A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems

Marwa SHOUSHA[1], Lionel BRIAND[2], and Yvan LABICHE[1]

[1] *Software Quality Engineering Laboratory*
Dept. of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada
{mshousha, labiche}@sce.carleton.ca

[2] *Simula Research Laboratory and University of Oslo*
P.O. Box 134, Lysaker
1325 Norway
briand@simula.no

***Abstract*—Concurrency problems, such as starvation and deadlocks, should be identified early in the design process. As larger, more complex concurrent systems are being developed, this is made increasingly difficult. We propose here a general approach, based on the analysis of specialized design models expressed in the Unified Modeling Language (UML) that uses a specifically designed genetic algorithm to detect concurrency problems. Though the current paper addresses deadlocks and starvation, we will show how the approach can be easily tailored to other concurrency issues. Our main motivations are (1) to devise 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 genetic algorithm is used to search for execution sequences exhibiting deadlock or starvation problems. 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.**

***Keywords*—Search based software engineering, MDD, deadlock, starvation, model analysis, concurrent systems, UML, MARTE, genetic algorithms**

I. Introduction

Concurrency problems, such as deadlocks and starvation, 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) [27], the choice of using Unified Modeling Language (UML) models and their extensions as a

Manuscript received September 1, 2008.

M. Shousha is a PhD candidate in the Systems and Computer Engineering Department, Carleton University, 1125 Colonel By Drive Ottawa, ON K1S 5B6, Canada (e-mail: mshousha@sce.carleton.ca).

L. Briand was with Carleton University, Ottawa, ON K1S 5B6, Canada. He is now with Simula Research Laboratory & University of Oslo, P.O. Box 134, Lysaker, Norway (e-mail: briand@simula.no).

Y. Labiche is with the Systems and Computer Engineering Department, Carleton University, 1125 Colonel By Drive Ottawa, ON K1S 5B6, Canada (e-mail: labiche@sce.carleton.ca).

Digital Object Indentifier 10.1109/TSE.2010.107

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. We hence propose an approach for the detection of concurrency problems that is based on design models expressed in UML [35]. 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 here is the standardization of the Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile [33] that addresses domain specific aspects of real-time, concurrent system modeling. Our aim is to develop a scalable, automated method that can be easily tailored to all 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 [27]. This is achieved through three steps: 1. Demonstrating the feasibility of extracting all relevant concurrency information from UML/MARTE design diagrams. 2. Showing the effectiveness of the proposed search based technique in detecting concurrency faults and 3. Demonstrating scalability in terms of fault detection as the size of the problem grows.

The approach we describe requires a UML/MARTE annotated front end model of the system under test. It relies on a backend based on a number of tailored Genetic Algorithms (GAs), each directed at finding a particular concurrency fault. Each tailored GA uses information available in the UML/MARTE design model of a software system to search for conditions under which threads can lead to either deadlocks or starvations.

We begin by automatically collecting all information relevant to a deadlock (e.g., thread/lock interaction) or starvation (e.g., thread/lock interaction, thread priorities) from the system's UML design model extended with the MARTE profile. This step can be easily automated using one of

various existing UML tools and the underlying UML/MARTE metamodel. The extracted information is then fed to the GA with a fitness function tailored to deadlock detection, or with a fitness function tailored to starvation detection, depending on the type of fault the test engineer is interested in discovering. Since deadlock and starvation problems 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 either fault.

The approach we adopt is meant to be general and can be adapted (as illustrated in this paper on two types of faults) to a variety of concurrency faults by tailoring the fitness function of our specifically designed GA in order to address problems such as deadlock, starvation, data races and data flow problems. The current paper addresses both starvation and deadlocks, which have a lot in common. We start here by fully addressing starvation, then address only the differences (namely inputs and fitness function) for deadlock. Future work will address other types of problems. 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, as later shown in the reported case studies in Section VIII.

We next present an overview of related work, followed by an overview about deadlock and starvation, highlighting the information needed as input (Section III). Section IV introduces the MARTE profile and discusses the mapping between the profile and various deadlock and starvation concepts, showing that the information needed as input (Section III) can be extracted from UML/MARTE design models. This section achieves the first step of our aim. Sections V and VI describe our method in detail, for starvation and deadlock detection, respectively. Section VII presents our tool support and Section VIII discusses six case studies along with their results (including a study of scalability), thus demonstrating the last two steps of our aim. We conclude

in Section IX.

II. Related Work

Our approach spans several fields of research, namely verification of concurrent systems (in terms of deadlocks, starvation, data races and data flow), 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 aforementioned order: model checking (Section II.A), search-based, non-functional testing (Section II.B), and uses of MARTE (Section II.C).

A. 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, concurrency, or other important properties [36]. 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 [36]. 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 [27] for modeling real-time, concurrency information. An aspect that results from these choices and that may be considered a drawback is that we have to rely on heuristic search algorithms to detect concurrency faults. 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 [19], 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 reuse*: Existing design models can be reused 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 [9], aim at detecting data races, while others are geared more towards deadlocks and starvation detection. These 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 [8] 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) [8]; SPIN [23] uses automata (expressed with the Process Meta Language (Promela)) and properties to be verified are expressed in Linear

Temporal Logic (LTL) [23]. In [17, 28, 41] 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 [17] 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 [18], rely on source code analysis to search for error states.

Model checkers use various search techniques: exhaustive search (SPIN [23]); graph exploration algorithms - such as depth first, breadth first and A* search techniques - constructing only relevant parts of the search space (HSF-SPIN [14], DELFIN+ [20]); heuristic searches such as GAs (Verisoft [18]) or Ant Colony optimization [4].

Direct, quantitative comparisons with the various model checking techniques we have encountered were not possible as some works, such as [14], 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 [18]. For others, such as [20], 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.

B. Related Work: Search-Based, Non-Functional Testing

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 [21]. Search-based, non-functional testing, the

closest to our aim, can be refined by goal into primarily five categories: usability, safety, execution time, buffer overflow and quality of service (QoS) [3]. Safety testing searches for inputs that violate a safety property and is probably the closest category to our work here. GAs and simulated annealing have both been used to generate inputs and sequences of inputs that aim at violating a safety property [3]. In [44], 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 [2], 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 [2].

Design-based verification works aim at uncovering deadlocks and starvation as well as data races. They do so without using search-based techniques. One approach for deadlock and starvation detection [26] 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 test and the underlying programming language concurrency mechanisms, specifically Java. The Symbolic Analysis Laboratory (SAL) model checker derives test sequences from the model, which are then executed with Concurrency Analyzer (ConAn), a deterministic, run-to-completion testing tool [26]. A similar work is presented by Lei, Wang and Li [30]. 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 [30]. They also serve to ensure that both code and design are consistent. Both works differ from our approach as our goal is to reuse existing UML design models rather than to create UML models specifically for testing language specific implementations.

Other works also aim at verifying concurrent systems in the context of detecting data races. Some of them [15, 16, 1, 25] do so using the code of the system under test. The work by Kahlon et al. in [25] 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. [10] 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 are very costly to fix. Furthermore, data races due to dynamically allocated shared resources might go undetected. Other works, such as Savage et al. [40], 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.

C. Related Work: Uses of MARTE

In the context of using UML profiles for concurrency, a number of works utilize MARTE's predecessor: the SPT profile. Such works, (e.g., [37]), mostly focus on performance analysis rather than the analysis of model properties. Other works such as [31] and [11] use the MARTE

profile. However, in [31], the profile is used to create an approach for real-time embedded system modeling along with transformations to execute those models. In [11], the authors aim at probing the capabilities of MARTE by applying it to a case study.

D. Summary

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. We begin with some background information by first introducing the concurrency fault taxonomy.

