A new approach to detect potential race conditions in component-based systems

Jean-Yves Didier, Malik Mallem

▶ **To cite this version:**

Jean-Yves Didier, Malik Mallem. A new approach to detect potential race conditions in component-based systems. 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2014), Jun 2014, Lille, France. pp.97–106, 10.1145/2602458.2602470 . hal-01024478

HAL Id: hal-01024478
https://hal.science/hal-01024478v1

Submitted on 16 Jul 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A New Approach to Detect Potential Race Conditions in Component-based Systems

Jean-Yves Didier
IBISC laboratory
40, rue du Pelvoux
91000 Evry, France
Jean-Yves.Didier@ibisc.univ-evry.fr

Malik Mallem
IBISC laboratory
40, rue du Pelvoux
91000 Evry, France
Malik.Mallem@ibisc.univ-evry.fr

ABSTRACT

When programming software applications, developers have to deal with many functional and non-functional requirements. During the last decade, especially in the augmented reality field of research, many frameworks have been developed using a component-based approach in order to fulfil the non-functional requirements. In this paper, we focus on such a specific requirement: race conditions issues in component-based systems. We present a heuristic that analyses data flows and detects components that may be subject to race conditions. A toy example introducing the problem and the solution is developed and implemented under the ARCS (for Augmented Reality Component System) framework. We also show the results of our algorithm on real size applications using up to 70 components and compare those results with some obtained by developers who had to make exactly the same work by hand.

Categories and Subject Descriptors

D.2.11 [**SOFTWARE ENGINEERING**]: Software architectures—*Domain specific architectures*; D.2.13 [**SOFTWARE ENGINEERING**]: Reusable software—*Reusable libraries*; D.2.m [**SOFTWARE ENGINEERING**]: Miscellaneous—*Rapid prototyping*

Keywords

Components; Race conditions; Heuristic; Augmented Reality; Concrete example

1. INTRODUCTION

Augmented Reality (AR) aims at linking virtual entities with real objects through a combination of dedicated hardware and software. Programming such an application requires several skills. Some are purely related to programming techniques and execution environment. They are called non-functional requirements whereas others are related to techniques more specific to the AR field and are therefore called functional requirements (such as image and signal processing, interactions in mixed reality, virtual reality and computer graphics, content authoring, etc).

As a field of research of a growing interest, many researches have been conducted to propose software frameworks tailored to AR needs aiming at reducing the burden for developers to consider non functional requirements when developing. AR is also technology dependent and thus, piece of software piloting these technologies should be as replaceable as the used technologies. The findings of new algorithms and methods are also driving architectures to more and more modularity and genericity. Over the past decade, many software framework for AR [10] have been proposed and they use a component-based scheme. Component based solutions are popular in AR framework since they usually propose software architectures that will address most of the non functional requirements of AR such as modularity, genericity, and reusability. To these requirements we can also add distributed computing which is raised by the current hype in cloud computing and multi-threaded computing which is the target of this paper.

Multi-threading is and has been used in the past to build AR software ([1] is an example of such use) because it can reduce the end-to-end latency of an application in the case several input sensors are needed by the system to extract characteristics from its environment. Moreover it can profit from the multicore processor architectures that were introduced in the consumer market since several years ago. Inexperienced programmers usually have difficulties in mastering thread safety and concurrent access issues. Therefore, furnishing tools that relieves programmers from the burden of managing such issues may be an interesting added value.

The remainder of the paper is organised as followed. First, we will present some related component based systems used in the AR field and introduce our own framework named ARCS (for Augmented Reality Component System). Then, in section 3, we will focus on the problem of concurrent access management in component systems and provide some hints about how to solve the problem. The same section will also introduce some of the general purpose techniques found in the literature to detect where race conditions may occur. We will also propose a heuristic that automatically finds components for which concurrent access is an issue. Section 4 will show how such a heuristic and the associated solution can be implemented in our framework in toy case and we will finish by giving the future road-map on this research and conclude.

2. RELATED AND PREVIOUS WORKS

2.1 Component based AR frameworks

Modular software architectures and frameworks for AR have been addressed during these past years by almost thirty different projects of frameworks. The surveys written by Endres et al. and Huang et al. [10, 17] are quite exhaustive. Therefore, we chose to cite the most prominent component based frameworks.

Amongst these frameworks some are intend to distribute components over a network and thus do not need to manage interference between threads of the same program. This is the case for the StudierStube project led by the Technical University of Vienna [14]. Its main principle is that each user has its own workspace that may be partly shared with other users. Similarly, the DWARF [3] (Distributed Wearable Augmented Reality Framework) project uses decentralised and distributed services. This framework relies on CORBA [23] (Common Object Request Broker Architecture) to provide services. As an example, each input sensor is associated to a service broadcasting data to other services (that could be filters, rendering loops,...). Each service is described by XML data and can be dynamically linked to the current application, which is seen as a distributed data flow. At start-up, applications only need one component manager. This last one discovers step by step other services in order to integrate them at execution. Some other frameworks are following the same guiding precepts such as the AMIRE (Authoring Mixed Reality) project [8] or MORGAN [24].

Another class of component based frameworks for AR do not rely on distribution mechanisms and thus sometimes involves multithreading programming. For example, Tinmith [25] is a library written to develop mobile AR systems. This framework is based on data-flow description and is a library of hierarchical objects. It manages data-flow from sensors, many data filters and rendering components. Objects written in C++ rely on a callback system and data are serialised using XML. Communications between objects are managed through a data-flow graph. Other frameworks from the same category have been proposed such as VHD++ [26], MRSS [18] (Mixed Reality Software Suite) or Avango NG [20].

An AR framework must cope with works in progress and future works built on the present ones. It should be able to integrate technologies of tomorrow in terms of new devices and algorithms. Works in progress would then require flexibility, future works extensibility and future works built on top of current ones reusability. One of the classical answers is to use a component based software architecture because it allows to separate an application into several components, that are, according to Szyperski [34], *"a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system"*. Another benefit associated to reusability is the ability to rapidly prototype applications.

