a high fault detection rate. Would all such highly symmetrical problems equally result in high detection rates?

Because our approach incorporates the use of a heuristic, namely a genetic algorithm, variance will occur in the results. While no fault detection does not guarantee that no faults exist, one can still feel more confident that such a case is unlikely (i.e., rare in the search space). Even in scenarios where no faults are detected, our tool is still useful. The sequences outputted (recall from Section 6.1 that they are the fittest individuals found) can then be used as future test cases. These would be particularly interesting as they represent scenarios close to faults. Because they are the fittest individuals found by the GA, they are likely very close to fault scenarios. As such, they may later uncover potential problems that can arise due to actual execution times of threads in locks slightly differing from estimates.

9 PUBLICATIONS

The following is a list of journal and conference papers published on the proposed approach.

9.1 Accepted Journal Papers

M. Shousha, L. Briand and Y. Labiche. "UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems", Accepted for publication in *IEEE Transactions on Software Engineering*, 2010.

9.2 Published Papers

M. Shousha, L. C. Briand and Y. Labiche, "A UML/MARTE Model Analysis Method for Detection of Data Races in Concurrent Systems", *MoDELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pp. 47-61, 2009.

M. Shousha, L. C. Briand and Y. Labiche, "A UML/SPT Model Analysis Approach for Concurrent Systems Based on Genetic Algorithms", *MoDELS '08: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, pp. 475-489, 2008.

10 REFERENCES

[Abad 06] M. Abadi, C. Flanagan and S.N. Freund, "Types for Safe Locking: Static Race Detection for Java," *ACM TOPLAS*, vol. 28, no. 2, pp. 207-255, 2006.

[Abde 01] O. Abdellatif-Kaddour, P. Thevenod-Fosse, and H. Waeselynck, "Property-Oriented Testing Based on Simulated Annealing," Available from: http:/ www.laas.fr/~francois SVF/seminaires inputs/02/olfapaper.pdf, 2001.

[Afza 09] W. Afzal, R. Torkar, and R. Feldt, "A Systematic Mapping Study on Non-Functional Search-Based Software Testing," *Information and Software Technology,* vol. 51, no. 6, pp. 957-976, 2009.

[Alba 07] E. Alba, F. Chicano, "Finding Safety Errors with ACO," *Genetic and Evolutionary Computation Conference (GECCO07)*, pp.1066-1073, 2007.

[Ali 09] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation," *IEEE Trans. Soft. Eng,* Vol. 35, No. 6, 2009.

[Back 92] T. Back, "Self-Adaptation in Genetic Algorithms," *Proc. European Conf. Artif. Life*, pp. 263-271, 1992.

[Baco 97] J. Bacon, *Concurrent Systems – Operating Systems, Database and Distributed Systems: An Integrated Approach*. 2nd edition. Addison-Wesley: England, 1997.

[Bake 04] P. Baker et al, "The UML 2.0 Testing Profile". *Proceedings of the '8th Conference on Quality Engineering in Software Technology 2004' (CONQUEST 2004)*, pp. 181-189, 2004

[Bake 05] P. Baker, et al, "Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams", *Proceedings of ACM SIGSOFT Foundations on Software Engineering (FSE)*, pp. 50–59, 2005.

[Bask 05] S. Baskiyar and N. Meghanathan, "A Survey of Contemporary Real-time Operating Systems", *Informatica*, Vol. 29, pp. 233-240, 2005

[Behr 04] G. Behrmann, A. David, and K.G. Larsen, "A Tutorial on Uppaal," Available from: http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, 2004.

[Bell 04] D. Bell, "UML's Sequence Diagram". *The Rational Edge Magazine*, January 2004.

[Bond 00] A. B. Bondi, "Characteristics of Scalability and Their Impact on Performance", *Proc. of 2nd International Workshop on Software and Performance,* pp. 195 – 203, 2000.

[Brat 00] G. Brat, K. Havelund, S. Park, W. Visser, "Java Pathfinder Second Generation of a Java Model Checker," *Proceedings of the Workshop on Advances in Verification*, 2000.

[Burn 93] A. Burns and G. Davies, *Concurrent Programming*, Addison-Wesley:England, 1993.

[Camb 10] M. E. Cambronero, V. Valero and G. Diaz, "Verification of Real-Time Systems Design", *Software Testing, Verification and Reliability*, Vol. 20, Issue 1, pp. 3-37, 2010.

[Chen 06] L. Chen, "The Challenge of Race Conditions in Parallel Programming", Sun Developer Network, Sun Microsystems, 2006. Available from: http://developers.sun.com/solaris/articles/raceconditions.html

[Chug 08] R. Chugh, J.W. Voung, R. Jhala and S. Lerner, "Dataflow Analysis for Concurrent Programs Using Datarace Detection," *ACM PLDI*, pp. 316-326, 2008.