III. Deadlocks and Starvation

Concurrency introduces the need for communication between executing *threads*, which require a means to synchronize their operations. One of these means is shared memory communication, which ensures that shared resources are accessed individually and appropriately [13]. The most common techniques for shared memory communication are semaphores and mutexes. Semaphores, or counting semaphores, represent multiple access locks: at most n tasks (n>1) will have access to a shared resource [13]. The resource's *capacity* is the maximum number of threads executing within a shared resource. Mutexes are single access locks and are equivalent to a binary semaphore (n=1) [13]. Both types of shared memory communication will be referred to as *locks* in the remainder of this paper. Locks are first *acquired* before they are used, then later *released* when threads no longer need them. Threads waiting for access to a lock are placed in a conceptual *wait queue*. Wait queues define an access policy, e.g., FIFO, round robin, shortest-job-first, priority [7]. The various access policies require additional knowledge about threads. FIFO and round robin imply knowledge of *thread access times of locks*. These access times may be specified as ranges or definite values, although ranges are probably more common due to uncertainty at design time. Shortest-job-first implies knowledge of *thread execution times within locks* while the priority access policy implies knowledge of *thread priorities*.

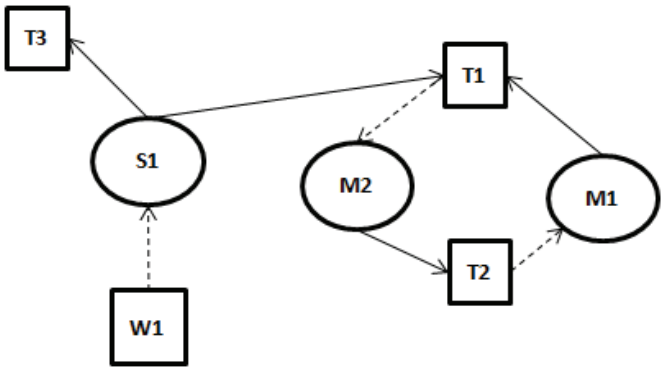

Fig. 1: Deadlock and starvation depicted in a RAG

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 [13]. Consider the Resource Allocation Graph (RAG) - which depicts the allocation of resources to threads [7] – in Fig. 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 conceptual 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.

In general, four conditions must be true before a deadlock occurs: 1. Threads share 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 [7].

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 [43]. In other words, if a thread is not allocated CPU cycles, it is never given the chance to execute, hence it starves. This type of starvation is often referred to as CPU starvation [32]. It is argued that scheduling algorithms largely influence the potential for CPU starvation [43]. This type of starvation, hence, lies outside the scope of this work since we are interested in starvation problems that are due to design decisions (e.g., decisions regarding priorities of threads). In the context of concurrency, starvation is related to locks, and is therefore coined lock starvation [32]. Much like CPU starvation, where threads wait for CPU access indefinitely, lock starvation occurs when some threads wait for lock access indefinitely [32], making them similar to a deadlock situation. However, unlike deadlocks, only some—not all—threads accessing a lock are at a

standstill. Consider, for example, a solution to the reader/writer problem. Readers are allowed to access the critical 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 Fig. 1, a situation arises where a writer (W1) arrives when readers (T1 and T3) are executing in the critical section, and hence W1is 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 [13].

To proceed with our method, we must first map both deadlock and starvation concepts, in particular those appearing in italics in this section, to UML and MARTE concepts, as they form the inputs of the genetic algorithm.

IV. MARTE Profile to Deadlock and Starvation Mapping

This section describes how we can fulfill the first of our three steps in achieving our goal. It demonstrates the feasibility of extracting all relevant concurrency information from UML/MARTE design diagrams. We begin by first describing the aspects of concurrency that are already present in UML. We then show how missing concurrency aspects can then be extracted from MARTE models.

In UML, active objects have their own thread of control, and can be regarded as concurrent threads [35]. Only extensions of the UML standard, such as the MARTE profile [33], provide mechanisms to model detailed information pertaining to concurrency. The MARTE profile is a replacement of the Schedulability, Performance and Time (SPT) profile [34]. 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. Much like its SPT predecessor, 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. 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.

Concepts are initially obtained from the foundations of MARTE, namely the Generic Resource Modeling (GRM) package. The `GRM::ResourceTypes` package introduces two stereotypes, namely `<<Acquire>>` and `<<Release>>`, that are of interest to us as they allow modeling the acquisition and release of resources, respectively. Resources cannot be accessed or released until both actions have been executed successfully. The Software Resource Modeling (SRM) sub-profile presents mechanisms for designing multitasking 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, 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`.

During system execution, resources may be shared by various concurrent actions and hence must be protected. Protected resources (i.e., locks) are stereotyped `<<SwMutualExclusionResource>>` in the `SRM::SW_Interaction` package. Attributes associated with this stereotype include `accessTokenElements`, which defines resource 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 would be stereotyped as `<<SwMutualExclusionResource>>`, whereby only one user at a time can access the seat for booking. A relationship of `<<Acquire>>` and `<<Release>>` must be present between the seats and user1. Each seat would additionally define its number of `accessTokenElements` as one, with a priority `waitingQueuePolicy` (an airline agent has higher priority than online users when reserving seats). 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. For the airline system, the time between seat reservations would be indicated as `interOccTime` with `execTime` designating the amount of time taken to book a seat.

This overview of MARTE illustrates that the input needed, i.e., the concepts related to deadlock and starvation discussed (presented in italics) in Section III, can be retrieved from a UML/MARTE design model. The mapping between those concepts and the profile is summarized in Table 1, illustrating the fact that we rely on three sub-profiles of the MARTE

profile. It is then clear that the information used by our GA can be automatically retrieved from UML/MARTE models, in particular from sequence diagrams where those stereotypes and tags are used, as illustrated in Section VIII on three examples.

Table 1: Concept to MARTE Mapping

Concept	MARTE Stereotype/Tag	MARTE sub-profile
Thread	`<<SwConcurrentResource>>`	SRM::SW_Concurrency
Lock	`<<SwMututalExclusionResource>>`	SRM::SW_Interaction
Lock acquire	`<<Acquire>>`	GRM::ResourceTypes
Lock release	`<<Release>>`	GRM::ResourceTypes
Wait queue access policy	`<<SwMutualExclusionResource>>/ waitingQueuePolicy`	SRM::SW_Interaction
Thread priority	`<<SwConcurrentResource>>/ priorityElements`	SRM::SW_Concurrency
Lock execution time	`<<gaStep>>/execTime`	GQAM::GQAM_Workload
Lock capacity	`<<SwMutualExclusionResource>>/ accessTokenElements`	SRM::SW_Interaction
Lock accessrange	`<<gaStep>>/interOccTime | <<gaStep>>/execTime`	GQAM::GQAM_Workload

V. Starvation Detection

We proceed with describing how the input required for our method, as described in Section III, is used for detecting starvation faults. As discussed in Section I, we are interested in finding thread execution interleavings that have a high probability of revealing starvation faults. These interleavings are obtained through thread interleavings of particular access times to locks. In other words, we are interested in those sequences that will result in the worst possible scenario of thread executions, namely lock starvation. Our objective is therefore a form of optimization. Hence, the values to be optimized to try to reach starvation are the particular access times of threads to locks. Techniques abound for solving optimization problems. These aim at efficiently searching the search space for a solution to the optimization problem. The search space is the set of all possible solutions to the problem. In our approach, we use Genetic Algorithms (GAs), which are known to perform well when the search space is large and complex - as is our case (Section VIII). GAs are based on concepts adopted from genetic and evolutionary theories [22]. They are comprised of several components: a representation of the solution (referred to as the

chromosome), a fitness of each chromosome (referred to as the objective or fitness function), the genetic operations of crossover and mutation which generate new chromosomes, and selection operations which choose chromosomes fit for survival [22].