2.2 Previous work: the ARCS framework

ARCS, for Augmented Reality Component System, is a project we started in 2006 [6, 7, 19]. As a framework, it provides several models that should be followed in order to build AR applications. Some choices are made concerning the application model as well as the component model. We describe them in order to introduce later how we will manage concurrent access in such an environment since it is our target framework.

2.2.1 Application model

The ARCS application model describes applications as a set of process (actually a set of threads). Each one of them is controlled by a finite state machine. When the internal state of the state-machine changes, it triggers changes in how components are connected to each other. A set of connections as well as a set of component invocations (in order to initialise them and launch data processing) is called a *sheet*. According to this, each thread is, at a given time, in a given state that corresponds to an active sheet. Sheets are sharing the same components instantiated from a component pool. Therefore, components do not belong to any thread in particular so they may be invoked by different threads.

As we said before, each process maintains one active sheet at a time. A sheet activation cycle follows the following steps:

- The controller (state-machine) receives a token (from a component in the active sheet) that will trigger one of its transitions and then change the controller state;
- All components from the current sheet will be disconnected from each other;
- The controller new state corresponds to another sheet. First, some invocations, called *pre-connection invocations*, are performed on the components to properly initialise them;
- Then connections are established according to the new sheet description;
- Post-connection invocations are finally performed in order to launch the actual processing of data.

2.2.2 Component model

ARCS describes components as entities having signals (outputs) and slots (inputs). The signal/slot mechanism is well known since it is commonly used in graphical user interface libraries such as Qt for instance. It is also deriving from the observer design pattern [15]. Thus, the communication through a signal/slot connection is synchronous. Composition of components can be performed in two different ways: the first one is through *connection composition*, where a component emits a signal that is processed by a slot from another component. The second one is through *invocation composition*: a component is passed as a parameter to a slot of another component.

Amongst its specifics, ARCS provides an abstract component model that allows to introduce new component types or new component behavior as long as they respect the signal/slot scheme.

This abstract model includes the following functionalities:

- Instantiation and destruction of the actual component;
- Signal/slot and connection management;
- Serialisation/deserialisation: this is mainly used at the instantiation of the actual components. Deserialisation allows to configure the component according to string contents passed through an XML description of the application.

In order to complete the integration of other component families, the framework has also abstract type factories that parse XML descriptions and instantiate directly objects of the considered type that can be used as parameters of components invocations in order to initialise the latter ones.

These functionalities are also exposed in the dynamic libraries the ARCS engine can load at runtime by parsing XML descriptions of an application. They usually contain:

- Type factories, in order to extend ARCS with new types when needed and serialise them;
- Native component factories that instantiate components handled by the previous version of the engine and that are the privileged components in the framework;
- Family component factories in order to make ARCS compatible with other component systems.

To finish, in technical terms, ARCS is written in C++ language for performance reasons and depends on the Qt library [29].

2.2.3 Handling other non-functional requirements

In ARCS , we decided to develop an engine that is as lightweight as possible. Therefore, the engine in itself does not implement all the non-functional requirements that would be necessary to make all kinds of applications. Nevertheless, it is possible to explore other ways to add non-functional requirements when they are really needed. As an example, we already propose in the framework a mechanism to create distributed applications over a network [5].

As defined here, the ARCS engine is able to create threads and perform different tasks inside these threads but there is no concurrent access management developed in component model. We will now see how it is possible to design and implement such a mechanism in a component based system behaving like ARCS.

3. MANAGING CONCURRENT ACCESS IN COMPONENT BASED SYSTEMS

Managing concurrent access in component based systems depends on the characteristics of components that are used and their environment. For example, in an environment using CORBA components, concurrent access is managed transparently due to the serialisation mechanisms that occur when the CORBA bus is used. Therefore, we will first outline the main characteristics of the component system on which we will work as a basis for our hypotheses.

3.1 Component system main characteristics

We are considering components as stated by Szyperski [34]. To sum it up, component are pieces of software that are reusable and subject to composition in order to produce the final software.

Here, we will restrain the kind of components we are using. We are interested in components having the following characteristics:

- Components have separated inputs and outputs;
- Even if components are black boxes (i.e. we do not know how it is implemented inside), they adhere to the following internal behaviour: if one an input is triggered, it will trigger outputs synchronously (none to all outputs could be triggered). We call this property *internal synchronicity*. However, without a thorough analysis of the behaviour of the black box, we do not known which output is triggered by which input. Therefore, we will consider that when an input is triggered, every output is triggered;
- Components are communicating synchronously: outputs from a component trigger inputs from another component. Outputs then trigger function calls. This property is the *external synchronicity*;
- Communications are supposed unidirectional: data are flowing from outputs to inputs;
- Components are configured by passing arguments to some of their inputs. Inputs may also be used to start processing functions;
- Components are black boxes but communication channels (or *connections*) between components are known and static, that is to say, persistent in time or at least for a known duration.

Such components are also working in a specific execution environment. Concerning this execution environment, we make the following suppositions:

- It is multi-threaded;
- Component configuration and initialisation may be performed by different threads;

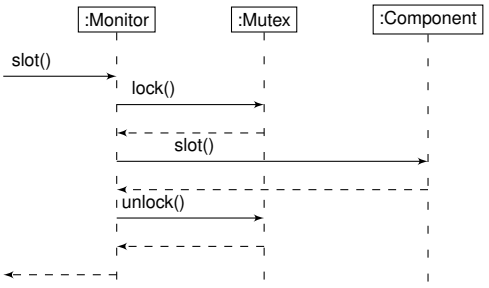


Figure 1: Behaviour of a monitor component wrapping the original component in a mutual exclusion.

3.2 Problem statement: concurrent access

Now that we have described the main characteristics of our component system and its execution environment, we can deduce some problems that will rapidly arise: how can we process issues concerning concurrent access or, more generally, thread-safety ?

