A UML/MARTE Model Analysis Method for Detection of Data Races in Concurrent Systems

Marwa Shousha[1], Lionel Briand[2], and Yvan Labiche[1]

[1] Carleton University, Software Quality Engineering Lab, 1125 Colonel By Drive Ottawa, ON K1S 5B6, Canada
{mshousha, labiche}@sce.carleton.ca

[2] Simula Research Laboratory & University of Oslo, P.O. Box 134, Lysaker, Norway
briand@simula.no

Abstract. The earlier concurrency problems are identified, the less costly they are to fix. As larger, more complex concurrent systems are developed, early detection of problems is made increasingly difficult. Meant to be used in the context of Model Driven Development, we have developed a general approach, based on the analysis of design models expressed in the Unified Modeling Language (UML) that uses specifically designed genetic algorithms to detect concurrency problems. All relevant concurrency information is extracted from systems' UML models that comply with the UML Modeling and Analysis of Real-Time and Embedded Systems profile. Our approach was previously shown to work for both deadlocks and starvation. The current paper addresses data race detection, further illustrating how our approach can be tailored to other concurrency issues. Our main motivations are (1) to devise practical solutions that are applicable in the context of UML design of concurrent systems without requiring additional modeling and (2) to achieve scalable automation. Results on a case study inspired from the Therac-25 radiation machine show that our approach is effective in the detection of data races.

Keywords: MDD, data races, model analysis, concurrent systems, UML, MARTE, genetic algorithms.

1 Introduction

Concurrency problems should be identified early in the design process. This is made increasingly difficult as larger and more complex concurrent systems are being developed. With the recent trend towards Model Driven Development (MDD) [15], 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. However, the analysis of concurrency properties should not require additional modeling or a high learning curve on the part of the designers, or should at least minimize it. When the UML notation is not enough to completely model 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 [19] that addresses domain specific aspects of real-time, concurrent system modeling. Our aim is to develop a general, automated approach that can be

tailored to several types of concurrency errors (such as deadlocks, starvation, data races and data flow problems), and that can be easily integrated into a Model Driven Architecture (MDA) approach, the UML-based MDD standard by the OMG [15]. Our approach relies on a genetic algorithm (GA) that is tailored to different types of concurrency errors.

In previous works, we have tailored a GA for the detection of deadlocks [23] and starvation [13]. This paper is a continuation of these works, where we adapt the approach to the detection of data races. It differs from its predecessors in three areas: 1. A different UML profile is used. Instead of the SPT profile in [23], which was the standard at the time, we use the MARTE profile; 2. We have different GA components. a.) A different chromosome representation (our previous structure of genes [23, 13] contained lock information, which is not needed in the current paper). Since the chromosomes are different, the genetic operators of mutation and crossover are also different (though the principles remain the same, the realizations are different). b.) We use a different fitness function. We used [23, 13] fitness functions specifically designed to detect deadlocks and starvation, respectively. Here, we provide a fitness function geared towards data races; 3. We improve performance comparison. In both previous works, we measured performance against random search only. Here, we also compare our approach with a hill climbing search. Performance of each type of error naturally entails different case studies, each geared towards the respective problem being examined.

We next provide an overview about data races, highlighting the information needed as input to our approach, and discuss the principles of genetic algorithms. Sections 3 and 4 provide details of our tailored GA, and tool support. Section 5 describes a case study inspired from the Therac-25 radiation machine, along with results comparing random, hill climbing and GA searches. Related work is presented in Section 6 and we conclude in Section 7.

2 Background

We next present some background information. In particular, we describe data races and aspects of relevance in the MARTE profile.

2.1 Data Races

Concurrency introduces the need for communication between executing *threads* [7]. Threads may communicate via a *shared memory location* during various *access times* for a defined *execution time*. These access and execution times may be specified as ranges, probability distributions, or definite values, although ranges are probably more common due to uncertainty at design time.

The term race condition has been generally used to describe situations where unsynchronized concurrent accesses result in unpredictable program states and behavior [1]. Data races, a specific type of race conditions, are quite common in concurrent systems [1]. These types of faults are due to unsynchronized access to a same memory location. Threads may access a shared location as either *reader threads* or *writer threads*. Problems then arise due to the order of execution of events [1].

