ERC Starting Grant
Research proposal (Part B2)

Verification of Concurrent Data Structures (VerCors)

1. Motivation

With the increasing demands on efficiency and computing power, the concurrent programming paradigm has become omnipresent. Hardware processor cores are not becoming significantly faster anymore; instead modern computer architectures have multiple cores. Thus, to make full use of the computing power of such machines, applications have to distribute the computing tasks over multiple threads. Multi-core model checking is a typical example: by distributing the model checking tasks over different cores, larger systems can be handled in the same time. In addition, multiple threads are also used to increase responsiveness of applications. For example, in a mobile phone, the main thread should be non-blocking, so that it can always react upon an incoming phone call. Therefore, any other activity will be spawned as a separate thread. Also, in many applications, graphical user interfaces are programmed as separate threads, so that the (often costly) operations on the graphical user interface do not impact the performance of the program itself.

Unfortunately, writing concurrent software is error-prone. Since executions of the threads can be interleaved in many different ways, the total number of executions in a program is quickly too large for a human to grasp. Thus, for a programmer, it is easy to overlook a special situation that might occur, and to introduce an error in the program. As a result, not only the production but also the maintenance of software constitutes a huge cost. Large amounts of money are being spent for coping with erroneous software, and there are not many fields where research has a similar potential for savings.

One of the most fruitful techniques for improving software productivity and software quality is catching large numbers of errors at compile-time, *i.e.*, before running the software. This project does exactly this, and does this where this is most needed, namely for concurrent software. In particular, the project will deliver a collection of verified implementations of concurrent data structures. In addition, a tool set will be made available that allows one to prove formally that other applications that use these data structures are correct, and that will automatically generate specifications to help the prover wherever possible. The verification technique that will be developed will be parameterised over locking policies – including lock-free implementations – and concurrency and synchronisation primitives. This allows to instantiate the verification technique for a different programming language easily. Finally, the verification technique will also be generalised to a distributed setting.

It is expected that after completion of this project, the verification technique and tool set will be usable for advanced developers of concurrent software, and that it will actually help them to increase the quality of their applications. Currently, formal Hoare logic-based specification and verification techniques for sequential applications, such as JML, are reaching a state where they are usable in industry for realistic applications. For concurrent applications, such techniques do not yet exist. However, the recent emergence of separation logic – an extension of Hoare logic – makes reasoning about concurrent programs manageable, because it allows thread-modular verification. Therefore, we believe that now all ingredients are available to develop a verification technique for concurrent applications that is both sound, and practically usable. We expect that after completion of the VerCors project, industrials can use similar tools as exist for JML to reason about concurrent applications.

2. Outline of the Proposal

This proposal is structured as follows. First, an overview of the current state-of-the-art in program verification is given, focusing in particular on verification techniques for concurrent software. Then, we describe the particular application domain of the VerCors project: concurrent data structures. After this, we briefly describe the main objectives of the project, followed by a detailed description for each of the objectives how it will be achieved. Then, a detailed project planning is given, and it is discussed how the different tasks depend on each other and are distributed among the members of the project team. We will also discuss possible collaborations with other research teams, working on related topics. Then, we conclude by emphasising once more the key points of the proposal, and we explain the requested resources for the project.

3. State-of-the-art in Program Verification

Program Logic-Based Verification. The development of efficient techniques and tools for the verification of software remains a grand challenge [19], even though significant progress has been made over the last years.

Model checkers can verify safety and liveness properties of large systems, but are not practically usable if the state space of a system is unbounded, *e.g.*, because infinite state data structures are used, unless the desired property allows to abstract the infinite data into a finite abstract domain.

However, for the verification of a program's functional specification, one typically wishes to show that for any possible input in some unbounded domain, the program behaves in a certain way. To achieve this, techniques based on program logics seem more appropriate. Hoare logic [18] has been the traditional basis for this approach, and several tools exist that successfully apply this to sequential programs written in realistic programming languages (*e.g.*, ESC/Java2 [12], Spec# [3], and Krakatoa [33]).

However, for concurrent programs, where multiple threads compete for shared resources, verification of functional specifications remains a challenge. Owicki and Gries [36] were the first to develop a program logic-based verification method for parallel programs. In their approach, each parallel thread is verified independently, and moreover one has to show that no intermediate assertion can be invalidated by any other thread. Later, Jones proposed the compositional rely-guarantee method [29], where each parallel thread specifies the properties it relies upon and guarantees to the other threads. Even though Jones's method is compositional, in practice it still remained a (too) large effort to verify any parallel program. More recently, Ábrahám adapted the Owicki-Gries method for a subset of Java (without inheritance and subtyping) [1], but also for this method applying it in practice remained a great difficulty.

An alternative is to use separation logic. Separation logic [39] was invented as an extension of Hoare logic that is particularly suited to reason about programs with pointers. In addition to traditional program logics, separation logic allows to specify that properties hold for separate parts of the heap. This means that if one part of the heap is changed, properties that hold for disjoint parts of the heap cannot be invalidated by this change. Thus, reasoning can be local: when verifying a method, one only has to be concerned with that part of the heap that is involved in the execution – the so-called footprint of the method – and the rest of the memory stays unchanged. Moreover, if two references are known to point to different parts of the heap, this implicitly says that these references are not aliases. This implicit absence of aliasing can greatly simplify reasoning, because it allows one not to consider the special case where the two references *are* aliases – a common source of problems. Parkinson extended separation logic to reason about object-oriented programs (in particular for a Java-like language) [38].

Separation Logic to Reason about Concurrent Programs O'Hearn was the first to show that separation logic is also suited to reason about concurrent programs [35]. If two threads operate on separate parts of the heap, they can be safely composed without breaking correctness. However, O'Hearn's approach is too restrictive, because it enforces that two different threads cannot *read* the same variable simultaneously, while this situation occurs frequently in concurrent programs, and is harmless. In fact, disallowing this can seriously affect the efficiency of a concurrent program. In a separate line of work, Boyland [10] had introduced the notion of permissions. In this approach, permissions are used to specify whether a variable is read-only or can be changed. In each program state, variables are associated with permissions. A full permission (1) gives the right to change a variable, while a fractional permission (any value between 0 and 1) only gives the right to read a variable. Bornat *et al.* have shown that the combination with permissions is a natural extension of separation logic [9]. Thus, combining permissions with separation logic for concurrent programs can be used to reason about programs where multiple reads to the same location are allowed. If the total number of permissions to access a certain location is never more than 1, this guarantees that only reads can happen simultaneously. A thread can only write to a location if it has the full permission to do this, and therefore all other threads have permission 0 to access this location, *i.e.*, they cannot even read this location.