There are at least three solutions in order to give a thread-safety property to such a system:

1. We can systematically implement thread safety mechanisms inside components. However, it is generally costly, especially if the component, in its execution context, does not require it;
2. We can implement it by hand in components that may be subject to concurrent access. The weak point here is that a component is a piece of reusable software and therefore it may be put in a situation that was not planned during the design phase of the component;
3. We implement a specific heuristic inside the execution environment that will ensure that components that are subject to concurrent access in their execution context will be guaranteed thread safe. This third way is the one we would like to investigate further in this paper.

Restated in other terms, we would like to give a thread-safe behaviour to components (that do not possess it) by analysing their execution context. We believe it has other strong advantages, one of them being that component developers would not have to worry about thread safety. This is also interesting because inexperienced programmers usually have difficulties in mastering thread safety and concurrent access issues.

3.3 Outline of our solution

The idea we will express here is just an intermediate solution that will give some hints on how the problem could be solved. We intend to give a simple, but practical solution. The main idea is to construct a wrapper component that will regulate concurrent access to the component it shelters. To achieve this, we apply the *monitor object design pattern* [31] as it is done, for instance, in java with the `synchronized` keyword. Components prone to concurrent access are then considered as a resource that will be properly framed inside a mutual exclusion (*mutex*) as represented in figure 1. For each component to protect from concurrent access, a *Monitor* will be generated, associated to an object managing a mutual exclusion (*Mutex*). Therefore, each call performed on *Component* is rerouted to *Monitor*.

Mutual exclusion are known to solve partly the problem of race conditions but they are known to be sensitive to other problems such as deadlocking (where two components are waiting for each

other and thus are locking each other, therefore the application cannot perform its intended task) or resource starvation (it occurs when two components are competing for the same resource and one of them is perpetually favoured over the other, thus tasks that were intended to be performed never happen). It would require another detailed analysis that we do not intend to present in this paper.

Our problem is then transformed into how to determine which component should be monitored? Before proposing a heuristic in order to solve it, we will briefly review some techniques used to solve such a problem.

3.4 Concurrent access in literature

In the literature, two complementary problems are addressed: the race detection in itself and the minimal number of places where mutual exclusion should be placed.

Race detections are detected using different approaches. The static approach enforce type systems [**?**], directly analyse the source code of the application [33, 28, 35] or analyse dataflow [11]. Such approach is not directly usable since we do not have access to the source code of components in every situation. The dynamic approach analyse actual execution traces. Algorithms and tools [30, 2, 27, 13] developed mainly rely on Lamport's happens-before relation [21] or use lockset analysis (a lock set is the set of locks that protect access to a shared variable). The tools have complete access to program context but will also reduce the performances of the program actually analysed and adds to the overhead of running components. Other techniques are also used such as post-mortem analysis but they are not suitable in our case.

Finding the minimal number of places where mutual exclusion should be placed is often called the Minimum Lock Assignment (MLA) problem. Many solutions have been proposed so far to determine where to put mutual exclusions. However, they usually concern parts of the software code (the potential critical sections) and not components as defined by Szyperski as we can see in [32, 16, 9, 37]. In such methods, critical parts of the code are annotated. Then a special compiler will determine where to put mutual exclusion by using special heuristics. MLA is known to be a NP-hard problem so other methods rely on a different approach: the idea is to determine specific use cases and associated use case scenarios and assess the behaviour of the software with respect to this scenario [12, 22]. If an interference is detected, then a mutual exclusion will be put where it occurred. Using use-case scenarios is not a solution that suited our needs since it needs some preparations. Determining MLA is also not our primary concern: the idea is not to be optimal but to relieve the developer from the burden to review his code in order to determine where to put mutual exclusions. The method must also be lightweight, since it should be run on the fly when the description of the application is loaded by our framework's engine. Fortunately, one of the conclusions in [16] is that "high accuracy in program analysis is not always necessary to achieve generally good performance". Therefore, we will provide a heuristic at component level that determines which components might be subject to interference.

3.5 A heuristic to find components subject to race conditions

To determine the solution of our problem, we construct a direct graph. Each vertex of the graph represents a component to which is associated a set of labels (the different threads in the application are also considered as components). Each time components are communicating together, a directed edge between the vertices associated to the components is drawn. The direction of the edge matches the direction of the communication.

We will now expose a heuristic that will determine what are the candidate components subject to monitoring.

3.5.1 Notations

- Let $\mathcal{C}$ be the set of components in the system. This set is also the set of vertices in the directed graph;
- Let $\mathcal{M} = \varnothing$ be the set of components that will be candidates to mutual exclusion monitoring;
- Let $\mathcal{L}$ be a set of labels. At initialisation, there are as much labels as threads in our system. $\mathcal{L}$ is very likely to grow, so the easiest way is to define $\mathcal{L}$ as a subset of $\mathbb{N}$ where $\mathcal{L} = [\![1; n]\!]$ where n is the number of threads;
- Let $\mathcal{E}$ be the set of edges of the directed graph;
- Each component c of $\mathcal{C}$ has a property named `labels` which will be noted $c.labels$. It is the set of labels associated to the component c. $c.labels$ is therefore a subset of $\mathcal{L}$;
- Each edge e of $\mathcal{E}$ has two properties: $e.head$ is the head of the edge and $e.tail$ is the tail of the edge. Note that $e.head$ and $e.tail$ are elements of $\mathcal{C}$;
- $|\mathcal{S}|$ is the cardinal number of the set $\mathcal{S}$;

3.5.2 An algorithm to find components to monitor

Our heuristic is exposed in algorithm 1. Apart from its initialisation, the heuristic is a loop iterating over three phases:

1. The *propagation* of labels along the edges (lines 14 to 20);
2. The *marking*, where candidate components subject to monitoring are marked (lines 22 to 24);
3. The *pruning and label generation* where some of the components are discarded because they do not need to be monitored with a mutual exclusion. Additional labels are created and labels associated to vertices are reinitialised (lines 25 to 35).