[Clar 00] E. M. Clarke Jr., O. Grumberg and D. Peled, *Model Checking*, MIT Press, 2000.

[Dava 92] S. Dava and L. Sha, "Sources of Unbounded Priority Inversions in Real-Time Systems and a Comparative Study of Possible Solutions", Technical Report NASA-CR-190711, Research Institute for Computing and Information Systems, University of Houston-Clear Lake, 1992.

[Dema 08] S. Demathieu, F. Thomas, C. Andre, S. Gerard, and F. Terrier, "First Experiments Using the UML Profile for MARTE," *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pp. 50-57, 2008.

[Dori 93] M. Dorigo and V. Maniezzo, "Parallel Genetic Algorithms: Introduction and Overview of Current Research", *Parallel Genetic Algorithms: Theory and Applications*, J. Stender, Ed, pp. 5-43, IOS Press, Inc: VA, 1993.

[Down 05] A. Downey, *The Little Book of Semaphores*, 2nd edition, 2005.

[Edel 01] S. Edelkamp, A. Lluch-Lafuente, S. Leue, "Trail-Directed Model Checking," *Electrical Notes in Theoretical Computer Science*, vol. 55, no. 3, pp. 343-356, 2001.

[Faug 07] M. Faugère, T. Bourbeau, R. De Simone and S. Gérard, "MARTE: Also an UML Profile for Modeling AADL Applications", *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS)*, pp. 359 – 364, 2007.

[Flan 00] C. Flanagan and S.N. Freund, "Type-Based Race Detection for Java," *ACM SIGPLAN Notices*, vol 35, no. 5, pp. 219-232, 2000.

[Flan 02] C. Flanagan et al, "Extended Static Checker for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 234-245, 2002.

[Fowl 04] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd edition. Addison-Wesley: Boston, 2004.

[Gagn 08] P. Gagnon, F. Mokhati, M. Badri, "Applying Model Checking to Concurrent UML Models", *Journal of Object Technology*, Vol. 7, No. 1, 2008.

[Gama 10] A. Gamatie, "Generalities on Real-Time Programming", *Designing Embedded Systems with the SIGNAL Programming Language*, Springer: New York, 2010.

[Gode 04] P. Godefroid, and S. Khurshid, "Exploring Very Large State Spaces Using Genetic Algorithms," *Intl. Journal Soft. Tools Tech. Trans*, vol. 6, no. 2, pp. 117-127, Aug. 2004.

[Gold 89] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston: Addison-Wesley, 1989.

[Goma 00] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley, 2000.

[Grad 06] S. Gradara, A. Sontone, and M.L. Villani, "DELFIN+: An Efficient Deadlock Detection Tool for CCS Processes," *Journal of Computer and System Sci*ences, vol 72, no. 8, pp. 1397-1412, 2006.

[Harm 07] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proceedings of the International Conference on Software Engineering (ICSE 2007)*, pp. 342-357, 2007.

[Harm 09] M. Harman, S. A. Mansouri and Y. Zhang, "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends, Techniques and Applications", Technical Report TR-09-03, Department of Computer Science, King's College, London, 2009.

[Haup 98] R.L. Haupt and S.E. Haupt, *Practical Genetic Algorithms*. New York: Wiley-Interscience, 1998.

[Holz 97] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering,* vol. 23, no. 5, pp. 279-295, May 1997.

[Hong 00] T. Hong et al. "Simultaneously Applying Multiple Mutation Operators in Genetic Algorithms". *Journal of Heuristics*, Vol. 6, pp. 439-455, 2000.

[Hou 94] E. Hou et al. "A Genetic Algorithm for Multiprocessor Scheduling". *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5. No. 2, 1994.

[Jaco 01] C. Jacob, *Illustrating Evolutionary Computation with Mathematica*, San Francisco: Morgan Kaufmann, 2001.

[Jens 09] K. Jensen, "Schedulability Analysis of Embedded Applications Modeled Using MARTE", Masters Thesis, Technical University of Denmark, 2009. Available from: http://www.imm.dtu.dk/English/Research/Embedded_Systems_Engineering/Publications.aspx?lg=showcommon&id=248418.

[Jia 05] W. Jia and W. Zhou, *Distributed Network Systems From Concepts to Implementations*, Springer Science + Business Media: New York, 2005.

[Kahl 07] V. Kahlon, Y. Yang, S. Sankaranarayanan and A. Gupta, "Fast and Accurate Static Data-Race Detection for Concurrent Programs," *LNCS*, vol. 4590, pp. 226-239, 2007.