Independently of each other, several groups applied this idea to more realistic concurrent programming languages.

- Gotsman *et al.* [15] and Hobor *et al.* [20] showed how permission-based separation logic can be used to reason about languages with single-entry locks, dynamic thread creation, and concurrency primitives that resemble POSIX threads, *i.e.*, a C-like language.
- Haack, Huisman and Hurlin [16,17,26] applied permission-based separation logic to reason about a Java-like language with reentrant locks, garbage collection, inheritance and subclassing, and Java-based concurrency primitives.

- Leino *et al.* [31] developed Chalice: a programming language and verification tool. They do not reason using separation logic; instead, they write explicit access predicates in the specifications, and they translate the specifications into first-order logic, before generating proof obligations. Their language contains single-entry locks, dynamic thread creation and termination.

Typical properties that are verified using these approaches are for example absence of data races or deadlock. However, these approaches are not yet suited to specify and verify functional specifications of applications, describing their precise behaviour. For libraries, one needs to have such functional behaviour specifications, that enable easy reuse of implementations, and allows proving correctness of complete applications. Moreover, the verification techniques are always tailored to a specific language with precise concurrency and synchronisation primitives. Within the VerCors project, the verification technique will be parameterised over these concurrency and synchronisation primitives, so that instantiations for new languages can be defined easily. In addition, we will separate the functional specification from the concurrency-specific part, so that the functional specification can be reused for implementations that differ in the locking policy, and in particular, we will enable support to reason about programs with (benign) data races. The logics that currently exist do not provide such flexibility.

4. Application Domain: Concurrent Data Structures

To make full use of the power of multi-core systems, programs use efficient concurrent data structures, where multiple threads can store and retrieve data simultaneously. Typical examples, among many others, are large central data bases that can be accessed from different sites, concurrent hash tables used for (multi-core) model checking, and request buffers for thread pools.

Many different implementations of such concurrent data structures are possible, see *e.g.*, [34]. A simple approach is to protect all accesses by a lock, so that always at most one process at the time can be manipulating the data structure, but of course, this is not very efficient. In particular, if the data structure is large, and accessed frequently by different threads, such a lock-based approach can give a large overhead, and seriously impact system performance. On the other side of the spectrum, one can have data structures that are not protected by any locks at all. Algorithms on such data structures are called *lock-free* or *wait-free*. In such data structures, if the reading and writing of data is atomic, the data that is stored cannot be corrupted – but occasionally it might be overwritten by a competing write, or read twice. Notice that if the underlying memory model does not ensure that values are written atomically, for example, because it manipulates memory blocks that can contain several values, then data corruptions still might occur.

Further, any variation between those two extremes can be imagined. For example: blocks of data can be protected by a single lock; only part of the data structure is protected by a lock; or special reader-writer locks allow multiple simultaneous reads, but only single writes to a location.

However, independent of the policy that is chosen to control access to the data structure, it should behave as expected. For example, if the data stored is sorted, it should remain sorted, and if it is guaranteed that data cannot get lost, then eventually all data that has been put in the data structure can be retrieved from it.

For sequential data structures, Hoare logic-based approaches, such as the different JML tools, are very good at both specifying and verifying such properties (see *e.g.*, [21,22]). However, for concurrent data structures, one needs to extend the specifications and verifications to capture the concurrency aspects. In fact, the specification can be split into two parts: the first part describes the behaviour of the data structure; the second part is related with the concurrency aspects. This first part should correspond to the specification of a sequential implementation of the data structure; the second part differs for each different locking policy. If the correctness of an application only depends on the first part of the specification, this allows changing the implementation, and to choose the one with the most efficient locking policy.

5. Objectives

Goal of the VerCors project is to develop specification and verification techniques for concurrent data structures, written in a wide range of programming languages, using different concurrency paradigms. Particular points of interest will be:

- *Different locking policies.* Specifications will be separated into two parts: the first describes the abstract behaviour of the data structure; the second is related with the concurrency aspects and the particular locking policy of the implementation. This means that a single specification can be verified for several implementations that vary in the degree of locking. Moreover, any application that can be verified on the basis of the abstract behaviour only, is correct for any implementation that varies only in the locking policy.

- *Lock-free data structures*. Since they do not constrain concurrency, lock-free data structures can positively influence the performance of an application, but they also require special care, because data might be corrupted, duplicated or removed. The specifications will make explicit what guarantees can be provided for those, or under which additional conditions a lock-free data structure functions correctly.
- *Variations in concurrency and synchronisation techniques*. Concurrent programming languages all use different primitives to model concurrency and synchronisation. Verification techniques for concurrent programs that have been developed so far have studied different languages (*e.g.* C and Java), but they have mainly concentrated on simple locks (possibly reentrant) as a synchronisation primitive. However, in actual applications, a much larger variety of synchronisation techniques is used: reader-writer locks, latches, barriers, futures etc. Which technique is chosen depends on the application at hand. A classification of synchronisation techniques will be established, and for each class of techniques, appropriate verification techniques will be studied. In addition, the verification logic will be parameterised with concurrency and synchronisation primitives, so that it can be easily instantiated for different programming languages.
- *Expressiveness and readability of specification language*. For sequential programs, expressive and readable specification languages such as JML exist. For concurrent programs, separation logic can be used to write specifications. This is expressive, but specifications are not always easy to read. A specification language will be developed that is a combination of readability à la JML and expressiveness à la separation logic.
- *Generation of permission annotations.* A large burden for program verification is that it is time-consuming and error-prone to write specifications. In particular, to ensure that verification succeeds, often many trivial details have to be specified. For sequential programs, experiments have been conducted to generate part of the specifications, based on the actual implementation. For concurrent programs, a similar approach will be used to generate for example specifications stating that some code fragment only reads a variable, while other code fragments update it.
- *Distributed data structures.* In modern architectures, data structures often are not only used concurrently, but also distributed over different nodes in the system. This requires that the distributed parts of the data structure exchange information, and ensure some global consistency properties. The verification logic will be extended to this setting.
- *Tool support*. All techniques that will be developed within the project will be supported by appropriate tools, making use of modern IDE (such as Eclipse) to provide an attractive and effective user interface.

6. Methodology

This section describes how the particular points of interest mentioned above will be addressed within the project. Different tasks are identified, and for each task it is described how it will be handled. Figure 3 on page 11 shows dependencies between tasks and how the different tasks are divided into subtasks.