While many times unsynchronized access to shared resources is 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 for writing, 3. Thread access to the memory location is unsynchronized. When these three conditions are met, a writer and reader thread may execute concurrently within the shared memory, resulting in inconsistent data.

To proceed with our approach, we must first map the data race concepts, in particular those appearing in italics in this section, to UML and MARTE concepts, as they form the inputs of the GA.

2.2 MARTE Profile to Data Race Mapping

In UML, active objects have their own thread of control, and can be regarded as concurrent threads [12]. Only extensions of the UML standard, such as the MARTE profile [19], provide mechanisms to model detailed information pertaining to concurrency. The MARTE profile is a replacement of the SPT profile [24]. MARTE is geared towards both the real-time and embedded system domains. The profile is roughly divided into three major sub-divisions: 1. MARTE foundation (containing the basis for real-time and embedded system modeling. It defines time concepts and use of concurrent resources), 2. MARTE design model (specializes the foundation, allowing modeling of various features of real-time and embedded systems) 3. MARTE analysis model (allows the annotation of models for system analysis purposes). Much like SPT, the MARTE profile is modular in structure, allowing users to choose the appropriate subsets needed for their applications. We next describe the aspects of the profile that are relevant to our work.

The Software Resource Modeling (SRM) sub-profile presents mechanisms for designing multitasking applications. SRM is 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, 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 the SW_Interaction package, shared resources are identified as `<<SharedDataComResource>>`.

The Generic Quantitative Analysis Modeling (GQAM) sub-profile 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. In Time Modeling, timed constraints can be specified on the occurrence of an event, on the duration of an execution, or on the temporal distance between two events. These are stereotyped with `<<TimedConstraint>>`.

The High-Level Application Modeling (HLAM) sub-profile 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`.

This overview of MARTE illustrates that the input to our approach (the concepts presented in italics in Section 2.1) can be retrieved from a UML/MARTE design model. The mappings between those concepts and the profile are summarized in Table 1. It is then clear that the information used by our approach can be automatically retrieved from UML/MARTE models, in particular from sequence diagrams where those stereotypes and tags are used.

Table 1 Concept to MARTE Mapping

Concept	MARTE Stereotype/Tag	MARTE sub-profile
Thread	`<<SwConcurrentResource>>`	SRM::SW_Concurrency
Unprotected resource	`<<SharedDataComResource>>`	SRM::SW_Interaction
Reader	`<<RtService>>/concPolicy = reader`	HLAM
Writer	`<<RtService>>/concPolicy = writer`	HLAM
Thread exec. time in res.	`<<gaStep>>/execTime`	GQAM:: GQAM_Workload
Thread access time of res.	`<<gaStep>>/interOccTime|` `<<gaStep>>/execTime`	GQAM:: GQAM_Workload
Time constraints	`<<TimedConstraint>>`	Time

2.3 Genetic Algorithms

GAs are a means of solving optimization problems. They are based on concepts adopted from genetic and evolutionary theories [10]. A GA first randomly creates an initial population of solutions, called chromosomes, then selects a number of these solutions and performs various genetic operators (mutation and crossover) to create new solutions. The measure of goodness of each solution, called fitness, is compared with other solutions, with only the fittest solutions retained. The process of selection, crossover and mutation, fitness comparison and replacement continues until the stopping criterion, such as a maximum number of generations [10], is reached.

3 Tailored Genetic Algorithm

To use a GA to detect the presence of data races, we must first tailor it by defining the chromosome representation, mutation and crossover operators as well as the fitness function, which we discuss next.

3.1 Chromosome Representation

A chromosome is composed of *genes* and models a solution to the optimization problem. 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. The access times must reflect schedulable scenarios. In other words, we need to ensure that all execution sequences represented by chromosomes are schedulable. This entails meeting system specifications of periods, minimum arrival times, and so on. Thus, we need to encode threads (`<<SwConcurrentResource>>`), resources (`<<SharedDataComResource>>`), read and write operations (`<<RtService>>/ concPolicy = read`, `<<RtService>>/concPolicy = write`) and access times (`<<gaStep>> / interOccTime` or `<<gaStep>> / execTime`), which are available in the input model (Table 1).

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. A tuple represents the execution of a thread when accessing the resource. Tuples are defined for a user specified time interval during which the designer wants to study the system's behavior. A heuristic for determining an appropriate time interval is given in Section 3.4. A special value of -1 is used to depict access times that lie outside this interval: (T, -1) represents a thread access that does not occur. 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 considered as a property of the thread itself, along with the range of valid access times.