A GA first randomly creates an initial, random population of chromosomes, then selects a number of these chromosomes based on a selection policy, and performs crossover and mutation to create new chromosomes. The fitness of the newly generated chromosomes is compared to others in the population. Following a replacement policy, individuals from the original population and children populations are merged into a single new population. The most commonly used replacement policy is elitism, whereby fitter individuals of the older population and newly created chromosomes are retained while less fit ones are removed. The process of selection, crossover and mutation, fitness comparison and replacement continues until the stopping criterion, such as a maximum number of generations [22], is reached.

Recall that in the context of concurrency, starvation occurs when threads cannot gain access to a lock because their executions are constantly 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 test 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 particular 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.

It is interesting to note that the chromosome representation and the mutation and cross-over operators (Sections V.A and V.B) are common to starvation and deadlock detection. What differs mainly among them is the fitness function (Section V.D). 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. Work is underway to adapt it to other types of concurrency problems.

It is important to note that the solution we propose below is applicable to any concurrent system, as long as the input discussed previously is available. In other words, the GA components are defined only once for targeted concurrency problems and what vary from system to system are the input values.

A. Chromosome Representation

A chromosome is composed of genes and models a solution to the optimization problem. The collection of chromosomes used by the GA is dubbed a population [22]. The values to be optimized during starvation detection are the particular access times of threads to locks, such that the target thread waiting for the target lock starves. These access times are the values that will be altered by the GA to try to reach a starvation situation. The access times must reflect schedulable scenarios. In other words, we need to ensure that all execution sequences represented by chromosomes are schedulable [19]. This entails meeting system specifications of periods, minimum arrival times, etc. Thus, we need to encode threads, locks and access times, which are available in the input model: `<<SwConcurrentResource>>`, `<<SwMutualExclusionResource>>`, and `<<gaStep>> / interOccTime` or `<<gaStep>> / execTime`, respectively (Table 1). 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 (V.B and V.C).

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 *lock access range* (Table 1). Access time is a specific time unit chosen within the acceptable lock access range. 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 starvation, 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 timing constraints. 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 lock access range. 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 starvation 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 [24].

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 interval. 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.

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 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 lock access range of the associated thread and lock, if start and end times are defined as ranges (Section IV).

Consider, for example, a set of three threads T1 (lock access range [23-25] time units), T2 (lock access range [15-22]) and T3 (lock access range [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.

B. *Crossover Operator*

Crossover is the means by which desirable traits are passed on from parents to their offspring [22]. 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 Fig. 2(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.

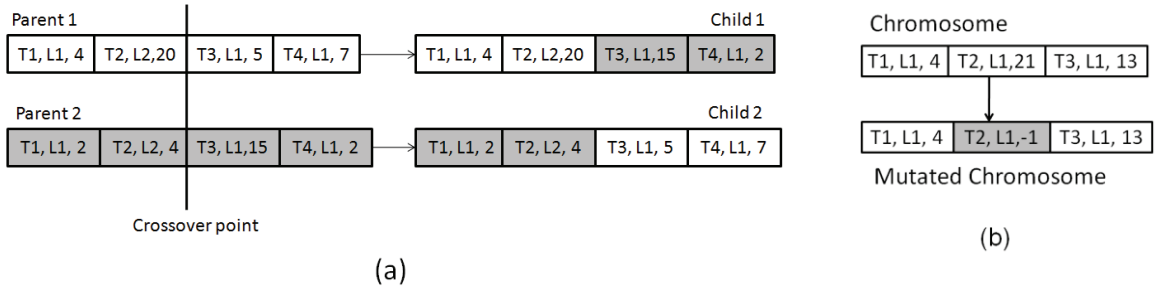

Fig. 2: (a) Crossover and **(b)** mutation examples

C. *Mutation Operator*

Mutation introduces new genetic information, hence further exploring the search space, while aiding the GA in avoiding getting caught in local optima [22]. 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 starvation. When a gene is chosen for mutation, a new

timing value is randomly chosen (using uniform distribution) from the range of possible lock 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 Fig. 2(b) is then valid. When the second gene of the chromosome is chosen for mutation (second chromosome in Fig. 2(b)), a new value (say, 27) is chosen from its lock access range [20-30]. Because this falls outside the time interval specified, the mutated gene is set to -1.

D. Objective Function

Recall that starvation detection requires the tester 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 A. The fitness function is weighted such that the longer the target thread spends waiting on the target lock, the greater its fitness.

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

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 $execThreads_i$ and $waitingThreads_i$ is the set of threads waiting for access to the target lock at time `i`. The sets $execThreads_i$ and $waitingThreads_i$ 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 (as further discussed in Section VI and VII).

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: $fitnessValue_0 = 1$
Time unit 1: $fitnessValue_1 = 1 + 1(fitnessValue_0) = 2$
Time unit 2: $fitnessValue_2 = 0 + 2\ (fitnessValue_1) = 2$
Time unit 3: $fitnessValue_3 = 0 + 2\ (fitnessValue_2) = 2$
$f((T1, L1, 0)\ (T2, L1, 0)) = fitnessValue_3 = 2$

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 target 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 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.

VI. Deadlock Detection

Deadlock detection [42] proceeds somewhat similarly to starvation. Our objective is to illustrate that we use the same kinds of inputs and that the main difference with starvation detection is the fitness function. It uses the same chromosome representation, crossover, and mutation operators. However, unlike starvation, deadlock detection does not require the specification of either target thread or lock. Deadlock detection also differs in the objective function:

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

`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 1). 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 Fig. 1. There are three threads waiting for access to a lock (T1, T2, W1), hence threadsWaiting>=2. Assuming that S1 has capacity 5, f(c) = 4 + 3 + 7 = 14. 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 Fig. 3, a modified version of Fig. 1. Here, f(c) = 7 + 2 + 7 = 16. The situation of Fig. 3 has higher fitness than Fig. 1, yet the latter is an instance of a deadlock while the former is not. In Fig. 1, a cycle is present: T1, M2, T2, M1, T1. In Fig. 3, 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, when `threadsWaiting≥2`. Cycles in a RAG indicated 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.

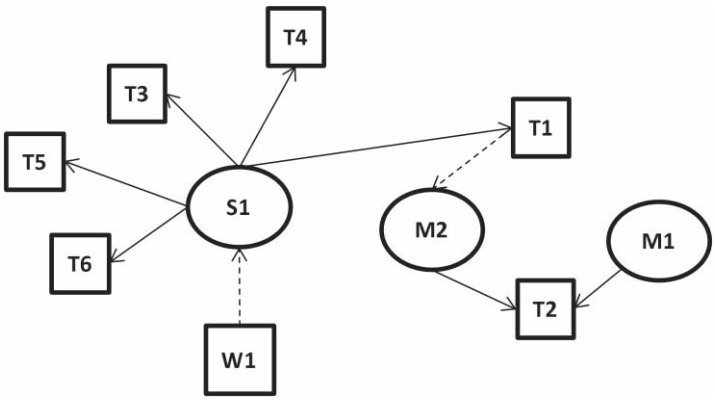

Fig. 3: Fitness function example

The fitness function in (2) 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 should increase 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 should increase 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.

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

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

Therefore, the time interval equals: `[0-(lt+ll)*2]`. Readers interested in more details on deadlock detection are referred to [42].