[Kim 05] S. Kim, L. Wildman, and R. Duke, "A UML Approach to the Generation of Test Sequences for JAVA-Based Concurrent Systems," *Proceedings of the Australian Software Engineering Conf*erence, pp. 100-109, 2005.

[Klep 03] A. Kleppe, J. Warmer and W. Bast, *MDA Explained - The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.

[Knap 07] A. Knapp and J. Wuttke, "Model Checking of UML 2.0 Interactions", *Lecture Notes in Computer Science*, Vol. 4364/2007, pp. 42-51, 2007.

[Koud 08] A. Koudri et al., "Using MARTE in a Co-Design Methodology", *Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML profile workshop*, Munich, Germany, 2008.

[Koza 92] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge: MIT Press, 1992.

[Kram 06] J. Kramer and J. Magee, *Concurrency: State Models and Java Programs*, 2nd edition, Wiley, 2006.

[Kupf 01] O. Kupferman and M. Vardi, "Model Checking of Safety Properties", *Formal Methods in System Design*, vol. 19, no. 3, pp. 291-314, 2001.

[Lei 08] B. Lei, L. Wang and X. Li, "UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency," *1st International Conference on Software Testing, Verification and Validation*, pp. 200-209, 2008.

[Leve 95] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[Mahf 95] S. W. Mahfoud and D.E. Goldberg: "Parallel Recombinative Simulated Annealing: A Genetic Algorithm". *Parallel Computing*, Vol. 21, No. 1, pp. 1-28, 1995.

[MART 09] OMG, *UML Profile for Modeling and Analysis of Real-time and Embedded Systems*, version 1.0, http://www.omg.org/spec/MARTE/1.0/PDF, 2009.

[Mrad 08] C. Mradiha, Y. Tanguy, C. Jouvray, F. Terrier, and S. Gerard, "An Execution Framework for MARTE-Based Models," *13th IEEE Conference on the Engineering of Complex Computer Systems (ICECCS)*, pp. 222-227, 2008.

[Oaks 04] S. Oaks and H. Wong, *Java Threads*, 3rd ed. O'Reilly, 2004.

[OMG 09] OMG, *Unified Modeling Language* (UML), version 2.2, http://www.omg.org/spec/UML/2.2/Superstructure/PDF/, 2009.

[Peni 10] P. Penil, J. Medina, H. Posadas and E. Villar, "Generating Heterogeneous Executable Specifications In SystemC from UML/MARTE Models", *Innovations in Systems and Software Engineering*, Vol. 6, No. 1-2, 2010.

[Pera 08] M. A. Peraldi-Frati and Y. Sorel, "From High-Level Modeling of Time in MARTE to Real-Time Scheduling Analysis", *Workshop on Architecting and Construction of Embedded Systems-Model Based (ACES-MB'08)*, 2008.

[PUMA 02] M. Woodside, D. Petriu and D. Amyot, "Performance From Unified Model Analysis (PUMA)", Project website: http://www.sce.carleton.ca/rads/puma/, initiated 2002.

[Rose 99] A. Rosete-Suárez, A. Ochoa-Rodríguez, and M. Sebag, "Automatic Graph Drawing and Stochastic Hill Climbing," *Proc. Genetic Evol. Comp. Conf. (GECCO)*, vol. 2, pp. 1699-1706, 1999.

[Safe 04] M. Safe, J. Carballido, I. Ponzoni and N. Brignole, "On Stopping Criteria for Genetic Algorithms", *Lecture Notes in Computer Science*, Vol. 3171/2004, pp. 405-413, 2004.

[Sava 97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM TOCS*, vol. 15, no.4, pp.391-411, 1997.

[Schn 99] F. Schneider, "UML and Model Checking", Proceedings of the 5th Langley Formal Methods Workshop, 1999.

[Sinc 02] R. Sinclair, *Codenotes for Java: Intermediate and Advanced Language Features*. Ed. Gregory Bill, Random House, 2002.

[Tai 94] K. Tai, "Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs". *Proceedings of the International Conference on Parallel Processing (ICPP)*. pp. 69-72, 1994.

[Tane 08] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed., Prentice Hall, 2008.

[Trac 99] N. Tracey, J. Clark, J. McDermid, and K. Mander, "Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification," *Proceedings of the 17th International Systems Safety Conference*, pp. 128-137, 1999.

[UML 2] Object Management Group: "Unified Modeling Language: Superstructure". Version 2.2, http://www.omg.org/spec/UML/2.2/Superstructure/PDF/, 2009.

[Vida 09] J.Vidal et al., "A Co-Design Approach for Embedded System Modeling and Code Generation with UML and MARTE", *Design, Automation and Test in Europe (DATE 09)*, pp. 226-231, 2009.