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 the resource in the given time interval. A thread can appear more than once in the chromosome if it accesses the resource multiple times.

Three 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 increasing access times. 2.) Thread access times of the resource must fall within the specified time interval or are set to -1. 3.) Consecutive genes for the same thread must have access time differences equal to at least the minimum and at most the maximum access time range of the associated thread, if start and end times are defined as ranges.

Consider, for example, the set of three threads accessing a resource named MEOS: T1 (access range [1 325] time units, repeats every 399 time units), T2 (access range [325 398], repeats 400) and T3 (access range [327 392], repeats 400). In a time interval of [0 350] time units, the chromosome length would be three since each of the threads can access the resource at most once during this time interval. The following is then a valid chromosome: (T1, 324) (T2, 340) (T3, -1) where T1 accesses the resource at time unit 324, T2's access is at time 340 and T3 does not access the resource before time 350.

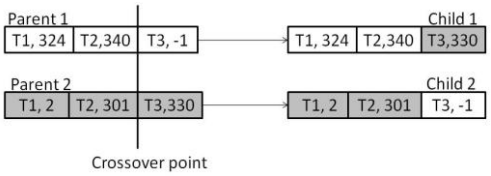
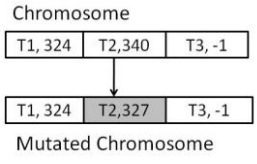


Figure 1. a.) Crossover example



b.) Mutation example

3.2 Crossover Operator

Crossover is the means by which desirable traits are passed from parent chromosomes to their offspring [10]. We use a one-point, sexual crossover operator: two parents are randomly split at the same location into two parts which are alternated to produce two children. For example in Figure 1a, the two parents on the left produce the offspring on the right. If, after crossover, any two consecutive genes of the same thread no longer meet their access time requirements (constraint 3 is violated), the second gene's access time is randomly changed such that constraint 3 is met. This is repeated until all occurrences of this situation satisfy constraint 3.

3.3 Mutation Operator

Mutation introduces new genetic information, hence further exploring the search space, while aiding the GA in avoiding getting caught in local optima [10]. 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. 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 result in data races. When a gene is chosen for mutation, a new timing value is randomly chosen 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 access time requirements, the affected genes are altered such that the requirements are met. For the example of Figure 1b with access times [1 325], [325 398] and [327 392] in a time interval of [0 350], assume Parent 1's second gene is chosen for mutation. A new value (say, 327) is chosen from its access time range [325 398], as shown in Figure 1b.

3.4 Fitness Function

The fitness function determines the merit of a chromosome. Recall that data races occur when at least two threads share a resource and at least one is a writer thread. For the fitness function to be effective, the time interval over which it is defined must be adequate: it should be long enough for data races to occur, but not too long to not hinder the performance of the search algorithm. This varies from system to system and depends on the amount of time resources available. We propose a heuristic for determining the time interval based on the longest thread execution time in the resource (*lt*) and the maximum resource access time of all threads (*lr*). Our heuristic is to guarantee, using these two variables, that all threads can completely access the resource at least twice. Therefore, the time interval equals: [0 *(lt+lr)*2*]. This is a minimum interval, as having threads access the resource just once may not be enough to uncover a data race. Designers can opt for a larger interval.

We define the following fitness function:

$$f(c) = \min_{i=startTime\,to\,endTime} \begin{cases} |W_i - N(W_i)| & if\ \#W_i \geq 1 \\ endTime & if\ \#W_i = 0 \end{cases} \quad (1)$$

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 that access the resource during the time unit *i*. *N*, W_i and $\#W_i$ are obtained after scheduling.

The fitness function of equation (1) 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*).

Let us consider the scheduling of the mutated chromosome in Figure 1b, where T1 is a writer thread and all other threads are readers. The time interval is assumed to be [0 350]. Using equation (1) for Figure 2, we examine the time units for resource R1:

At 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.