VII. Tool and GA Parameters

We have built a prototype tool, Concurrency Fault Detector (CFD), supporting our method. CFD is an automated system that identifies concurrency faults in any concurrent application modeled with the UML/MARTE notation. Currently, it can help identify deadlock and starvation faults, and work is in progress for the detection of other types of concurrency faults. CFD involves a sequence of steps. Users first input two 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 fault targeted: 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 (i.e., from its sequence diagrams).

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 V, and VI. 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 currently 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 (executing threads and waiting threads for each lock as well as lock access times), the

corresponding RAG and the fitness value. If no deadlock is found, CFD terminates, showing both the fitness value and output details of the highest fitness chromosome found.

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 fitness value as well as executing and waiting threads of the target lock for the highest fitness chromosome found. It also shows the times when the target thread executed within the target lock as well as the times the target thread requested access of the target lock. 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. This is however 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!
Times T1 accessed L1:
22
chromosome: (T1, L1, 22) (T1, L1, 129)(T1, L1, 236)(T1, L1, 343)(T1, L40, 23)…
```

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 800 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.

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 replace worst replacement scheme and 50% replacement, as suggested in [22]. The population size we apply is 200. This is higher than the size suggested in [22], but works more effectively for larger search spaces (see below). The selection operator is rank selector, whereby chromosomes with higher fitness are more likely to be chosen than ones with lower fitness [29]. 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 [6] and [22], respectively. The termination criterion we apply is number of generations. In particular, the GA terminates after 1000 generations if no deadlock or starvation is found. According to [39], the value of the termination criterion requires some knowledge of the application to determine an appropriate search length [39]. 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.

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 situation or starvation. This is the main reason why we developed a strategy that only reports one deadlock or starvation 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 requires, however, the problem detection to be efficient enough and scale up, even on large UML/MARTE models.

CFD is used to investigate whether scenarios can be generated where starvation or deadlock problems 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 timing constraints as well as the acceptable probability of not detecting a deadlock or starvation. If designers are bound by timing constraints, 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, assuming 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 constraints. 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.

VIII. Case Studies

We use our tool to run six case studies. These studies aim at evaluating the effectiveness of our method at detecting deadlock and starvation faults based on UML/MARTE design models. We also look at run-time efficiency though we realize that a great deal of improvement could be obtained with that respect by using more powerful hardware and distributed GAs [12]. Results demonstrate the scalability of our approach with respect to fault detection rates. The case studies cover a variety of different classes of 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 and starve problems, respectively.

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 before discussing results.

A. Deadlock Detection Case Studies

1) The Dining Philosophers Problem (Phil)

The renowned n-dining philosophers problem has commonly been used to demonstrate deadlock detection and avoidance [18, 20]. It is an interesting problem as it provides complex resource sharing as well as a large search space. The problem is summarized as follows: 40 philosophers 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. When they are thinking, philosophers do not access forks. When they 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 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 threads (philosophers) accesses to locks (forks) for at least one that yields a deadlock. This set of possible sequences is called the search space. To get an idea of the magnitude of the search space, the state space (states that the philosophers can be in, independent of the solution) is approximately[2] $1.2 * 10^{19}$. For the particular solution we use, factoring in the chosen time interval and thinking and eating times, the search space is approximately $1.9 * 10^{116}$.

A search space is further characterized by its complexity. Points in the search space that result in concurrency problems 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.

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

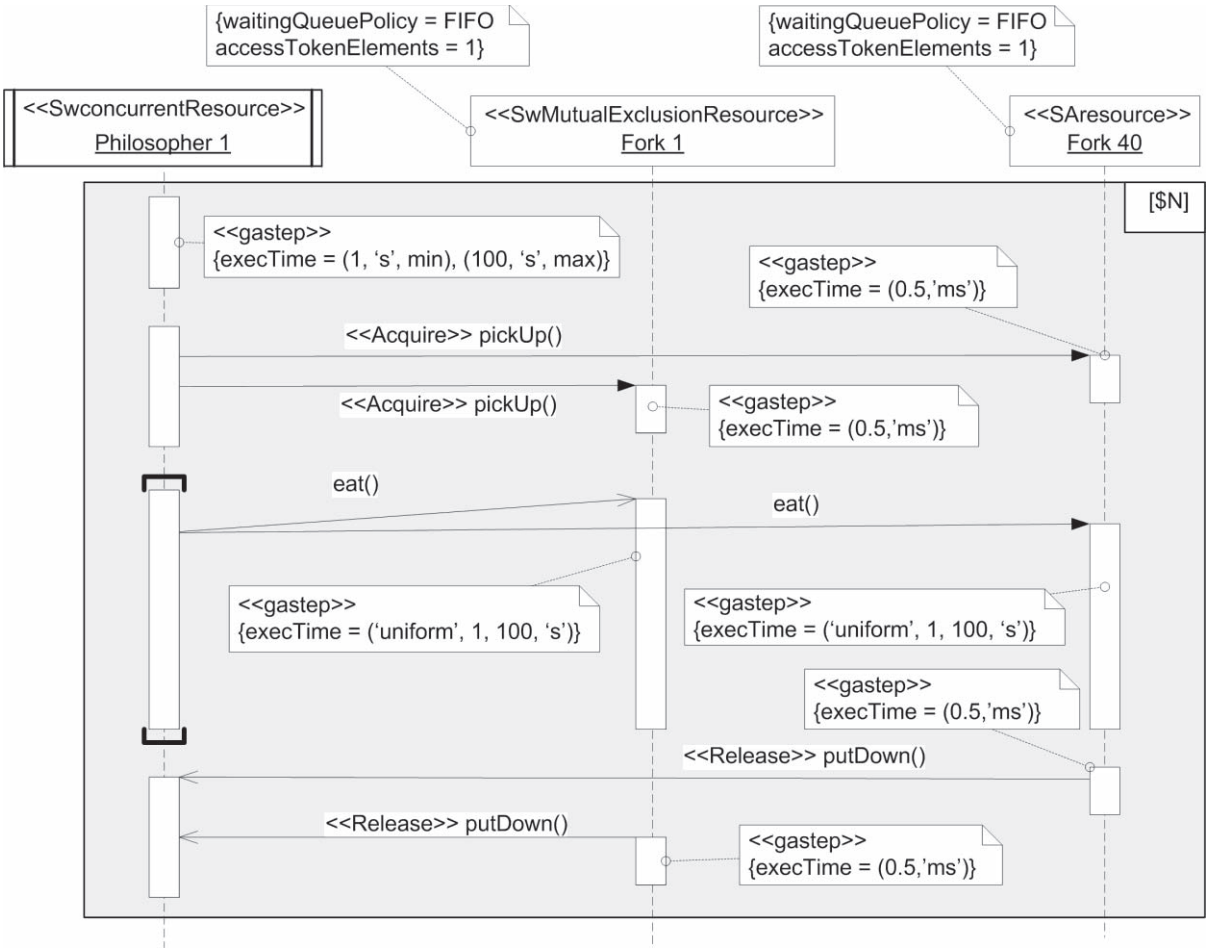

Fig. 4: Dining philosophers sequence diagram