These steps are reiterated until the set of components to monitor is computed.

The initialisation consists in generating the first set of labels (noted $\mathcal{L}$ in the algorithm) and attach them with components considered as threads. Each component that is a thread is initialised with its own label and all other components are associated to an empty set of labels.

The propagation propagates labels along the edges until the label sets associated to each vertex is stabilised. Therefore, we know which thread can access to which component. It corresponds to the *internal* and *external synchronicity* hypothesis: if a thread accesses to an input, therefore it accesses to all corresponding outputs of the same components and inputs that are connected to these outputs.

The marking step creates a temporary set of components (noted $\mathcal{M}_c$) and puts into it components that are very likely to monitor. To find such components, we look at the incoming edges. If the originating component of such an edge is not accessed by the same threads than the target component of the same edge, then it means the component is subject to concurrent accesses and is to be added to the list of components to monitor.

The pruning step removes from study the set of components that are accessed by at most one thread since they are not concerned by concurrent access issues and the edges that come along. Each component that was marked to be monitored is generating a new label (it means we consider the data sent by it thread-safe since it is monitored and therefore it behaves as the data was sent by another thread) and all other components associated labels are reset.

Then we reiterate until there is not any component left to process.

Algorithm 1: Heuristic to determine components to monitor

Data: $\mathcal{C}$ – set of components, $\mathcal{E}$ – set of edges
Result: $\mathcal{M}$ – the set of components to monitor

```
// Initialization
M ← ∅;
L ← ∅;
l ← 1;
foreach c ∈ C do
    if c represents a thread then
        l ← l + 1;
        L ← L ∪ {l};
        c.labels ← {l};
    else c.labels ← ∅;
end
repeat
    // Propagation
    repeat
        applied ← false;
        foreach e ∈ E do
            if e.head.labels ≠ e.head.labels ∪ e.tail.labels
            then
                e.head.labels ←
                e.head.labels ∪ e.tail.labels;
                applied ← true;
            end
        end
    until applied = false;
    // Marking
    M_c ← ∅;
    foreach e ∈ E do
        if e.head.labels ≠ e.tail.labels then
        M_c ← M_c ∪ {e.head};
    end
    // Pruning and label generation
    foreach c ∈ C do
        if |c.labels| ≤ 1 then C ← C \ {c};
        else
            if c ∈ M_c then
                c.labels ← {|L| + 1};
                L ← L ∪ c.labels;
            else c.labels = ∅;
        end
    end
    foreach e ∈ E do if e.head ∉ C or e.tail ∉ C then
    E ← E \ {e};
    M ← M ∪ M_c;
until C = ∅;
```

3.5.3 Applying the heuristic: an abstract example

Figure 2 is illustrating how our heuristic is run. At first, we consider the set of components displayed in 2(a). We introduce in figure 2(b) two new components T_1 and T_2 that are actually the threads that start the application by performing invocations on components A and B. The label set associated to each component is noted between braces. Therefore, components T_1 and T_2 are associated to labels 1 and 2 and other components are associated to an empty label set. According to our heuristic, we then perform a first propagation and marking step (see figure 2(c)). Two components are noted as being subjects to monitoring, here C and D (they are noted with a star $*$). They are marked because preceding components do not share the same set of labels. Then, in the pruning phase (figure 2(d)), the components T_1, T_2, A and B are discarded since they all have only one label associated to them (it means they do not need to be monitored). New labels are generated for C and D that are respectively 3 and 4 whereas other components are associated to empty label sets. We then reiterate through the same steps a second (figure 2(e)) and a third time (figure 2(f)). At the end of the third iteration, since we had a cycle in the graph, all components are associated to empty label sets (figure 2(g)), therefore the algorithm completes in the next step. If we look at the set of components to monitor computed by the algorithm, we will find $\mathcal{M} = \{C, D, E\}$. This result is consistent since components C and D can be accessed through A and B that are invoked by two different threads. The outputs of C and D can come from the first or the second thread, therefore E needs to be monitored too.

Another side effect illustrated in this example is that mutual exclusions should be re-entrant. That is to say, a thread that has locked a mutual exclusion should still be able to enter in the component it has locked a second time. If this condition is not respected, it could lead to a lock in our abstract example since we have a communication cycle between D, E, G and F. Therefore, D can be locked by a thread and the same thread may access to D later through E, G and F.

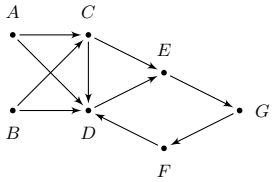
4. CONCURRENCY MANAGEMENT APPLIED IN ARCS

4.1 A toy case introducing a race condition

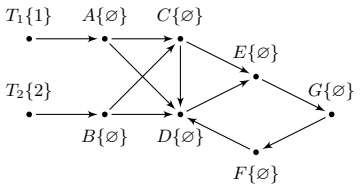
In order to understand how it works, we will not consider a real AR application but rather a toy case implementing a race condition. It is organised around three main components: two components (named *loop1* and *loop2*), each in a separate thread, iterate from 0 to 9, and one component (named *sum*) adds all iteration values received from the two other components. The components are, by default, developed without any code that synchronises them (Fig. 3 is showing their pseudo-code). Therefore, our *sum* component, which should reach a total of 90 ($2 \times \sum_{i=0}^{9} i$), does not necessarily reach the right sum in a multi-threading context (the counting slot is called from two different threads and therefore *tmp* and *sum* are in a race condition situation).