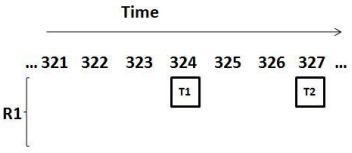


Figure 2. Fitness function example

4 Tool and GA Parameters

We have built a prototype tool, Concurrency Fault Detector (CFD), for detection of data races using our approach. CFD is an automated system that identifies concurrency errors in any concurrent application modeled with the UML/MARTE notation. Currently, it can help identify deadlock, starvation [23, 13] and data race errors. Work is in progress for the detection of other types of concurrency errors. CFD involves a sequence of steps. Users first input three categories of 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 error targeted: data race, deadlock or starvation. In the latter case, the target thread and target lock are also inputted. CFD then extracts the required information from the inputted UML/MARTE model (mainly from its sequence diagrams) and feeds it to the appropriate GA.

CFD is decomposed into two modular portions: a scheduler and a genetic algorithm. This modularity ensures that modifications can seamlessly be adapted to meet a wider set of requirements. Modifications to the scheduling strategy would only

require altering the scheduler. Hence, the scheduler does not affect the applicability of our approach as it is merely a black box that aides in the calculation of the fitness function. It emulates single processor execution as it tracks all thread executions.

In the GA for data races, if a data race is detected, CFD outputs the sequence resulting in the data race as well as the time unit at which the data race occurs and a depiction of the threads executing within the resource at that time. If no such sequence is found, CFD terminates after 1000 generations, outputting the execution sequence with the lowest fitness value (since it is a minimization function). This does not guarantee that no data races exist. However, one can still feel more confident that such a case is unlikely (i.e., rare in the search space).

Since collecting input data is easy to automate 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, data race, or starvation. This is the main reason why we developed a strategy that only reports one concurrency 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.

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, population size, mutation and crossover rates and selection operator. We use a steady state GA, with a replacement percentage of 100%. The population size we apply is 200. This is higher than the size suggested in [10], but 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 [18]. 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 [16] and [10], respectively.

All parameter values are based on findings reported in the literature, except population size, which was fine tuned after some experimentation. 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 error. 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.

We have used CFD on the case study presented next to assess our approach.

5 Case Study: Therac-25 (Therac)

The case study we use was inspired from the Therac-25 machine. The infamous Therac-25 was a computer controlled radiation therapy machine that was responsible for overdosing six patients. 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 [4]. The original design of the Therac-25 system—or simply Therac—has been altered, whereby access times of threads to resources have been increased to provide a larger search space, thus reflecting more realistic situations.

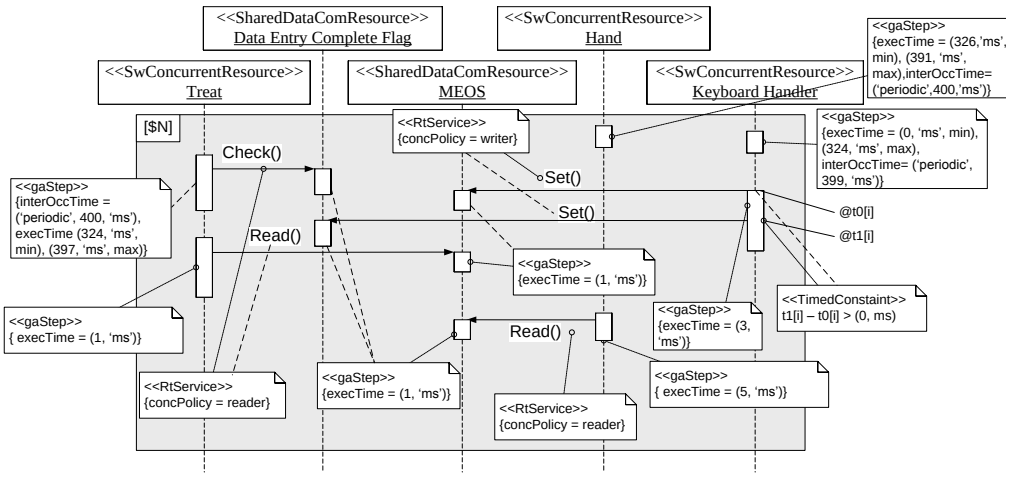


Figure 3. Therac sequence diagram

5.1 Therac Extended with MARTE

Figure 3 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 inputted. 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.