Concurrency aspects for the philosopher problem are depicted in Fig. 4 for philosopher 1 and its 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. 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 (lock access range) 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, CFD requires two inputs. The first input is the complete sequence diagram. The second input is the time interval. We used our heuristic (Section VI): 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 is 400.

2) The Bank Transfer Problem (Bank)

The bank fund transfer is based on a simple banking functionality: fund transfer between accounts; and simulates multiple threads transferring funds among different accounts [46]. 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. 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. In this problem, a deadlock can occur if one thread is transferring from account A to 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 number of accounts and t is the number of threads[3]. For 10 threads and 50 accounts, the state space is approximately $7.7 * 10^{33}$. For the particular solution we use, the search space is approximately

2.7 * 10^{66}. Again, all the necessary information used by the GA, including data to determine the time interval, can be retrieved from the UML/MARTE model: see Fig. 5.

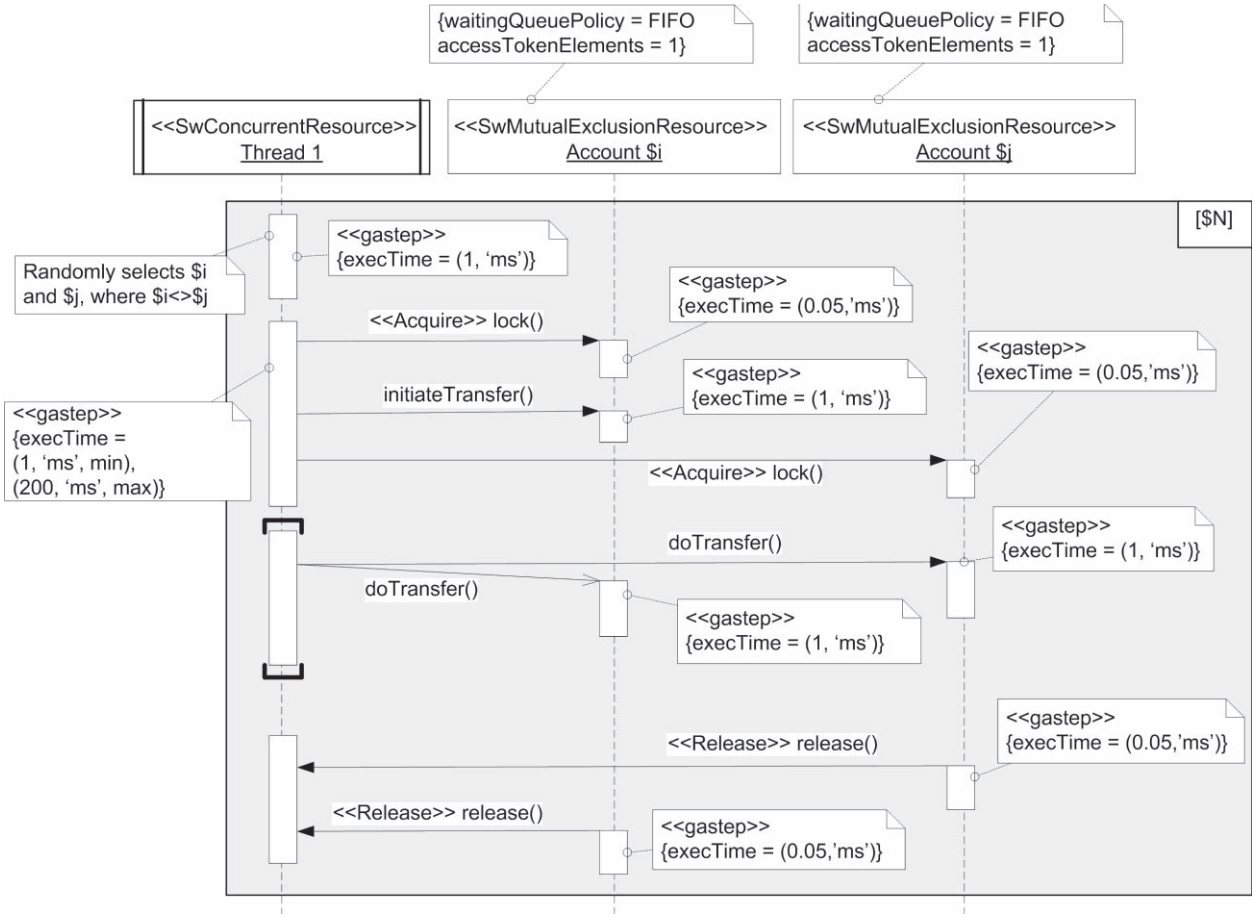

Fig. 5: Bank transfer sequence diagram

In Fig. 5, 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: first action stereotyped `<<gastep>>` that 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

[3] The search 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 search space is then the cross product of the thread search spaces

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 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.

For deadlock detection, the input 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.

For starvation, this problem would not be interesting as the designated target thread might never access the designated target lock.

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 Fig. 6.

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 test case scenario, which tests a number of threads and locks, as shown in Fig. 7:

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

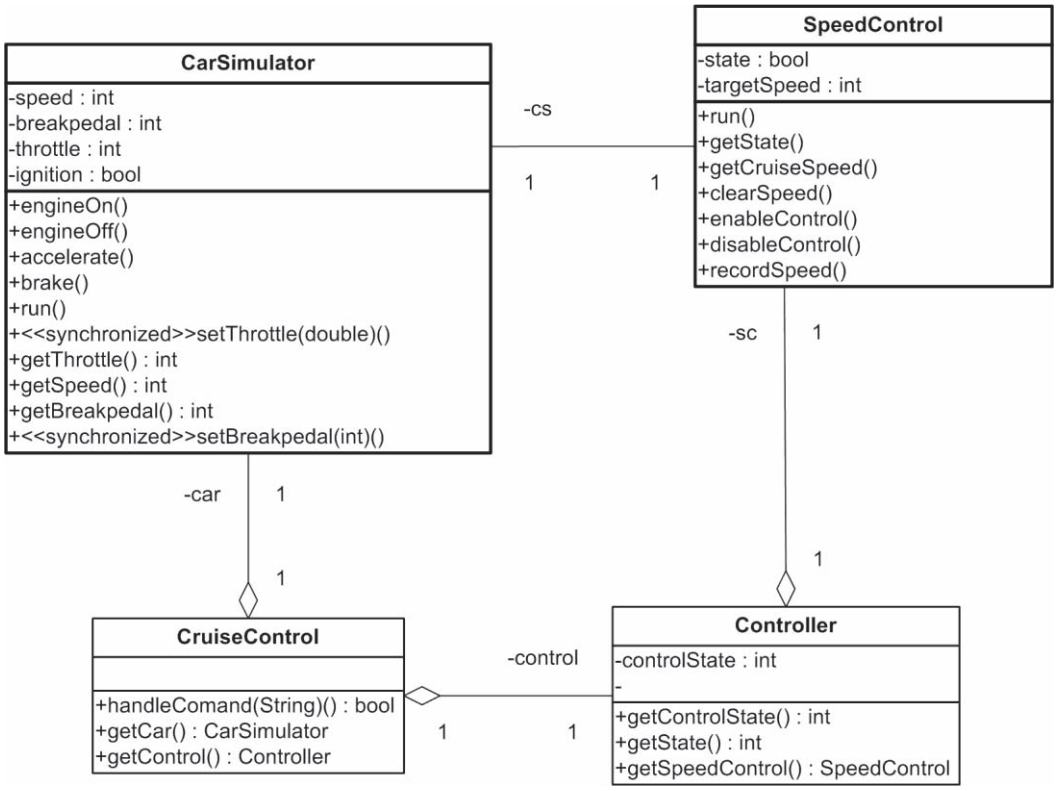

Fig. 6: Cruise control class diagram

A deadlock can occur during breaking when the `CarSimulator` is executing the brake command and the `SpeedControl` is in its periodic run. Furthermore, in these particular states, a deadlock shows up only when the timing is just right so that both acquire two locks (`brakepedal` and `throttle`) at just the right times.

For deadlock detection, the input 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.

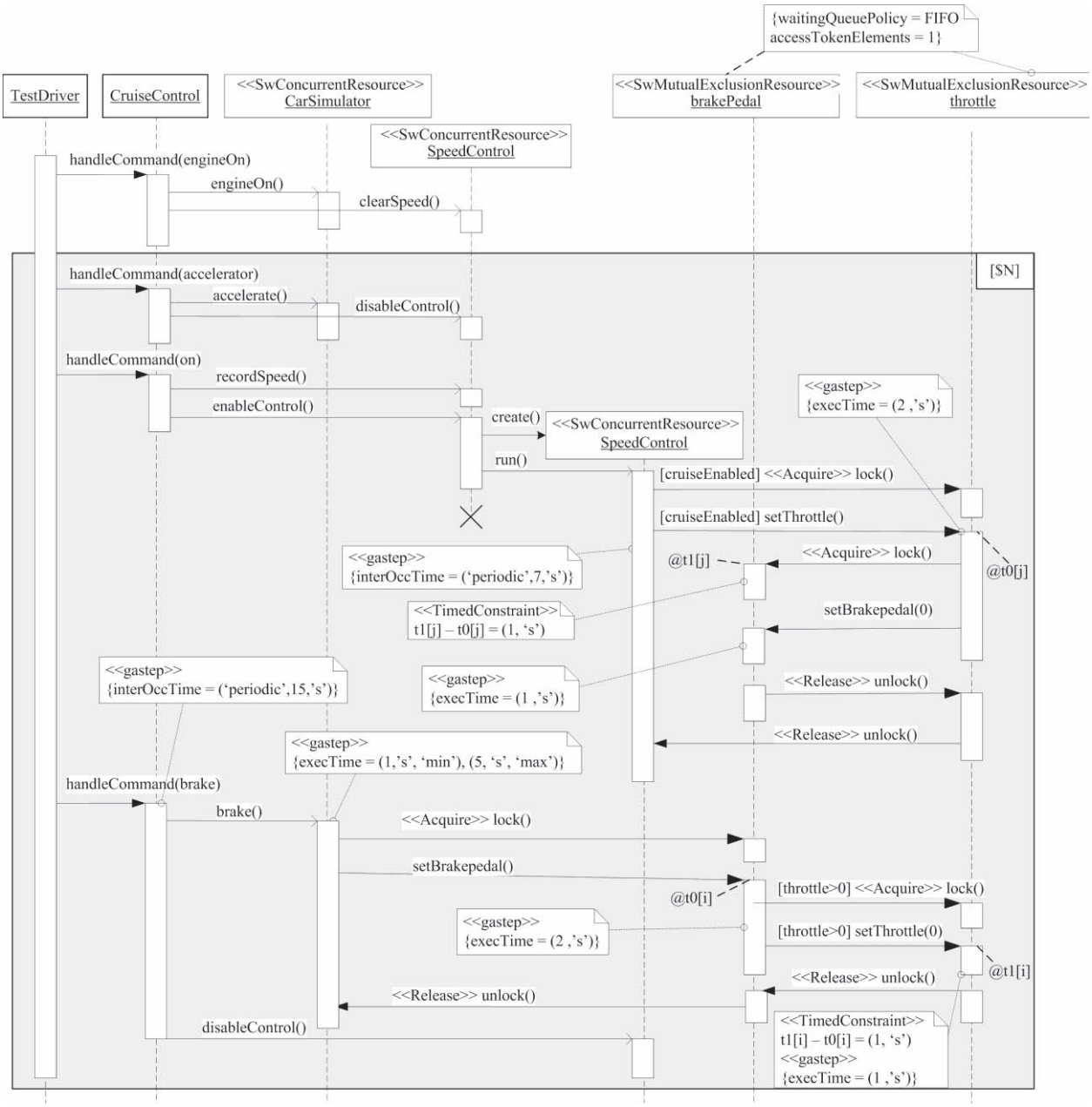

Fig. 7: Cruise control sequence diagram

It is important to note that, in practice when using a modeling tool, the sequence diagram of Fig. 7 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.

B. *Starvation Detection Case Studies*

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 follows, as shown in Fig. 8: 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. Philosopher 1 is assigned low priority while others are randomly assigned either a high or low priority. A philosopher can starve in this solution if the surrounding three philosophers constantly pick up the shared fork before the philosopher has the chance to do so. The state space[4] here is approximately $9.8 * 10^{37}$. The actual search space is $3.9 * 10^{232}$.

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

[4] The search 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 for an additional #phil-1 philosophers.

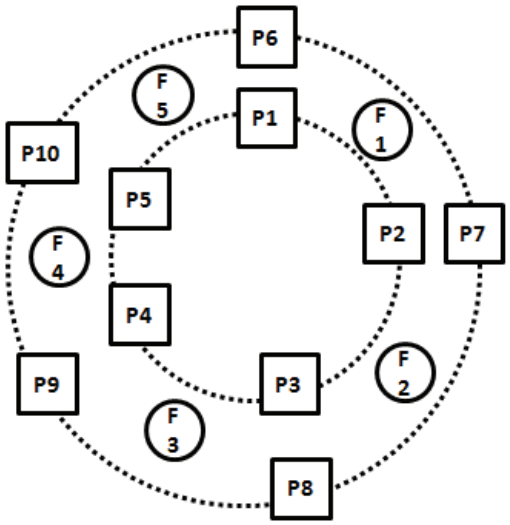

Fig. 8: Modified dining philosophers for starvation

2) The Starvation Problem (Starve)

The previous case studies were originally designed for verifying deadlocks. Bank is unfit for starvation, and Cruise is too simple for starvation detection. We hence created the starvation problem to create an environment where verification of starvation would prove more challenging. Six threads access a common lock with properties defined in Table 2.

Table 2: 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

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 $4.8 \text{ X } 10^8$: there are $4.8 \text{ X } 10^8$ 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 lock access range 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[5]. Hence, only 0.14% of the population results in starvation.

3) Modified Cruise Control Problem (ModCruise)

The same cruise control problem 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.

C. Case Studies Design

Here, we briefly describe the techniques used for deadlock and starvation detection, then we discuss how the case studies were set up to ensure that all techniques are comparable.

We use three different techniques to detect both deadlocks and starvations: 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 [5].

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

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

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 [38]. One random point is generated, then a neighboring point is generated by mutating the current point. 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 until a maximum number of sequences are generated [38].