Permission-based Separation Logic Basis of the work conducted in the VerCors project will be the permission-based separation logic developed by Haack, Huisman and Hurlin [16,17,26]. This logic allows to reason about a Java-like language with dynamic thread creation and termination (*i.e.*, thread termination can be detected by joining a thread – this allows to retrieve the permissions that were released by the terminated thread) and reentrant locks. In addition, the logic contains rules to reason about wait and notify and thus covers the essential ingredients of concurrency in Java. Soundness of this logic has been proven (in full detail) with pen and paper.

The logic contains the standard operators from separation logic, *i.e.*, the points-to predicate that allows to capture the pointer structure of the data; the separating conjunction (the star operator) that specifies that two predicates are validated by disjoint parts of the heap; and the magic wand (or separating implication) that specifies conditional properties about pointer structures. In addition each pointer is associated with a permission that specifies whether the owner of the pointer can either read or update the location that is being pointed to. Permissions are a value in the domain [0, 1]. These permissions make the logic suitable to reason about concurrent programs in a thread-modular way. Soundness of the verification system ensures a global correctness criterion, namely that the total number of permissions in the system pointing to a particular location never exceeds 1. If a thread has a pointer with permission 1, this means that this thread is the only thread that can access this location, and thus the location can safely be modified. If a thread only has a fractional permission, *i.e.*, a permission that is less than 1, this means that other threads might also be accessing the same location simultaneously, and thus the value stored there can only be read, and not updated. Notice that a program can only be verified with this logic, if it does not contain any data races (and of course, if it respects the method specifications). This is a desirable property for a (Java) program, because

the Java Memory Model prescribes that data race free programs behave sequentially consistent. In contrast, if a program has data races, its behaviours can be unpredictable (or more precisely, the allowed behaviours have to respect the so-called happens-before order, but they are difficult to grasp for humans, because program optimisations are possible).

Tool Support, Specification Language and Automated Verification An important first step to make the verification technique usable is to develop appropriate tool support for it. Experience with small examples have revealed that it is error-prone to write specifications in this logic by hand, and thus tool support is needed in an early stage of the project. For the tool to be usable, it should provide automated support for reasoning.

As mentioned above, several tools exist that allow to reason about sequential Java, using JML as specification language (*e.g.*, ESC/Java [12], Key [5]). Currently, several JML tool builders are joining efforts to develop one front-end for JML in Eclipse, that allows to use different JML-related tools in a seamless way (for more information, see http://www.jmlspecs.org). Since we wish to provide support for a complete language, it is a natural choice to connect to such existing tools, and to extend them for our verification method. This requires developing a specification language that combines permission-based separation logic with JML. Basically, this would mean that separation logic operators (the points-to predicate, separating conjunction and implication) are added to the JML expression grammar. Moreover, separation logic (including the version of permission-based separation logic that we developed) uses abstract predicates to abstract in a mathematical way over data structures. JML provides model methods for the same goal. Integrating JML and separation logic would thus also require the unification of abstract predicates and model methods. An advantage of this approach is that it will increase the readability of specifications. The JML specification language is developed to be easily readable for an average Java programmer, while specifications in permission-based separation logic are complex and difficult. Thus, they will benefit from the possibility to use more readable specification constructs from JML.

With respect to concurrent applications, Leino, Müller and Smans developed Chalice [31], but this tool uses a simplified language as input language. Leino *et al.* do not generate proof obligations in separation logic; instead, they translate all specifications into first-order logic predicates, before generating proof obligations – using the Boogie methodology. However, because of the semantics of the separation conjunction, this translation can result in nested quantifications, which are an impediment to automation [6]. Consequently, existing static assertion checkers for fragments of separation logic proceed differently. We are aware of two lines of work on static assertion checking for separation logic: (*i*) the Smallfoot checker [6], which is based on especially tailored proof-theoretical decision procedures for logical entailment of a restricted subset of separation logic, and (*ii*) the use of a linear logic constraint solver to solve the proof obligations that arise from the separation logic assertions [28,8], so that a larger subset of the logic can be handled automatically.

Within the tool set, we plan to use the best of both approaches. Initially, the Smallfoot and linear logic solvers, adapted to our particular flavour of separation logic, will be used in parallel to solve the generated separation logic proof obligations. In addition, one or more SMT solvers, as used in JML-verification-based tools, will be used to solve the first-order logic proof obligations that arise for example from the requirement that an array is indexed within its bounds. Using the tool set on different applications (in particular, for the specification and verification of concurrent data structures as described below) will teach us about the frequency of certain proof obligations in practice. It will also teach us which approach is better on which kind of property, and what are the kinds of properties that are not covered at all. Eventually, we will then improve the more promising of the two approaches, so that it captures our logic completely.

Another issue that needs to be solved to provide fully automated tool support is the need to reason about the absence of aliasing because of reentrant locks. In our verification method, each lock is associated with a resource invariant. When a thread acquires a lock, it obtains the resources as specified by the resource invariant. However, in the case of reentrant locks, these resources only should be obtained upon the initial acquirement – otherwise resources are duplicated, which causes the system to be unsound. Thus, a thread can only safely obtain a resource invariant for a lock l if l is not held by the thread yet – and thus it has to be proven that no alias of l is in the set of all the locks held by the thread.

In earlier work, we have shown how this problem could be handled by combining separation logic assertions with ownership types. We will explore how this can be done in a more systematic way. We believe that if ownership relations are systematically specified, many of the proof obligations about absence of aliasing can be discharged automatically. An additional advantage of this approach is that verification of type-based techniques is lightweight. Moreover, the new annotation syntax proposed by JSR-308 [30], which is part of Java 7, provides a great opportunity to put such type systems onto Java.

Finally, all theoretical developments that are described below will be integrated in the tool set and validated on realistic example applications.

Concurrent Data Structures with Different Locking Policies For efficient multi-core implementations, the data structures that are used should be concurrent. This allows several threads to access data stored in the data structure simultaneously. To prevent data races, locks are used to ensure that different threads cannot write or read/write a location simultaneously. However, the choice of different locking policies can seriously influence efficiency of the application. If large chunks of data are protected by a single lock, then threads are expected to wait for the release of a lock frequently. In contrast, if the locking policy is fine-grained and each lock protects only a few memory locations, threads might not have to wait for lock releases frequently. However, if the data structure is large, the storage overhead for all the locks can also have a significant negative impact on the performance of the application. An example application where such issues are important is multi-core model checking. Concurrent hash tables are used to efficiently store states that have been explored, and experiments show that different locking policies can have a significant impact on the running time of the model checker. This will be used as a motivating example throughout the VerCors project.