5.2 Analysis of Search Space

To detect a data race, we need to search the set of possible (i.e. ones that adhere to the input requirements) events received by shared resources (hence referred to as sequences) for at least one that yields a data race. This set of possible sequences is called the search space. The search space differs for the two resources. For MEOS, it is based on the access time intervals of the `Treat`, `Hand` and `KeyboardHandler` threads as well as the timing interval. For a timing interval of 802 time units (based on our heuristic, Section 3.4), the search space is approximately $9.8 * 10^{12}$. Of these, $4.1 * 10^{8}$ yield a data race. For DECF, the search space is $1.7 * 10^{9}$, with 360,750 resulting in a data race.

To further enhance our case study, we altered the access times 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 search space is approximately $4.7 * 10^{9}$, with 4510 yielding a data race. For the altered DECF, or simply DECF2, with a timing interval of 800, the search space is $1.0 * 10^{7}$, with 99 resulting in a data race. We can then better assess how the performance of search techniques is affected by the difficulty of the search.

A search space is further characterized by its complexity. Points in the search space that result in data races 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 data race. The more local optima in the search space, the more complex it is. In both MEOS resources, the search space is complex, with many local optima. For DECF and DECF2, the search space has a few local optima, thus simplifying the search.

5.3 Case Study Design

We begin by briefly describing the techniques used for data race detection, then relating how the case study was set up to ensure that all techniques are comparable.

Description

We use three different techniques to detect 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 [25].

In random generation, a point in the search space (representing a sequence of resource accesses by various threads) is randomly chosen and checked for a data race. Running a random search involves running a pre-determined, usually large number of points in the search space.

For hill climbing, one random point is generated, then neighboring points are examined, with the one better than the current point replacing it. This continues until a point is reached that has no better neighbor. For Therac, a neighbor is one that differs by just one access value from the current point. 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 value from the current point.

Fairness of Comparisons

Because each of the three techniques proceeds differently, we analyzed the number of sequences generated by the GA and generated the same number for other techniques to ensure a fair comparison. As GAs are a heuristic optimization technique, variance occurs in the results they produce. To account for the variability in results, we ran our case study 50 times on an Intel Core 2 2.0 GHz processor. Random generation and hill climbing were also run 50 times, with each run generating the same number of sequences as the number created and evaluated by a GA run. In the original design, with a timing interval of 802, a GA run generates on average 5184 sequences for MEOS and 4640 for DECF. In the altered design, the GA generates 6786 sequences on average for MEOS2 in a timing interval of 2500, and it generates 11681 sequences on average for DECF2 in a timing interval of 800. In all cases, for random, hill climbing and the GA, when a data race is detected, execution stops and a new run of the 50 is executed.

5.4 Results

Results of the detection rate of data races are presented in Table 2. All three techniques are capable of detecting data races in both MEOS and DECF, but with very different probabilities. Hill climbing does not fare very well in the former case. It appears to be oftentimes caught in local optima: 96% of the time, it is unable to detect a data race. This empirically suggests that the search space for MEOS is complex. We observe that our GA does better: 34% detection rate for MEOS. This confirms that where the search space is large and complex, GAs are known to yield much better results than the two other techniques [17]. 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.

Table 2. Comparison of Performance

		MEOS	**DECF**	**MEOS2**	**DECF2**
	Search Space Size	$9.8 * 10^{12}$	$1.7 * 10^{9}$	$4.7 * 10^{9}$	$1.0 * 10^{7}$
	% of Data Race Sequences	0.004	0.02	$9.5 * 10^{-5}$	$9.5 * 10^{-4}$
Random	#Detections/#Runs	3/50	10/50	0/50	2/50
	Total Runtime (min:sec:ms)	01:07::281	00:44:324	04:29:819	01:52:818
	Detection rate	6%	20%	0%	4%
GA	#Detections/#Runs	17/50	49/50	4/50	43/50
	Total Runtime (min:sec:ms)	01:34:255	01:01:80	05:40:749	02:46:760
	Detection rate	34%	98%	8%	86%
Hill Climbing	# Detections /#Runs	2/50	50/50	1/50	50/50
	Total Runtime (min:sec:ms)	00:53:980	0:12:862	04:38:062	00:14:087
	Detection rate	4%	100%	2%	100%

On the other hand, in the case of the simpler DECF search space, hill climbing does exceedingly well, detecting data races in all runs. Here too, random search performs relatively well, owing to the higher percentage of sequences leading to a

data race (0.02%). A GA is therefore of no benefit in this case, although it too performs well.