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. All sequences for both random and hill climbing are generated in such a way that they satisfy the same constraints imposed on the genes when using a GA. As GAs are heuristic optimization techniques, variance occurs in the results they produce. To account for the variability in results, we ran each case study 25 times on an Intel Core 2 2.0 GHz processor. Twenty five runs is a reasonable number for the case studies provided: they execute within reasonable time constraints, they result in fine-grained enough data to compare the detection rate of search techniques, and they turned out to be sufficient to find the cases of starvation and deadlocks at least once in all case studies. Random generation and hill climbing were also run 25 times, with each run generating 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 detection. 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 of the 25 is executed.

D. Results

Results for the detection of both deadlocks and starvation are presented in Table 3 for each of the six case studies. For all cases, except Phil, we based our analysis on 100 runs. Phil takes a much longer time to run and results are sufficiently clear and statistically significant with 25 runs. To assess the significance of the difference between our proposed GA and the other techniques regarding detection rates, we conducted a Fisher Exact hypothesis test to assess differences in proportions. Two-tailed p-values[6] are reported in Table 3. A widespread practice is to consider p-values below 0.05 as significant.

1) Deadlock

For deadlock detection, most of the three techniques are capable of uncovering a fault, but with very different probabilities. For Cruise, the case study with a very small search pace, 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 3: 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 chromosome 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 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.

2) Starvation

For starvation, all three techniques are also capable of detecting concurrency faults. It is not surprising that the three techniques produce perfect detection rates for ModPhil and ModCruise, suggesting that these problems do not really need a GA to obtain starvation sequences. ModeCruise has a small search space and in ModPhil, though the search space is large, 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

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

deadlocks, we see that the relative performance of our GA depends on the size and complexity of the search space, but that it performs at least as well as the two other alternatives.

For both ModCruise and ModPhil, false positives (Section V.D) 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. For Starve, increasing the time interval to [1 100] results in no faults being reported, hence indicating that all previous results were false positives.

Table 3: Results

		Phil	**Bank**	**Cruise**	**ModPhil**	**Starve**	**ModCruise**
	Fault type	**Deadlock**			**Starvation**		
	Search space	$1.9 * 10^{116}$	$2.7 * 10^{66}$	245	$3.9 * 10^{232}$	$4.8 * 10^{8}$	175
Random	#Faults/#Runs	0/25	70/100	100/100	100/100	17/100	100/100
	Total Runtime (hr:sec:min: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
	Min. Runtime (hr:sec:min:ms)	0:00:23:164	0:00:00:019	0:00:00:000	0:00:00:006	0:00:00:041	0:00:00:000
	Max. Runtime (hr:sec:min:ms)	0:00:35:967	0:00:14:208	0:00:00:060	0:00:00:036	0:00:00:182	0:00:00:011
	Detection rate	0%	70%	100%	100%	17%	100%
	p-value	< 0.0001	0.0996	1.0	1.0	< 0.0001	1.0
GA	#Faults/ #Runs	22/25	81/100	100/100	100/100	98/100	100/100
	Total Runtime (hr:sec:min: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
	Min. Runtime (hr:sec:min:ms)	0:00:20:238	0:00:00:011	0:00:00:001	0:00:00:029	0:00:00:013	0:00:00:000
	Max. Runtime (hr:sec:min:ms)	0:43:11:998	0:00:05:904	0:00:00:051	0:00:01:010	0:00:05:233	0:00:00:026
	Detection rate	88%	81%	100%	100%	98%	100%
Hill Climbing	#Faults/ #Runs	7/25	30/100	30/100	100/100	86/100	100/100
	Total Runtime (hr:sec:min: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
	Min. Runtime (hr:sec:min:ms)	0:00:20:596	0:00:00:011	0:00:00:000	0:00:00:010	0:00:00:010	0:00:00:000
	Max. Runtime (hr:sec:min:ms)	0:05:27:992	0:00:15:941	0:00:00:055	0:00:02:435	0:00:00:250	0:00:00:036
	Detection rate	28%	30%	30%	100%	86%	100%
	p-value	< 0.0001	< 0.0001	< 0.0001	1.0	0.0029	1.0

3) Execution time analysis

Though we see from Table 3 that CFD (GA) systematically detects deadlocks and starvation 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 CFD to detect faults than for the other techniques. Execution times in both starvation and deadlock detection are driven by a number of factors. In the former case, the number of threads accessing the target lock 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. 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 the maximum number of threads produce a deadlock. CFD's execution times are still reasonable, considering the sizes of the search spaces (even with the most complex search space for deadlock Phil). Assuming designers would use a verification plug-in based on their MARTE designs, using our approach would take, in the worse cases and based on low-end hardware, a couple of hours or so. Such performance could be, 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.

E. Scalability