The ARCS framework strictly adheres to the component system and environment characteristics we presented in section 3.1. But, as a practical framework, it will require some adjustments and additional components in order to conform to its application model. Figure 4 represents the simplified connection graph of the toy case (as implemented using ARCS – see also figure 5) where we can find the main components as well as thread controllers (state-machines *s*, *s1* and *s2* which are mandatory in the application model[1]. Here

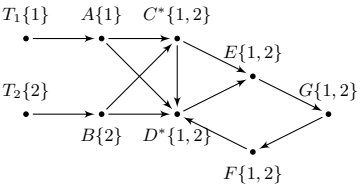
[1]One can notice difference between the abstract data flow and the one presented in the screen capture. This is due to the fact the



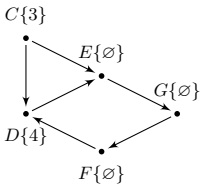
(a) Initial set of components



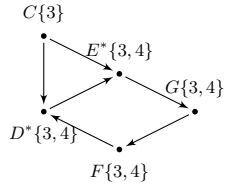
(b) Initialization step of the heuristic



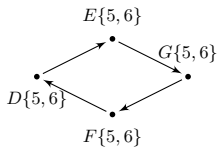
(c) Propagation and marking (1^{st} iteration)



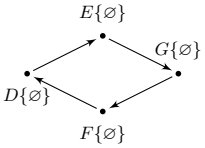
(d) Pruning and label generation (1^{st} iteration)



(e) Propagation and marking (2^{nd} iteration)



(f) Propagation and marking (3^{rd} iteration)



(g) Pruning and label generation (3^{rd} iteration)

Figure 2: Iterations of our heuristic from initialization to the last step before completion

```
For i from 0 to 9
  RandomSleep()
  Emit i
EndFor
Emit "end"
```

(a) *Loop* pseudo-code

```
tmp <- i
RandomSleep()
sum <- sum + tmp
```

(b) *Sum* pseudo-code

Figure 3: *loop* **and** *sum* **components pseudo-code.**

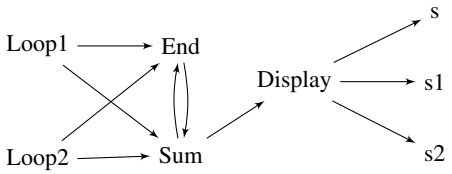


Figure 4: Simplified connection graph of components in the toy case.

s,*s1* and *s2* are only composed of two states: one initial and one final state in order to shut down the application). Some logical glue components are also used: *end* which triggers the end of the application when *loop1* and *loop2* have finished iterating, and *display* which displays the final result. *s*, *s1* and *s2* are controllers that may be accessed by different threads. A first thread triggers *loop1* and a second thread triggers *loop2*. Therefore, if we apply our heuristic, components *end*, *sum* should be monitored. To this set, we will add *s*, *s1* and *s2* since they may be accessed bu different threads.

The result of the implementation of the components without any management of concurrent accesses is shown in figure 6(a). As we can observe, the two threads are sleeping a random duration and wake up in order to send data to the *sum* component. In the end, it displays 92 as a result of the sum whereas it should have been 90 if there was not any race condition.

In order to implement concurrency access management, ARCS also requires some other components that are not shown: a *Monitor component* that acts as a wrapper component for the actual components that have been marked as to be monitored and a *ConcurrencyManager component* that accesses to the application structure, analyses it and compute the set of components to monitor. The application has been modified to import the *ConcurrencyManager* component (it results in modifying one line in the application description file).

We can verify in figure 6(b) that *Monitors* have been put around the right components and that the race condition was suppressed by such modifications. Indeed, the first lines of the screenshot shows the resulting list of components that should be monitored as planned by our heuristic. The last lines indicate that the resulting sum is 90 which means the race condition has been suppressed.

None of the components of the application has specific code for handling mutual exclusion, therefore the *ConcurrencyManager* seems viable in order to manage concurrency on components that were not specifically designed to handle it. We then have built a first step towards automated concurrency management for components in software applications.

heuristic takes into account all connections in all application sheets (in ARCS terms) whereas the screen capture represents one of those sheets, therefore a subset of the connections between components.

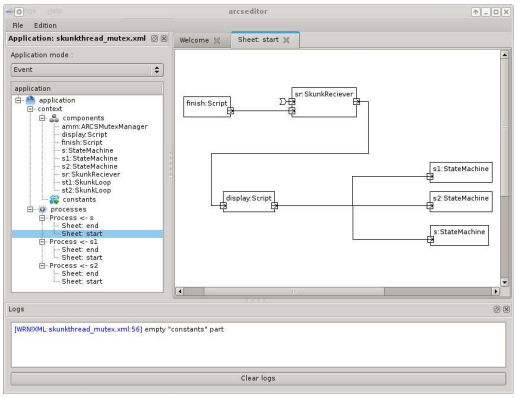


Figure 5: Toy case in development stage using ARCS graphical editor

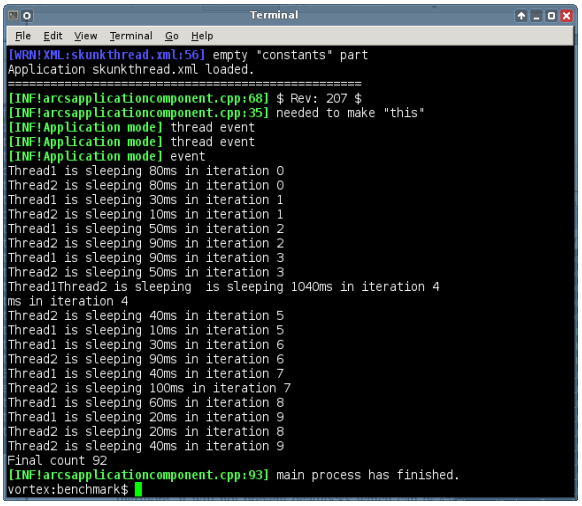
4.2 Real case study: the RAXENV project

In the past, we developed for the RAXENV project [4, 36] a multisensor system that relied on multi-threading capabilities. This project aimed at demonstrating the practical use of an outdoor AR system for environmental sciences and techniques, both in terms of technology development as well as end-user adoption.