For MEOS2 and DECF2, the search spaces are of similar complexity as MEOS and DECF, respectively, but with smaller sizes and lower percentage of sequences leading to a data race. For MEOS2, the search space is large and complex, with $9.5 * 10^{-5}$% of sequences leading to a data race. Both random and hill climbing perform very poorly. The GA, while performing worse than for MEOS, still manages to detect data races four times as much as hill climbing. In DECF2, random performs worse because of the lower percentage of data race sequences ($9.5 * 10^{-4}$%). The GA too performs worse than for DECF. Hill climbing remains unaffected by the size of the search space and changes in data race probabilities, probably due to the simplicity of the search space (few local optima).

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. As expected, the execution time of the GA is longer than the other techniques, yet in complex search spaces (MEOS) the difference with hill climbing is of the order of 20-30%. 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 constraints, to obtain the highest possible overall probability of detecting data races. Using the most complex case (MEOS2) as an example, with an 8% probability of data race detection, 50 GA runs results in a probability of less than 2% (0.92^{50}) not to detect a data race in at least one run, with a bit more than five minutes of execution time. Such execution times can of course be brought down significantly with faster hardware and parallel computing. Even when in practice 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.

6 Related Work

In the context of detecting data races in concurrent systems, a number of works exist. Some [8, 9, 11, 14], do so using the code of the system under test. Kahlon et al. [14] begin 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. [20] also use a form of static analysis. In their work, they use program code to develop a data flow analysis for the system under test. They combine this with an independent race detection engine to return a version of the 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 are very costly to fix. Furthermore, data races due to dynamically allocated shared resources might go undetected. Other works, such as

Savage et al., 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 [21]. 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.

Model checking has been used to detect data races in concurrent systems, such as in the Java Path Finder [22]. The aim here is the same as our aim: to detect problems arising from system models. However, what differs is the context: our approach is meant to be used in the context of MDD, specifically MDA. As such, we rely on UML extended with profiles, rather than temporal logic specifications.

Of particular interest is the work by Lei, Wang and Li [2]. 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. They also serve to ensure that both code and design are consistent. Lei, Wang and Li present results for two case studies. In the online store system, they discover five instances of data races. In the elevator system, they discover none. The authors note that they use random testing for comparison, but do not report results for it. They also do not provide execution times for their approach [2].

With the current trend towards MDD [15], models are regarded as the essence of system development. While their development may be time consuming, they can be used to partially automate other activities. 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 MARTE profile, thereby reusing existing design models instead of developing specific models (as in the work by Lei et al.) or waiting until the system is implemented to execute its code. As all information required by our approach can be incorporated in the UML model of a system, this eliminates the need for additional modeling activities (e.g., using temporal logic). Use of the standard MARTE extension also eliminates the need for haphazard additions (e.g., extensions for modeling data sharing by Lei et al.). Furthermore, standard profiles tend to be implemented within commercial tools, once the profile has been approved. While the sequence diagrams required by CFD may not be as detailed as required when the system is initially designed, adding information to these pre-existing diagrams for testing purposes is probably easier than working with a different model, such as in the case of Java Path Finder. In essence, our approach can be thought of as a scalable, guided random search to be used in the context of MDA.

In the context of MDD, a number of works utilize MARTE's predecessor: the SPT profile. Such works, (e.g., [3]), mostly focus on performance analysis rather than the analysis of model properties. Other works such as [6] and [5] use the MARTE profile. However, in [6], the profile is used to create an approach for real-time embedded

system modeling along with transformations to execute those models. In [5], the authors aim at probing the capabilities of MARTE by applying it to a case study.

7 Conclusions

Concurrency abounds in many software systems, where threads typically access many shared resources. If not handled properly, such accesses can lead to concurrency errors, which may lead to serious system failures. The earlier any such problem is detected during the design process, the better. In this paper, we describe an approach, based on a tailored genetic algorithm (GA) search, for detecting one type of concurrency error: data races. The approach is based on the analysis of design representations in UML completed with the MARTE profile. Since our goal is to provide an automated approach that can be applied in the context of model-driven, UML-based development, the choice of UML/MARTE was natural as it is the de facto standard for the object-oriented modeling of concurrent, real-time applications. This is also practical as it reduces the need for complex tooling and training, while reusing models already required for UML-based development. In other words, instead of adopting more formal representations (e.g., temporal logic), we reuse a standard UML profile but rely on carefully designed, search-based heuristics for the detection of data races. Our findings suggest that the GA has much higher chances than simpler alternatives (e.g., hill climbing) to detect data races when the search space is large and complex and few sequences lead to a data race, a situation we expect to be increasingly common in the design of industrial concurrent systems. Results further show that even in our most complex case the probability of not detecting a data race is less than 2% using a bit more than five minutes of execution time on a standard PC. Our current work focuses on providing a general framework that can be easily adapted to different types of concurrency problems, e.g., deadlocks and starvation.