There are two dimensions according to which we can assess the scalability of the GA solution we propose: execution time and fault detection rates. With respect to the former, our results could be substantially improved by using better hardware and distributing GA computations [12], as further discussed below. 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. It is, however, interesting to assess

whether the detection rate is affected by the size and complexity of the search space. To do so, we ran the Phil problem with varying sizes of philosophers and forks. We chose the dining philosophers in particular because it represents a highly complex situation where deadlocks are difficult to detect. Fig. 9 shows our results in terms of (a) detection rate and (b) execution time when run on our CFD tool. The two figures are the result of a linear regression analysis and a smoothing spline fit, respectively. The number of philosophers used ranges from 5 to 60. This range is representative of most concurrent systems from the very simple, with five threads, to the very complex, with 60 threads accessing a specific lock. The goal of the figures is to graphically show the shape of the relationships and help quantify the impact of larger search spaces.

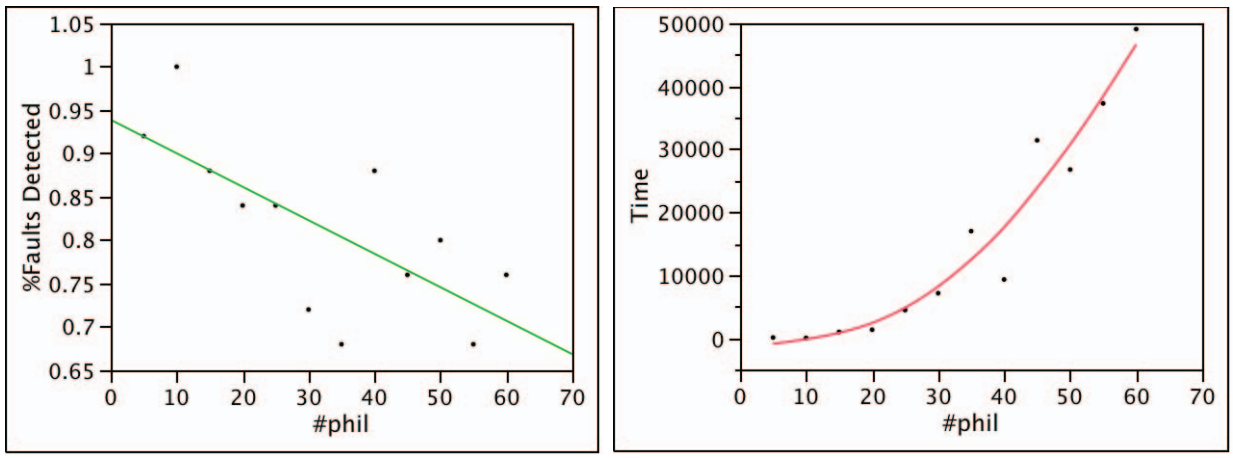

Fig. 9: Scalability in terms of **(a)** fault detection and **(b)** execution time in seconds

Plots in Fig. 9(a) indicate a lowest detection rate of 17/25 (68%), which is encouraging, and a highest rate of 25/25 (100%). The figure also shows the result of a regression analysis between the number of faults and philosophers: there is a significant, negative linear relationship (R^2 = 0.5) showing an average of one less fault detection (out of 25 runs), or a decrease of 4% in fault detection rate, per additional ten philosophers. While the decrease is linear, not exponential, and takes an additional 10 threads accessing a specific lock for a 4% decrease in detection rate, it is clear that for large spaces, more runs will be necessary to ensure high probabilities of detection.

The scatter plot of Fig. 9(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 powerful hardware and especially distributed GAs [12]. 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 [12]. The nature of genetic algorithms lends itself easily to distribution simply by allowing populations to evolve in parallel [12]. The gains of distribution strategies in terms of execution time are promising, as shown in [45]. For the traveling salesman problem (TSP), 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 [45]. 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 [45], 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 [12].

IX. Conclusions and Future Work

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 and deadlock 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. In this paper, we describe 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 sequences can help with debugging is reserved for future work.

We demonstrated the effectiveness our approach (in terms of concurrency fault detection) through six case studies run on our tool support. These showed very promising results in terms of detecting deadlocks and starvation 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 to achieve very high chances of detection.

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. Execution time could however be easily improved on more powerful hardware or by parallelizing the execution of the genetic algorithms [12], though this will be part of future work as it involves assessing and comparing strategies for parallelizing the GA. 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.

Because our approach incorporates 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 starvation or deadlocks are detected, our tool is still useful. The sequences outputted can then be used as future test cases. These would be particularly interesting as they represent scenarios close to faults; they may later uncover potential problems that can arise due to actual execution times of threads in locks, slightly differing from estimates.

Our approach provides a general framework that can be easily adapted to different types of concurrency problems: originally developed for deadlock detection, adapting it the problem of

starvation detection mainly required the definition of a new fitness function in the genetic algorithm. Though we focus on deadlocks and starvation here, work is underway to tailor our approach and tool (i.e., mainly its fitness function) to other concurrency problems associated with shared resources (namely data races) and thread communication.

X. Appendix A

The following demonstrates how the fitness function of equation (1) 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.

XI. Appendix B

The following demonstrates how the fitness function of equation (2) 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.

XII. References

1. M. **Abadi**, C. Flanagan, S.N. Freund, "Types for Safe Locking: Static Race Detection for Java," *ACM TOPLAS*, vol. 28, no. 2, pp. 207-255, 2006.
2. O. **Abdellatif-Kaddour**, P. Thevenod-Fosse, and H. Waeselynck, "Property-Oriented Testing Based on Simulated Annealing," http://www.laas.fr/~francois/SVF/seminaires/inputs/02/olfapaper.pdf. 2001.
3. 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.
4. E. **Alba**, F. Chicano, "Finding Safety Errors with ACO," *Genetic and Evolutionary Computation Conference (GECCO07)*, pp.1066-1073, 2007.
5. 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.