[Yang 06] H. Yang, "Deadlock," *Java Tutorials – Herong's Tutorial Notes,* version 4.10, *http://www.herongyang.com/java/index.html*, 2006.

11 APPENDICES

11.1 Appendix A

We present here a general proof of concept implementation showing how to automate extraction of relevant information from a MARTE annotated sequence diagram.

A number of commercial tools exist that implement the MARTE profile. Papyrus is one such tool. It is available as an Eclipse plug-in. Once a sequence model is built using Papyrus and annotated with the appropriate MARTE stereotypes, it can be exported to an XMI file. Using the XMI file, it is then a matter of using Eclipse's EMF based implementation of UML to extract the required information.

For example, for the airline reservation example in Section 3, seats are stereotyped as `<<SwMutualExclusionResource>>`. To extract this information, the XMI file is generated for the sequence diagram. Then, the following loop can be used to extract all `<<SwMutualExclusionResource>>`:

```
List<Element> elements =SequenceDiagramModel.getOwnedElements() ;
for(int i=0;i<elements.size();i++){
  ListofStereotypes<Stereotype> stereotypes =
  elements.get(i).getAppliedStereotypes("SwMutualExclusionResource");
  //further process the stereotypes, finding out capacities, wait queue poli-
cies, etc.
}
```

The names associated with the stereotype can then be output to a file, set up in the format required by the GA backend of our approach.

11.2 Appendix B

The following demonstrates how the fitness function of equation (1) satisfies the properties required for deadlock detection.

Property 1 holds: When lock queues are empty, threadsWaiting = 0 and f(c) = #LockExecs which is in the range `[0 - lockCapacities]`. When one thread is waiting, `threadsWaiting = 1`, and `f(c) = #LockExecs + 1` which is in range `[1 - lockCapacities]`. When at least two threads are waiting (second case in the definition of our fitness function), `threadsWaiting ≥ 2` and `f(c) = #LockExecs + threadsWaiting + lockCapacities`. The minimum value of `f(c)` is then `#LockExecs + 2 + lockCapacities`, with `#LockExecs > 0`. This is always greater than `lockCapacities + 1`. By using `lockCapacities` in the second case of the fitness function, we ensure that situations where lock queues hold up to one thread always have lower fitness than ones where at least two threads are held in lock queues, thus satisfying property 1.

Property 2 holds: When the number of threads executing in locks increases, `#LockExecs` increases, and therefore `f(c)` increases.

Property 3 holds: When the number of waiting threads increases, threadsWaiting increases, and therefore f(c) increases.

11.3 Appendix C

The following demonstrates how the fitness function of equation (2) satisfies the properties required for starvation detection.

Property 1 holds: When the target thread is the only thread waiting in the wait queue of the target lock during any time unit, f(c) = i+(i-1)+…+1+0, and fitness can only increase as additional threads wait in the queue. When the target lock's wait queue is empty during a time unit, f(c) = (i-1)+(i-2)+…+1+0 (Situation a.). When the wait queue is not empty during a time unit of the testing interval, but the target thread is not in the wait queue during that time unit, f(c) = (i-1)+(i-2)+…+1+0 (Situation b.). Both situations have lower fitness values than when the target thread is waiting in the wait queue, thus fulfilling Property 1.

Property 2 holds: At any given time unit in the time interval, when the target thread accesses the target lock, the accumulated fitness value remains the same; calculations thereafter continue as before in subsequent time units.

11.4 Appendix D

The following demonstrates how the fitness function of equation (3) satisfies the properties required for data race detection.

Property 1 holds: The smaller the difference of resource access times between writer threads and any other thread (reader or writer), the closer a data race situation: A data race situation arises when f(c)=0. As $|W_1 - N(W_1)|$ decreases, it will eventually tend to zero.

Property 2 holds: Scenarios where data races are possible are always rewarded over situations where no data races are possible: No data races are possible when zero or one thread accesses the shared resource. A.) When zero threads access the shared resource, $\#W_1 = 0$ and f(c) = endTime. B.) When only one reader thread accesses the shared resource, $\#W_1 = 0$ and f(c) = endTime. C.) When only one writer thread accesses the shared resource, $\#W_1 = 1$, #T=1 and f(c) = endTime. Data races are possible when at least two threads are executing and at least one is a writer. Here, $\#W_1 = 1$, #T>=2 and $f(c) = |W_1 - N(W_1)|$, each term being in the range [startTime-endTime]. The fitness can thus be |startTime – endTime| or zero. In either case, it is less than the fitnesses of scenarios A, B and C, thus fulfilling property 2.