8 References

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

[2] Lei, B., Wang, L., Li, X., "UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency", *ICST*, 200-209, (2008).

[3] Petriu, D. C.: “Performance analysis with the SPT profile”. *Model-Driven Eng. Dist. Embed. Sys.*, 205-224. (2005).

[4] Leveson, N., *Safeware: System Safety and Computers*, Addison-Wesley, (1995).

[5] Demathieu, S., Thomas, F., Andre, C. Gerard, S. and Terrier, F.: “First experiments using the UML profile for MARTE”. *IEEE ISORC*. 50-57. (2008).

[6] Mradiha, C., Tanguy, Y., Jouvray, C., Terrier, F. and Gerard, S.: “An execution framework for MARTE-based models”. *IEEE ICECCS*. 222-227 (2008).

[7] Downey A.B., *The Little Book of Semaphores*, Green Tea Press, 2nd Ed. (2005).

[8] Flanagan, C. and Freund, S. N. "Type-Based Race Detection for Java", *ACM PLDI*, 219–232, (2000).

[9] Flanagan, C., Rustan, K., Leino, M., Lillibridge M., Nelson, G., Saxe, J. B., and Stata, R. "Extended Static Checker for Java", *ACM PLDI*. 234–245, (2002).

[10] Haupt, R. L. and Haupt, S. E.: *Practical genetic algorithms*. Wiley, (1998).

[11] Abadi, M., Flanagan, C., and Freund, S. N. "Types for Safe Locking: Static Race Detection for Java", *ACM TOPLAS*, Vol. 28, No.2, 207–255, (2006).

[12] OMG: *Unified Modeling Language* (UML). Version 2.1.2. (2007).

[13] Shousha, M., Briand, L.C., Labiche, Y., "A UML/MARTE Model Analysis Methodology for Detection of Starvation and Deadlocks in Concurrent Systems," Carleton University, Technical Report SCE-09-01, squall.sce.carleton.ca, 2009.

[14] Kahlon, V., Yang, Y., Sankaranarayanan, S. and Gupta, A. "Fast and Accurate Static Data-Race Detection for Concurrent Programs", *CAV,* 226-239, (2007).

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

[16] Back, Thomas: "Self-adaptation in genetic algorithms". *Proc. European Conf. Artif. Life*, 263-271 (1992).

[17] Mahfoud, S. W. and Goldberg, D. E.: "Parallel recombinative simulated annealing: a genetic algorithm". *Parallel Comp*. Vol 21, No. 1, 1-28. (1995).

[18] Koza, John R.: *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge (1992).

[19] OMG: UML Profile for Modeling and Analysis of Real-time and Embedded Systems. 1.0 Beta (2008), www.omg.org/cgi-bin/apps/doc?ptc/08-06-08.pdf

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

[21] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs ", *ACM Trans. Comp. Sys.*, Vol. 15, No. 4, 391–411, (1997).

[22] Brat G., Havelund, K., Park, S., and Visser, W. "Java Pathfinder Second Generation of a Java Model Checker", *Proc. Workshop Adv. Verif.*, (2000).

[23] Shousha, M., Briand, L., and Labiche, Y.: "A UML/SPT model analysis methodology for concurrent systems based on genetic algorithms". *Proc. ACM/IEEE MODELS*, 475-489, (2008).

[24] OMG: UML Profile for Schedulability, Performance and Time Specification. Adopted Specification (2005), http://www.omg.org/docs/formal/05-01-02.pdf

[25] Ali, S., Briand, L. C., Hemmati, H. and Panesar-Walawege, R. K.: "A Systematic Review of the Application and Empirical Investigation of Search-based Test-Case Generation", Simula Research Laboratory, Technical Report Simula.SE.293 (2009).