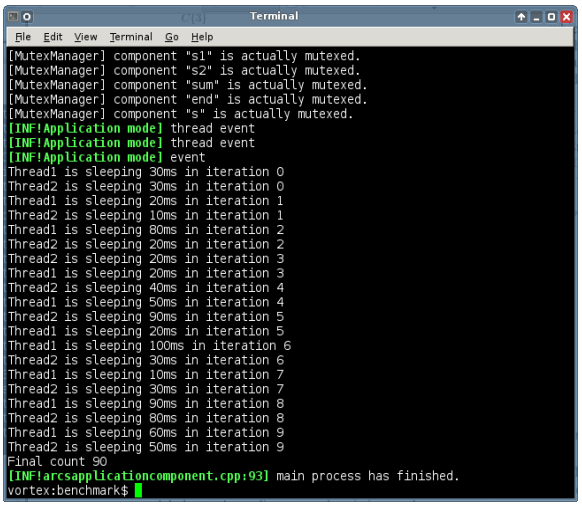
The overall software architecture is depicted in figure 7. The application used data provided by three different sensors: a camera, an inertial unit and a GPS. Data sources were fused to provide a robust localisation of the system in its environment. During the development of such a system, we encountered some difficulties with multi-threading (each sensor provided data in a separate thread) and concurrency management that was implemented by hand. We would like to run our present solution over the RAXENV software in order to assess if concurrency management was implemented at the right place and see if it could eventually be performed in less components than we did and maybe raise some issues we did not see.

The RAXENV software had three nominal states (corresponding to three sheets) where multi-threading capabilities had been enabled. This means our algorithm should then be applied to these three sheets in order to determine which components should be monitored. Moreover, some of the components were composite components (components made of an assembly of components). So we ran our algorithm against two series of tests: one with composite components considered as blackbox components (annotated BB) and another one where composite components are considered as whiteboxes (annotated WB – therefore we consider that we can monitor subcomponents of composite components). The numerical results are gathered in table 1 where we also included our algorithm's result concerning the abstract example and the toy case we presented earlier. The table contains the number of components, the number of edges of the generated graph as well as the final size of the label set, the number of iterations and finally either the size of the component set to monitor or the component set itself. We also represented in Fig. 8 one of the component set (the one annotated MIBB) using the dot utility from graphviz software [2]. It also shows the composite components and components that are to be monitored. One can notice that these first results show that few iterations are in fact needed in order to determine the set of components to be monitored even for datasets containing up to 70 components and 140 edges.

[2]http://www.graphviz.org



(a) Execution of the toy case without concurrency access management



(b) Execution of the toy case with concurrency access management

Figure 6: Toy case implemented and executed using the ARCS framework

In table 2, we regrouped the results obtained from our algorithm, especially the list of components subject to monitoring, for the three datasets RIBB, MIBB and VPBB coming from the actual implementation of the RAXENV software application. For each component, the table states if it should be monitored (*Yes*) or not (*No*) or if it was not used in the dataset (marked with a cross). It exhibits for three components in the list that they should be monitored in some datasets but not in all datasets. It means the very same instance of a component can be subject to monitoring or not depending on the actual state of the application. This is interesting because it is an actual example of a situation where an implementation by hand to make a component thread-safe would lead to a situation where the code written to solve concurrency problems may be useless therefore consuming memory and computation time for nothing.

Interestingly, we compared these results to what was really implemented at that time by developers. Data were only available for the MIBB, RIBB and VPBB dataset and we recorded the differ-

Dataset	Components	Edges	$\|\mathcal{L}\|$	Iterations	Duration	$\mathcal{M}$ / $\|\mathcal{M}\|$
Abstract example	9	12	6	7	3 ms	C, D, E
Toy case	11	15	11	3	3 ms	$End, Sum, s, s1, s2$
Raxenv: reinitialisation (BB) – RIBB	28	34	14	4	4 ms	$ss\_d, widget, iBuf, gm, tc$
Raxenv: manual initialisation (BB) – MIBB	40	62	27	5	4 ms	9
Raxenv: vision predominance (BB) – VPBB	38	53	29	5	4 ms	10
Raxenv: reinitialisation (WB)	35	50	14	4	4 ms	$ss\_d, widget, iBuf, gm, tc$
Raxenv: manual initialisation (WB)	72	145	71	7	20 ms	21
Raxenv: vision predominance (WB)	59	112	44	7	12 ms	19

Table 1: Results gathered from our algorithm ran on different data sets

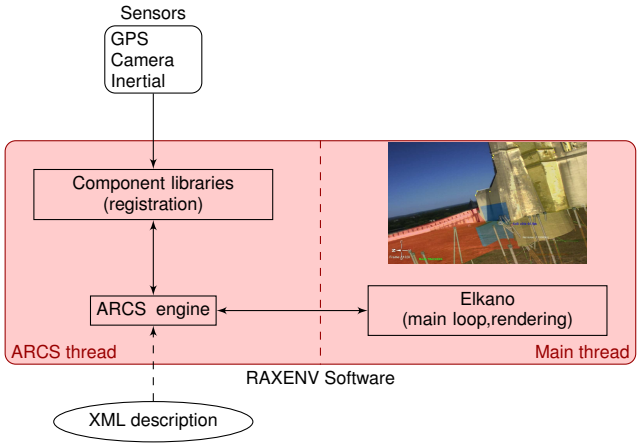


Figure 7: Global interaction between ARCS and RAXENV software modules

Name	In $\mathcal{M}_{\mathrm{RIBB}}$	In $\mathcal{M}_{\mathrm{MIBB}}$	In $\mathcal{M}_{\mathrm{VPBB}}$
ss_d	Yes	Yes	Yes
widget	Yes	Yes	Yes
iBuf	**Yes**	**No**	**Yes**
gm	Yes	Yes	Yes
tc	Yes	Yes	Yes
tmp1	×	Yes	Yes
visP	×	Yes	Yes
isa	×	Yes	×
gm2	×	Yes	×
rinit	**No**	**Yes**	**Yes**
gm3	×	×	Yes
alP	**No**	**No**	**Yes**
chrono	×	×	Yes

Table 2: Recurring components to monitor in RIBB, MIBB and VPBB datasets.