However, the behaviour of the concurrent data structure should be independent of the particular locking policy that is used. As a starting point, we will specify and verify implementations of concurrent data structures [34] with a fine-grained locking policy, *e.g.*, where each lock protects a single entry in the data structure. The specifications will specify the functional behaviour of the data structure, *i.e.*, data that is stored in the data structure can eventually be retrieved from it, and operations that manipulate the data structure, such as concatenation and sorting, behave as expected. The specifications should be such that they can be used to prove correctness of an application that uses the data structure.

As a next step, the implementation will be changed by varying the locking policy. We will study which parts of the specification need to be adapted – in particular, the resource invariants that specify which resources are obtained upon acquiring a lock will have to reflect the different locking policy – and which parts can remain unchanged. The verification will also be adapted to reflect the changes in the specification and implementation. Eventually, this should lead to a systematic approach to adapt specifications and verifications to different locking policies. In particular, we foresee that the specification can be split into two parts: the first part describes the behaviour of the data structure in an abstract way. This part of the specification is common to all different implementations. The second part of the specification describes the particular locking policy that is being used, and will vary for each locking policy. Proofs that only depend on the abstract specification can be reused for any implementation. Proofs that depend on the locking policy will differ per implementation, but the changes in the locking policy indicate how the proofs should be adapted.

Within the project, several different data structures, such as lists, hash tables and buffers will be studied. For these data structures, an abstract specification will be developed. Case studies will ensure that the specifications are sufficient to prove correctness of applications that use the data structures, and that implementations with different locking policies can be plugged in – and still result in a correct application.

Variations in Concurrency and Synchronisation Techniques The verification logics that currently exist to reason about concurrent applications are typically developed for one particular language, considering the particular concurrency and synchronisation primitives of that language. For example, our permission-based separation logic has been developed for Java and Java's reentrant locks.

However, even within the same programming language, many different synchronisation techniques are used by applications. For example, the Java API contains implementations of barriers, latches, read-write locks, and futures. Barriers enable several threads to synchronise regularly on reaching a particular state, before progressing; latches are used to keep track of whether the system is satisfying a particular condition, for example to check whether all threads have finished their initialisation phase; read-write locks allow multiple threads to read data simultaneously; and futures are used to wait for the result of a computation that is done by a different thread. In Java, these synchronisation techniques are defined in terms of the core synchronisation primitives. To extend the verification logic to such techniques, the APIs implementing these techniques have to be formally specified (and verified) in such a way that they can be used as any other synchronisation technique in verifications. In our existing work, this approach has already been applied to specify Java's wait and notify mechanism [26].

Commonalities in the specifications of the synchronisation techniques will allow us to develop a general classification of synchronisation techniques. The adaptation of the verification techniques will be based on this classification, *i.e.*, synchronisation techniques that fall in the same class will give rise to similar

techniques. Moreover, the automated verification procedures, mentioned above, will be fine-tuned to these different synchronisation techniques, so that they can be reasoned about just as efficiently as for locks.
In a second step, our verification techniques will be adapted for other concurrent programming languages. Currently, our version of permission-based separation logic is tailored to Java, but it is important that we can also reason about for example C programs, because many applications in which efficiency is important are implemented in this language – because of the possibility to write code that is closely linked with the hardware. This will require us to adapt the logic to the concurrency and synchronisation primitives of the language that we wish to verify. We expect that the classification of synchronisation techniques will allow us to make this adaption in a relatively easy and efficient way.
Eventually, we plan to identify a core logic that is parameterised with concurrency primitives and synchronisation techniques. A verification logic for a particular programming language can then be defined by instantiating this core logic with the concurrency and synchronisation primitives of the language at hand.

Generation of Permission Annotations An important problem for the application of formal methods in real-life applications is the burden of specification writing. Even though programmers are usually used to document their programs extensively – in particular when they work in a team – it is still a large step to transform these comments into a formal specification that is amenable to tool support. Part of this is due to the fact that it can be difficult to express the program behaviour as a formal property. This remains a difficult task, even though it can be lightened by the development of appropriate specification languages, such as the combination of permission-based separation logic with JML. However, it is also due to the fact that many ``obvious ´´ facts have to be written explicitly and formally for the verification to succeed.
To overcome this problem, a technique will be developed to generate a large part of these obvious specifications automatically. In earlier work, the PI has already studied how weakest precondition calculi can be used to generate annotations to prevent run-time exceptions, caused by null pointer dereferencing and array out of bounds indexing [4]. In addition, more heuristic-based methods have been used to generate obvious annotations in a JML-like language [14], or to generate class invariants [13] based on program executions. However, these methods all focus on the behaviour of sequential programs, and do not consider the concurrent aspects of an application.
Within the VerCors project, we plan to generate annotations that focus on the concurrent aspects of an application. In particular, it will be analysed when a variable is written (and thus a thread requires full permission to access the variable) and when it is read (and thus access to the variable can be shared by other threads). This analysis can be done locally; verification of the generated annotations afterwards will ensure that if a thread requires full access to a certain variable at a certain point, no access via other references to the same location can happen simultaneously.
In addition, alias analysis can be performed to improve the precision of the generated annotations. If several aliases to a certain memory location exist, this can give an indication that permissions should be split. Also, we will generate possible monitor invariants by deriving from the program code what state is actually protected by a lock, *and* which permissions are obtained by acquiring the lock.
For the generation of annotations, we plan to use an approach that is based on a combination of weakest precondition calculus and static analysis. The first technique can tell us under which conditions we can guarantee that a method will terminate normally, the second technique will provide us with additional assumptions that we can make on the program and data structure, which can help us to improve the precision of our results.

Lock-Free Data Structures A special class of concurrent data structures are the so-called lock-free data structures, where no locks are used at all. This of course results in an efficient implementation, but has the drawback that data can sometimes be lost. If two values are written to the same location simultaneously, then one of the two will be lost – provided of course, that the writes are atomic, otherwise the data might simply be corrupted. Moreover, in case of a Java implementation, in the presence of race conditions, the Java Memory Model [32] specifies that allowed behaviours can become rather unexpected for humans.
Nevertheless, the expected increase in performance is sometimes more important than the drawbacks. In particular, for some applications, the (incidental) loss of a value might affect performance, but does not influence correctness of the application. For example, if a lock-free hash table is used to store the analyzed states in a multi-core model checker, then for many algorithms, the loss of a state value only means that the state will be visited twice, but it will not change the outcome of the algorithm [2]. However, it is important to know exactly where value losses might occur, or what the possible impacts of a race condition are on the overall behaviour of the application. The effects of a race conditions can be considered *locally*, *i.e.*, a race condition can only impact variables and program code that is (indirectly) related with it, but not a completely