6. T. **Back**, "Self-Adaptation in Genetic Algorithms," *Proc. European Conf. Artif. Life*, pp. 263-271, 1992.

7. J. **Bacon**, *Concurrent Systems – Operating Systems, Database and Distributed Systems: An Integrated Approach*. England: Addison-Wesley, 2nd Ed., 1997.
8. G. **Behrmann**, A. David, and K.G. Larsen, "A Tutorial on Uppaal," http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf. 2004.
9. G. **Brat, K.** Havelund, S. Park, W. Visser, "Java Pathfinder Second Generation of *a Java Model Checker," Proc. Workshop on Advances in Verification*, 2000.
10. R. **Chugh,** J.W**.** Voung, R. Jhala, S. Lerner, "Dataflow Analysis for Concurrent Programs Using Datarace Detection," *ACM PLDI,* pp. 316-326, 2008.
11. S. **Demathieu**, F. Thomas, C. Andre, S. Gerard, and F. Terrier, **"First** Experiments Using the UML Profile for MARTE," *11th IEEE Sym. Object Oriented Real-Time Distr. Comp. (ISORC), pp*. 50-57, 2008.
12. 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.
13. A. B. **Downey**, *The Little Book of Semaphores*. Green Tea Press, 2nd Ed., 2008.
14. S. **Edelkamp**, A. Lluch-Lafuente, S. Leue, "Trail-Directed Model Checking," *Elec. Notes Theoretical Comp. Sci*, vol. 55, no. 3, pp. 343-356, Oct. 2001.
15. C. **Flanagan**, S.N. Freund, "Type-Based Race Detection for Java," *ACM SIGPLAN Notices*, vol 35, no. 5, pp. 219-232, 2000.
16. C. **Flanagan**, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, "Extended Static Checker for Java," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 234-245, 2002.
17. P. **Gagnon**, F. Mokhati, M. Badri, "Applying Model Checking to Concurrent UML Models", *Journal of Object Technology*, Vol. 7, No. 1, 2008.
18. 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.
19. H. **Gomaa**, *Designing Concurrent, Distributed, and Real-Time Applications with UML,* Addison Wesley, 2000.
20. S. **Gradara**, A. Sontone, and M.L. Villani, "DELFIN+: An Efficient Deadlock Detection Tool for CCS Processes," *J. Comp. System Sci*., vol 72, no. 8, pp. 1397-1412, Apr. 2006.
21. M. **Harman**, "The Current State and Future of Search Based Software Engineering," *Proc. Intl Conf Software Eng (ICSE 2007)*, pp. 342-357, 2007.
22. R.L. **Haupt**, and S.E. Haupt, *Practical Genetic Algorithms*. New York: Wiley-Interscience, 1998.
23. G. J. **Holzmann**, "The Model Checker SPIN," *IEEE Trans. Soft. Eng,* vol. 23, no. 5, pp. 279-295, May 1997.
24. C. **Jacob**, *Illustrating Evolutionary Computation with Mathematica*. San Francisco: Morgan Kaufmann, 2001.
25. V. **Kahlon**, Y. Yang, S. Sankaranarayanan, A. Gupta, "Fast and Accurate Static Data-Race Detection for Concurrent Programs," *LNCS*, vol. 4590, pp. 226-239, 2007.
26. S. **Kim**, L. Wildman, and R. Duke, "A UML Approach to the Generation of Test Sequences for JAVA-Based Concurrent Systems," *Proc. Austrl. Soft. Eng. Conf*., pp. 100-109, 2005.
27. A. **Kleppe**, J. Warmer and W. Bast, *MDA Explained – The Model Driven Architecture: Practice and Promise*. Boston: Addison-Wesley, 2003.
28. A. **Knapp** and J. Wuttke, "Model Checking of UML 2.0 Interactions", *Lecture Notes in Computer Science*, Vol. 4364/2007, pp. 42-51, 2007.

29. J.R. **Koza**, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge: MIT Press, 1992.
30. B. **Lei**, L. Wang, X. Li, "UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency," *1st Intl Conf Software Testing, Verification and Validation*, pp. 200-209, 2008.
31. C. **Mradiha**, Y. Tanguy, C. Jouvray, F. Terrier, and S. Gerard, "An Execution Framework for MARTE-Based Models," *13th IEEE Conf. Eng. Complex Comp. Sys. (ICECCS)*, pp. 222-227, 2008.
32. S. **Oaks**, and H. Wong, *Java Threads*, 3rd ed. California: O'Reilly Media Inc., 2004.
33. **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.
34. **OMG**, *UML Profile for Schedulability, Performance and Time Specification*, http://www.omg.org/docs/formal/05-01-02.pdf. 2005.
35. **OMG**, *Unified Modeling Language* (UML), version 2.2, http://www.omg.org/spec/UML/2.2/Superstructure/PDF/. 2009.
36. G. **Palshikar**, "An Introduction to Model Checking", *Embedded Systems Design*, 2004. http://www.embedded.com/columns/technicalinsights/17603352?_requestid=68213
37. D. C. **Petriu**: "Performance Analysis with the SPT Profile," *Model-Driven Eng. Dist. Embed. Sys.*, pp. 205-224, 2005.
38. 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.
39. 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.
40. S. **Savage**, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM TOCS*, vol. 15, no.4, pp.391-411, 1997.
41. F. **Schneider**, "UML and Model Checking", Proceedings of the 5th Langley Formal Methods Workshop, 1999.
42. M. **Shousha**, L. Briand, and Y. Labiche, "A UML/SPT Model Analysis Methodology for Concurrent Systems Based on Genetic Algorithms," *Proc. ACM/IEEE Intl. Conf. Model Driven Eng. Lang. Systems (MODELS)*, pp. 475-489, 2008.
43. R. **Sinclair**, *Codenotes for Java: Intermediate and Advanced Language Features*, Ed. Gregory Bill, Random House, 2002.
44. N. **Tracey**, J. Clark, J. McDermid, and K. Mander, "Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification," *Proc. 17th Intl Sys. Safety Conf.*, pp. 128-137, 1999.
45. L. **Wang**, et. al. "A Comparative Study of Five Parallel Genetic Algorithms Using the Traveling Salesman Problem", *Proc.12th Intl Parallel Processing Symposium*, pp. 345, 1998.
46. H. **Yang**, "Deadlock," *Java Tutorials – Herong's Tutorial Notes,* version 4.10, *http://www.herongyang.com/java/index.html.* 2006.