Dataset	RIBB	MIBB	VPBB
Components unprotected	2	5	6
Components overprotected	2	5	2
Differences: total	4	10	8
Differences ratio	14%	25%	21%

Table 3: Comparison with implementation by hand

ences we found out into the implementation and the results provided by our algorithm (see table 3). We counted the number of unprotected components, that is to say components our algorithm marked as subject to be monitored whereas developers did not implemented any thread-safe mechanism, the number of overprotected components (components for which developers have implemented unneeded thread-safe mechanisms) and computed the ratio of the total of such differences to the number of components used in each dataset. One of the most striking result is that we obtained a difference ratio of 25%. This may seem quite high but it does not mean that developers had made mistakes in all situations : one of the prominent problem is that, at that time, mechanisms to monitor composite components were not available whereas it was the kind of components we needed to monitor in this application (It also means that developers did not have, at that time, means to write completely thread safe applications). Another thing to take into consideration is the fact that developers have access to the internals of components and consider them as whiteboxes. Therefore, their analysis may slightly differ than the one from our algorithm because some components may not need to be monitored due to their internal structure. However, overprotected components can be considered as avoidable mistakes.

4.3 Discussion and further work

Concerning our algorithm many things may be subject to discussion. For example, a lot of formal work is still needed in order to assess the complexity of our algorithm as well as proving that it converges in all situations.

Another thing to keep in mind is that our algorithm is only indicating which components are assumed to be monitored. It raises then several other issues. The first one is about the automation of a solution to build monitors at runtime for these components. In some situations, especially if we use the monitor design pattern, we may reach a deadlock situation (when two thread lock components that are needed by each other), therefore a further analysis is needed to detect these situations or we should cast away the monitor paradigm and use another one. However, even if the tool is not fully automated at the execution, the results it produces are interesting for developers since the obtained component set indicates where developers have to focus in order to detect race conditions in their software application.

The last thing we would like to keep in mind is that it may also be interesting to adapt our algorithm in case we may have finer indications of the behaviour of components (i.e. when components are considered as white boxes instead of black boxes). We may then need to write a specific parser and decompose components in several independent vertices in order to build the graph needed by our algorithm.

5. CONCLUSION

We introduced in an actual component based AR framework some automated concurrency management mechanism. To achieve this, we rely on the monitor object design pattern as well as a

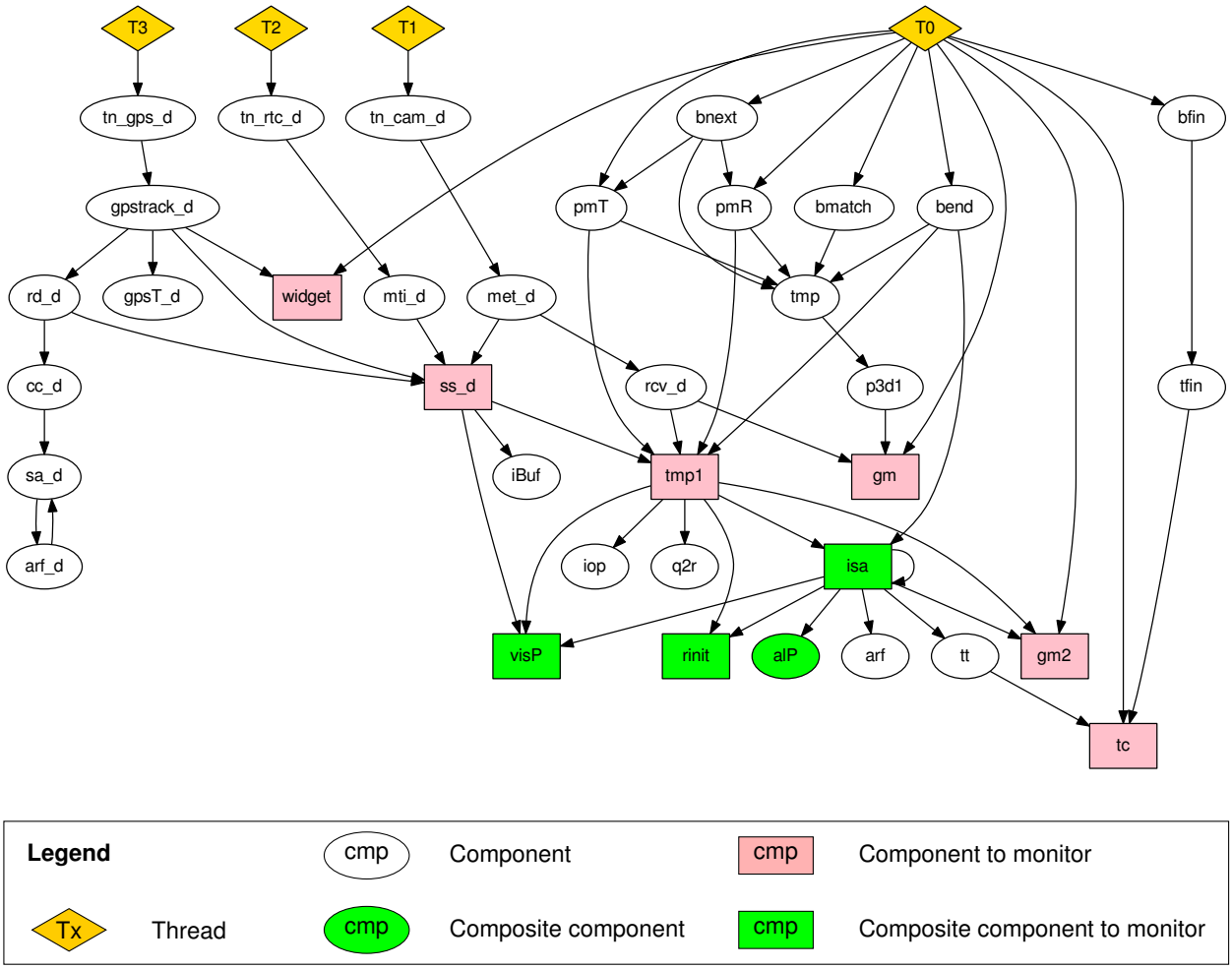


Figure 8: Components configuration in dataset labelled MIBB

heuristic that computesš the components that must be monitored. Applied on a toy case in the ARCS framework, it gave satisfactory result. Therefore we believe that it could be implemented in a larger scale on an actual complex AR application or, more generally, for any component based software using a similar model to the one we presented.