unconnected piece of code. However, so far a completely formal proof of this modularity claim does not exist. Further, we believe that the data race freeness guarantee of the Java Memory Model can be extended to benign race conditions: if a race condition is benign, *i.e.*, two threads simultaneously write the *same* (atomic) value, then the behaviour of the program is still sequentially consistent.
Therefore, it is important to be able to specify formally which guarantees can be given for a lock-free implementation of a data structure. As a starting point, we will take the specifications for a concurrent data structure with locking policy – in particular the abstract part of it, that describes the general behaviour that is common to all implementations, independent of the locking policy – and we will see how much of this specification still holds for the lock-free implementation. Goal is to develop a systematic way to reason about these cases, and to specify formally which guarantees can be given for lock-free data structures.
Further, we will also study how benign data races can be handled with our verification technique and which changes are necessary to the verification rules. Currently, the logic is set up in such a way that if a program can be verified with the logic, then it cannot reach configurations in which a data race can occur, *i.e.*, the program is free of data races. This guarantee follows from the permission approach: if a thread wishes to write to a variable, it needs to have the *full* permission to access the variable – thus other threads cannot access the variable simultaneously. A program can only be verified if the total number of permissions to access a variable never exceeds 1. To handle benign data races, this policy will have to be released at specific points – just as in information flow theory some data can be explicitly declassified, in the verification logic specific variables can be marked as permission-free. The user will have to specify these permission-free variables explicitly. Two flavours of permission-free variables can exist: those allowing benign data races only, and those allowing arbitrary data races. A program is verified if all permissions to access a variable never exceed 1, except for these permission-free variables. For an arbitrary permission-free variable, no check is applied at all; for a benign permission-free variable, verification of the program ensures that always the same value is written when the data race occurs. To understand what behaviours a verified program allows, the Java Memory Model will have to be studied again, and in particular, the claims that the effects of race conditions are local and that programs with benign data races are still sequentially consistent will have to be proven formally.

Distributed Data Structures Finally, the verification logic will be extended for an environment that is both concurrent and distributed. In particular, in a distributed environment, a data structure can be distributed over different sites. This means that not only the concurrent accesses within the data structure have to be handled correctly, but also consistency within the distributed data structure has to be maintained. Special protocols can be used to ensure this, just as cache coherence protocols are used to ensure that values in the cache correspond with values in the main memory, so that the use of the cache is completely transparent, *i.e.*, only the efficiency of the program is affected by the use of the cache, not its behaviour. For distributed data structures, a similar requirement can be stated: the behaviour of a program should not be affected by the fact that the data structure is distributed over different states.
When adapting the verification logic to distributed data structures, two issues have to be considered. First of all, the logic has to contain rules to reason about data that is moved from one site in the network to another. Typical techniques to develop distributed applications are for example Remote Method Invocation (RMI) and the Message Passing Interface (MPI). Standard APIs are available for these techniques. These libraries will have to be annotated. Further, the underlying heap model has to consider that data can be distributed over different sites, and might be duplicated on different sites. This is an extension of the standard heap model. The rules in the logic will be adapted and proven sound w.r.t. this extended distributed heap model. Moreover, the APIs that are used to develop distributed programs, such as RMI and CORBA, will be proven sound w.r.t. this extended version of the logic.
Second, the protocol that is used to keep the data structure consistent has to be proven correct. Some work in this direction already exists for cache consistency, see *e.g.*, [37], but it remains a challenge to prove this consistency within the verification logic. Thus, the methods that implement the protocol have to be annotated appropriately, and the overall consistency criteria have to be specified formally within the logic. Then it has to be shown that the method specifications are sufficient to guarantee the overall consistency criteria, and, of course, it has to be proven that the implementation of those methods respect their specification.

Case studies As mentioned above, the techniques and tool set will be validated on several case studies. Two larger case studies are planned in the 3rd and 5th year of the project. The first case study will focus on the use of concurrent hash tables in the implementation of a multi-core model checker. The second case study will extend this to the case where the hash table is also distributed.

Year 1: Setting up the Framework	
Who	PI, PhD student 1, post doc 1, scientific programmer
What	Implementation of verification tool, implementing permission-based separation logic for a Java-like language, as presented in Hurlin's thesis [26], with dynamic thread creation and termination and reentrant locks. Specification and verification, using verification tool, of concurrent lock-based implementations of data structures like hash tables, linked lists and vectors. Integration of separation logic constructs into JML specification language, to improve readability of specifications, and integration of language in tool. Development of automated verification techniques for proof obligations generated by tool.
Year 2: Development of Basic Theory	
Who	PI, PhD student 1, PhD student 2, post doc 1, scientific programmer
What	Develop technique to adapt specifications and verification of lock-based concurrent data structures to concurrent data structures with more flexible locking policies. Adapt verification techniques for different synchronisation techniques (reader-writer locks, barriers, latches, futures etc.) by classifying them in different categories and implement this in tool. Develop ownership-based type system to show absence of aliasing Continue development of automated verification techniques for proof obligations generated by tool.
Year 3: Automation and Generalisation	
Who	PI, PhD student 1, PhD student 2, post doc 1, post doc 2, scientific programmer
What	Develop specification technique that allows one to abstract away from specific locking policy that is used by implementation. Generalize verification approach to other concurrent languages, based on classification of synchronisation techniques, by parametrising the logic with concurrency and synchronisation primitives. Continue development of automated verification techniques for proof obligations generated by tool. Develop techniques to automatically generate permission annotations, by deriving from the code when a variable is read, respectively written, and implement this in tool. Case study: hash tables for multi-core model checker, to show validity of approach to concurrent data structures.
Year 4: Advanced Theory	
Who	PI, PhD student 1, PhD student 2, post doc 2, scientific programmer
What	Develop technique to automatically adapt proofs for one implementation to new implementations with different locking policies. Write up of thesis PhD student 1. Study how specifications and verifications of concurrent data structures should be adapted to lock-free data structures. Extend techniques to generate permission annotations by implementing heuristics to derive monitor invariants (i.e., deriving which resources are protected by a lock), implementing this in a tool, and validating it on several case studies. Adapt verification techniques to a distributed context.
Year 5: Consolidation	
Who	PI, PhD student 2, post doc 2, scientific programmer
What	Adapt verification techniques to benign data races, as might occur in lock-free data structures Write up of thesis PhD student 2 Adapt results on concurrent data structures to a distributed setting. Case study: distributed hash tables, to show validity of approach on distributed concurrent data structures Ensure tool is mature enough to be usable for external users.

Figure 1: Project Planning

7. Project Planning

Even though it is difficult to foresee all the outcomes of the project, Figure 1 gives a rough planning. For each year we list the people that are involved in the project, and which scientific problems will be studied. Results will be regularly published in well-established conferences and journals in the area.

Tasks	PhD 1	PhD 2	Post doc1	Post doc2	Sc. Prog.
Tool support	**1-4**	**2-5**	**1-3**	**3-5**	**1-5**
Integration of separation logic and JML			**1**		**1**
Automated verification techniques for proof obligations			**1-3**		
Concurrent data structures with different locking policies	**1-4**		**1-3**		
Variations in concurrency and synchronisation techniques		**2,3**		**3**	
Generation of permission annotations				**3,4**	**3,4**
Lock-free data structures		**4,5**		**4,5**	
Distributed data structures				**4,5**	
Case studies	**3**	**4,5**	**3**	**4,5**	

Figure 2: Distribution of Tasks

8. Distribution and Dependencies of Tasks

The distribution of work is summarised in Figure 2. The PI is not explicitly mentioned, as she will be involved in the initiation and supervision of all these tasks. Dark coloured fields mean that a person is involved in the task; a light coloured field means that a person will contribute to discussions and provide input where possible. The numbers indicate in which years of the project the task is expected to be done (thus a task can cover several problems listed in the project planning in Figure 1).

Figure 3 (on the next page) shows how the tasks are divided into the subtasks mentioned in Figure 1's project planning. In addition it shows how the different (sub)tasks depend on each other. All tasks interact with tool support, and tool support continues during the whole duration of the project; therefore this is not depicted explicitly in the figure. Also, the case studies will benefit from and test developments such as annotation generation and automatic verification of proof obligations (*i.e.*, case studies will be done using the latest version of the tool set), but for clarity these dependencies are not listed explicitly here.

9. Collaborations

The project will be conducted completely within the Formal Methods and Tools group at the University of Twente, Netherlands. However, the good international contacts of the PI will give rise to useful collaborations with other teams working on topics such as program verification (of concurrent applications), and separation logic. Among others, the PI has been collaborating in European projects with

- the *Programming Methodology* group, led by Prof. Peter Müller, at ETH Zurich, Switzerland;
- the *Software Engineering using Formal Methods* group, led by Prof. Reiner Hähnle, at Chalmers University, Sweden, that is involved in the development of the KeY-tool; and
- the *KindSoftware: Software Engineering with Applied Formal Methods* group, led by Dr. Joseph Kiniry, at UCD, Ireland, that is involved in the development of ESC/Java.

Because of the European collaborations, the PI has also been in regular contact with Dr. Rustan Leino, from Microsoft Research, Redmond, US, who has contributed to the development of several program verification tools, including ESC/Java and Chalice. Rustan Leino was a member of the scientific advisory board of the Mobius project, and actively took part in the discussions on the development of the Mobius IVE, in which also the PI contributed. Moreover, the PI still has scientific contacts with former colleagues in the Digital Security group at the University of Nijmegen, notably Dr. Erik Poll, the Marelle and Indus team at INRIA Sophia Antipolis, and IMDEA Software in Madrid, Spain.

It is planned that both PhD students will visit another research group abroad during three months. In the experience of the PI this is an important step in the education of a young researcher, since it fosters new collaborations, and broadens the view on the conducted research.

10. Summary and Conclusions

Goal of the VerCors project is to develop specification and verification techniques for concurrent data structures, written in a wide range of programming languages, using different concurrency paradigms. This will result in an efficient static verification technique for concurrent applications that will significantly improve software quality and reduce maintenance costs. Key project results include:

1. A large collection of fully specified and verified implementations of concurrent data structures.
2. A specification language that is readable, expressive, and allows one to concisely specify concurrency-related behaviour of an application.