The algorithm was also used on actual data coming from a concrete application and exhibited some differences with the actual implementation made by developers, providing hints about the places where race conditions may occur.

In the future some loose ends associated to concurrency should also be addressed such the problem of deadlock or resource starvation in order to bring out a complete automated concurrency management system that would relieve the developer to determine it and implement it by hand. Therefore, he can focus and spend more time on the functional requirements of software applications.

6. REFERENCES

[1] R. Azuma, B. Hoff, H. N. Iii, and R. Sarfaty. A motion-stabilized outdoor augmented reality system. In *Proceedings of the IEEE Virtual Reality*, VR '99, pages 252–, Washington, DC, USA, 1999. IEEE Computer Society.

[2] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 69–78. ACM, 2006.

[3] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, pages 45–54, octobre 2001.

[4] BRGM. The raxenv project. http://raxenv.brgm.fr/?lang=en.

[5] M. Chouiten, J.-Y. Didier, and M. Mallem. Component-based middleware for distributed augmented reality applications. In *Proceedings of the 5th International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2011)*, page 3, July 1-3 2011.

[6] J.-Y. Didier, S. Otmane, and M. Mallem. A component model for augmented/mixed reality applications with reconfigurable data-flow. In *8th International Conference on Virtual Reality (VRIC 2006)*, pages 243–252, Laval (France), 26-28 avril 2006.

[7] J.-Y. Didier, S. Otmane, and M. Mallem. Arcs : Une architecture logicielle reconfigurable pour la conception des

applications de réalité augmentée. *Technique et Science Informatiques (TSI), Réalité Virtuelle - RéalitéAugmentée*, 28(6-7):891–919, Juin-septembre 2009. Numéro spécial.

[8] R. Dörner, C. Geiger, M. Haller, and V. Paelke. Authoring mixed reality. a component and framework-based approach. In *First International Workshop on Entertainment Computing (IWEC 2002)*, pages 405–413, Makuhari, Chiba, Japon, 14-17 mai 2002.

[9] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *ACM SIGPLAN Notices*, volume 42, pages 291–296. ACM, 2007.

[10] C. Endres, A. Butz, and A. MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems Journal*, 1(1):41–80, janvier–mars 2005.

[11] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[12] Y. Eytani, E. Farchi, and Y. Ben-Asher. Heuristics for finding concurrent bugs. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 8–pp. IEEE, 2003.

[13] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[14] A. Fuhrmann, D. Schmalstieg, and W. Purgathofer. Fast calibration for augmented reality. In *Proceeding of ACM VRST'99*, pages 166–167, Londres, December 1999. ACM.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] R. L. Halpert, C. J. Pickett, and C. Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 353–364. IEEE Computer Society, 2007.

[17] Z. Huang, P. Hui, C. Peylo, and D. Chatzopoulos. Mobile augmented reality survey: A bottom-up approach. *arXiv preprint arXiv:1309.4413*, 2013.

[18] C. E. Hughes, C. B. Stapleton, D. E. Hughes, and E. M. Smith. Mixed reality in education, entertainment, and training. *IEEE Comput. Graph. Appl.*, 25:24–30, November 2005.

[19] IBISC. Arcs: Augmented reality component system. http://arcs.ibisc.univ-evry.fr.

[20] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the avango vr/ar framework - lessons learned. In M. Schumann, editor, *Virtuelle und Erweiterte Realitat : 5. Workshop der GI-Fachgruppe VR/AR*, pages 209–220, Aachen, 2008.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[22] B. Long, P. Strooper, and L. Wildman. A method for verifying concurrent java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 19(3):281–294, 2007.

[23] ObjectManagementGroup. Omg's corba website. http://www.omg.org/corba/.

[24] J. Ohlenburg, I. Herbst, I. Lindt, T. Fröhlich, and W. Broll. The morgan framework: enabling dynamic multi-user ar and vr projects. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST 2004*, pages 166–169, Honk Kong, China, 10-12 novembre 2004.

[25] W. Piekarski and B. Thomas. An object-oriented software architecture for 3d mixed reality applications. In *The Second IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'03)*, pages 247–256, Tokyo, Japan, octobre 2003.

[26] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, and D. Thalmann. Vhd++ development framework: Towards extendible, component based vr/ar simulation engine featuring advanced virtual character technologies. *Computer Graphics International Conference*, 0:96–104, 2003.

[27] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[28] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[29] Qt project. Qt web site. http://qt.digia.com/, url.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[31] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. John Wiley & Sons, 2000.

[32] V. C. Sreedhar, Y. Zhang, and G. R. Gao. A new framework for analysis and optimization of shared memory parallel programs. Technical report, 2005.

[33] N. Sterling. Warlock-a static data race analysis tool. In *USENIx Winter*, pages 97–106, 1993.

[34] C. Szyperski. *Component Software - Beyond Object-Oriented Programming (Second edition)*. Addison-Wesley, Harlow, England, 2002.

[35] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214. ACM, 2007.

[36] I. Zendjebil, F. Ababsa, J.-Y. Didier, and M. Mallem. Large scale localization for mobile outdoor augmented reality applications. In Springer, editor, *International Conference On Computer Vision Theory and Applications*, 2011.

[37] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum lock assignment: A method for exploiting concurrency among critical sections. In *Languages and Compilers for Parallel Computing*, pages 141–155. Springer, 2008.