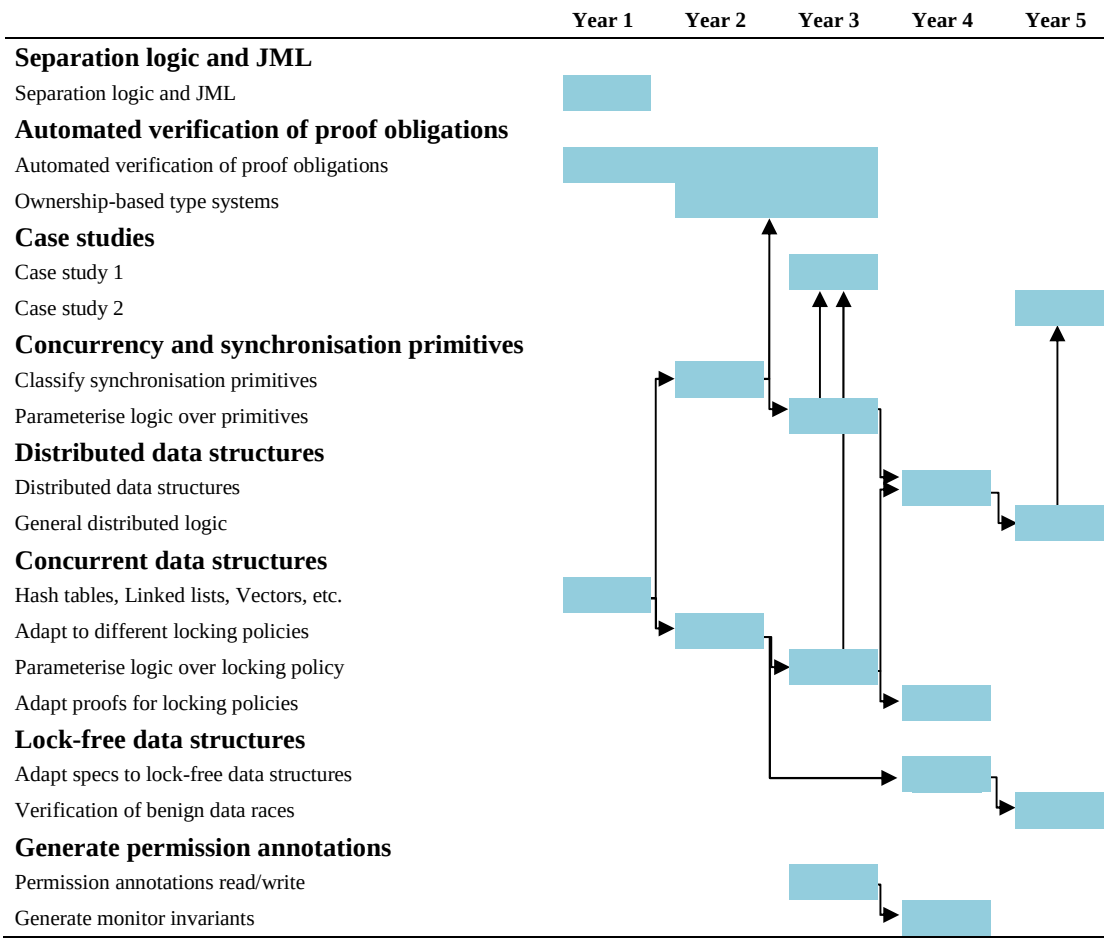

Figure 3: Dependencies between Subtasks

3. A verification technique that is parametric over the locking policy that is used in a particular implementation. Specifications are separated into two parts: the first part describes the abstract behaviour of an application; the second part specifies the locking-specific requirements. Changing the locking policies only requires the locking-specific part to be reverified, the abstract behaviour specification remains valid.
4. A verification technique that allows one to specify that certain data races are intended, and can be reasoned about. Validity of this approach will be shown on case studies with benign data races and on case studies where the loss of data does not affect the correct behaviour of the application – only its efficiency.
5. A verification technique that is parametric over concurrency and synchronisation techniques, and therefore easily can be adapted for a different concurrent programming language.
6. An extension of the verification technique for a distributed setting, where data is distributed over different sites, but has to respect global consistency conditions.
7. A technique to generate annotations that can be directly inferred from the code, including monitor invariants, *i.e.*, including the specification what state is protected by a lock. The generated annotations are correct, but might not be sufficient. However, they reduce the annotation burden of the specifier, allowing him to concentrate on the essential parts of the specification writing process.
8. Full tool support for the developed verification techniques, integrated in an IDE such as Eclipse, to make the tool accessible to programmers.
9. Automated verification techniques to prove (or refute) the separation logic proof obligations that are generated by the tool.
10. A collection of case studies that demonstrate validity of the approach.

The project focuses on concurrent data structures as particular application domain. The main reasons for this are:

- concurrent data structures are widely used;
- their correctness is essential for the correctness of many concurrent applications; and

- they have many interesting functional properties that have to be verified.

The omnipresence of concurrent applications makes the results of the VerCors project highly relevant with a huge potential for cost savings in application development and maintenance. Recent theoretical advances, such as the development of separation logic and JML, have led to new insights, which make verification of concurrent applications feasible in practice.
Multi-core model checking algorithms are typical example applications where concurrent data structures are used; their implementations will be used as motivating examples throughout the project. However, the verification approach that will be developed is also applicable in a wider context, for any concurrent (and later also) distributed application.
The PI has the proven experience, knowledge, and leadership to head this project:

- She already has gained a large experience with the use of permission-based separation logic to reason about concurrent applications written in Java (notably the work described in Hurlin's PhD thesis [26] was done under her supervision), and this logic will be the basis for the work in the VerCors project.
- She knows about the Java Memory Model and has worked on formalising and proving data race freeness, in collaboration with Gustavo Petri [24,25], thus she has the right knowledge to extend the verification logic to (benign) data races and to apply it on lock-free algorithms.
- She has contributed to the development of several tools for Java program verification, notably the LOOP tool (for Logic of Object-Oriented Programming) [7,27] and JACK (Java Applet Correctness Kit) [4].
- She combines a good theoretical knowledge with an eye for practical applicability. Theory is not just developed for the sake of theory, but she also makes sure that it can be applied on real-life examples, as demonstrated in several large case studies [23,21,11,22].
- She has good contacts with researchers working on program verification, the JML specification language, and separation logic.
- She has good managerial qualities, and can successfully lead and motivate a group of young researchers.

11. Resources (Including Project Costs)

The VerCors project will run for five years. Figure 4 shows the budget requested for the project. The PI will be involved in the project for 50 % of her time during the full duration of the project.
Further, funding is asked for two PhD students (PhD projects normally run for four years in the Netherlands), two post docs, and a scientific programmer.
Both PhD students are expected to start in the beginning of the project (year 1 and year 2). The first PhD student will work on specification and verification of concurrent data structures with different locking policies. He or she will study how specifications can be written in such a way that they can be reused for different implementations. Expected outcome is a collection of specifications of commonly used concurrent data structures, with verifications of several implementations. The second PhD student will work on generalising the verification techniques to different classes of concurrency and synchronisation techniques. In addition, he or she is expected to work also on the correctness of lock-free data structure implementations. In particular, he or she will study how specifications as developed by the first PhD student have to be adapted in the lock-free case to be still correct and usable.
Both post docs will be appointed for 3 years. Both post docs are supposed to be actively involved in the full project, and they are supposed to collaborate with the PhD students. The first post doc will be appointed in the beginning, the second in the middle of the project, so that there always is a post doc involved in the project, who can provide back-up support for the PhD students (for example, in case the PI is travelling). In addition, the first post doc will work on the development of an appropriate specification language, combining readability à la JML and expressiveness related with concurrency as in separation logic. Further, he or she will also work on the automated verification of proof obligations, by developing or extending appropriate solvers for separation logic, and developing ownership type-based techniques to prove the absence of aliasing automatically. As mentioned above, the second post doc, will start later in the project, when initial results will have been developed. He or she will study the generation of ``obvious´´ specifications, and how the developed techniques can be adapted and generalised to distributed data structures (combining concurrent and distributed aspects of programming languages).
The last focus point of the project is tool support. Both PhD students and post docs are expected to implement their results. In addition, the project will also fund a scientific programmer of the Formal Methods and Tools team, for 20 % of his time, for the full duration of the project. The scientific programmer will ensure that the different developments are combined into a single tool, provide an effective user interface and guarantee maintenance of the tool. To ensure that the outcome of the project is a tool that

	Cost Category	Year 1	Year 2	Year 3	Year 4	Year 5	Total (Y1-5)
Direct Costs:	*Personnel:*						
	PI	39.150	39.150	39.150	39.150	39.150	195.750
	PhD Student 1	44.000	44.000	44.000	44.000		176.000
	PhD Student 2		44.000	44.000	44.000	44.000	176.000
	Post doc 1	64.000	64.000	64.000			192.000
	Post doc 2			64.000	64.000	64.000	192.000
	Other	12.800	12.800	12.800	12.800	12.800	64.000
	Total Personnel:	**159.950**	**203.950**	**267.950**	**203.950**	**159.950**	**995.750**
	Other Direct Costs:						
	Equipment	4.000	2.000	2.000			8.000
	Consumables	3.000	3.000	3.000	3.000	3.000	15.000
	Travel	7.500	17.500	20.000	10.000	7.500	62.500
	Publications, etc						
	Other						
	Total Other Direct Costs:	**14.500**	**25.000**	**27.500**	**15.500**	**10.500**	**85500**
	Total Direct Costs:	**174.450**	**228.950**	**295.450**	**219.450**	**170.450**	**1.088.750**
Indirect Costs (overheads):	Max 20% of Direct Costs	34.890	45.790	59.090	43.890	34.090	**217.750**
Subcontracting Costs:	(No overheads)		2.500	2.500	2.500		**7.500**
Total Costs of project:	(by year and total)	**209.340**	**274.740**	**354.540**	**263.340**	**204.540**	**1.306.500**
Requested Grant:	(by year and total)	**209.340**	**274.740**	**354.540**	**263.340**	**204.540**	**1.306.500**

For the above cost table, please indicate the % of working time the PI dedicates to the project over the period of the grant:	**50%**

Figure 4: Budget VerCors Project

actually is more than an academic prototype, and that it can be used by external parties, we think it is essential to have a professional developer funded by the project.

The only equipment that is required by the project is a computer per person (for people hired directly on the project, *i.e.*, the PhD students and post docs). In addition, travel costs for the PI, PhD students and post docs are estimated on 2,500 Euros per year per person. Further, in the second and third year an additional 7,500 Euros are budgeted to finance a long-term visit (expected duration: 3 months) of the PhD students to another research group abroad. Other costs include various consumables and the accountant statements. For the indirect costs, 20 % is computed as overhead.

Bibliography

1. E. Ábrahám. *An Assertional Proof System for Multithreaded Java – Theory and Tool Support.* PhD thesis, University of Leiden, 2004.
2. J. Barnat, L. Brim, P. Šimeček, and M. Weber. *Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking.* In C.R. Ramakrishnan, J. Rehof, eds., Proceedings of TACAS 2008, LNCS 4963, pp. 48-62, 2008.

3. M. Barnett, K.R.M. Leino, and W. Schulte. *The Spec# programming system: An overview.* In Proceedings of Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04), LNCS 3362 . Springer, 2005.
4. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. *JACK: A tool for validation of security and behaviour of Java applications.* In Formal Methods for Components and Objects (FMCO 2006), LNCS 4709, pp. 152-174. Springer, 2007.
5. B. Beckert, R. Hähnle, and P.H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334. Springer, 2007.
6. J. Berdine, C. Calcagno, and P.W. O'Hearn. *Smallfoot: Modular automatic assertion checking with separation logic.* In Formal Methods for Components and Objects, LNCS 4111, pp. 115-137, Springer, 2005.
7. J. van den Berg and B. Jacobs. *The LOOP compiler for Java and JML.* In T. Margaria and W. Yi, eds., Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), LNCS 2031, pp. 299-312. Springer, 2001.
8. K. Bierhoff and J. Aldrich. *Modular typestate verification of aliased objects.* In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 2007.
9. R. Bornat, P.W. O'Hearn, C. Calcagno, and M. Parkinson. *Permission accounting in separation logic.* In J. Palsberg and M. Abadi, eds., Principles of Programming Languages, pp. 259-270. ACM Press, 2005.
10. J. Boyland. *Checking interference with fractional permissions.* In R. Cousot, eds., Static Analysis Symposium, LNCS 2694, pp. 55-72. Springer, 2003.
11. C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. *Formal methods for smart cards: an experience report.* Science of Computer Programming, 55(1-3):53-80, 2005.
12. D. Cok and J.R. Kiniry. *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system.* In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, eds., Proceedings of the International Workshop CASSIS 2004, LNCS 3362, pp. 108-128. Springer, 2005.
13. M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. *Dynamically discovering likely program invariants to support program evolution.* IEEE Transactions on Software Engineering, 27(2):1-25, 2001.
14. C. Flanagan and K.R.M. Leino. *Houdini, an annotation assistant for ESC/Java.* In J.N. Oliveira and P. Zave, eds., Formal Methods Europe 2001 (FME'01): Formal Methods for Increasing Software Productivity, LNCS 2021 , pp. 500-517. Springer, 2001.
15. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. *Local reasoning for storable locks and threads*. In Z. Shao, eds., Asian Programming Languages and Systems Symposium, LNCS 4807, pp. 19-37. Springer, 2007.
16. C. Haack, M. Huisman, and C. Hurlin. *Reasoning about Java's reentrant locks*. In G. Ramalingam, eds., Asian Programming Languages and Systems Symposium, LNCS 5356, pp. 171-187. Springer, 2008.
17. C. Haack and C. Hurlin. Separation logic *contracts for a Java-like language with fork/join.* In J. Meseguer and G. Rosu, eds., Algebraic Methodology and Software Technology, LNCS 5140, pp. 199-215. Springer, 2008
18. C.A.R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576-580, 1969.
19. C.A.R. Hoare and J. Misra. *Verified software: Theories, tools, experiments vision of a grand challenge project.* A companion paper for VSTTE 2005, held in Zürich, Switzerland, 2005.
20. A. Hobor, A. Appel, and F.Z. Nardelli. *Oracle semantics for concurrent separation logic*. In Programming Languages and Systems: Proceedings of the 17th European Symposium on Programming, ESOP 2008, LNCS 4960, pp. 353-367. Springer, 2008.
21. M. Huisman, B. Jacobs, and J. van den Berg. *A Case Study in Class Library Verification: Java's Vector Class*. Software Tools for Technology Transfer, 3/3:332-352, 2001.
22. M. Huisman. *Verification of Java's AbstractCollection class: a case study.* In E. Boiten and B. Möller, eds., Mathematics of Program Construction (MPC'02), LNCS 2386, pp. 175-194. Springer, 2002.
23. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. *Checking absence of illicit applet interactions: a case study*. In M. Wermelinger and T. Margaria, eds., Fundamental Approaches to Software Engineering (FASE 2004) LNCS 2984, pp. 84-98. Springer, 2004.
24. M. Huisman and G. Petri. *The Java memory model: a formal explanation*. In VAMP 2007: Proceedings of the 1st International Workshop on Verification and Analysis of Multi-Threaded Java-Like Programs, number ICIS-R07021 in Technical Report. Radboud University Nijmegen, 2007.

25. M. Huisman and G. Petri. *BicolanoMT: a formalisation of multi-threaded Java at bytecode level*. In BYTECODE, ENTCS. Elsevier, 2008.
26. C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université de Nice Sophia Antipolis, 2009.
27. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. *Reasoning about classes in Java (preliminary report).* In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98), pp. 329-340. ACM Press, 1998.
28. L. Jia and D. Walker. *Ilc: A foundation for automated reasoning about pointer programs.* In European Symposium on Programming, LNCS 3924, pp. 131-145, Springer, 2006.
29. C.B. Jones. *Tentative steps toward a development method for interfering programs.* ACM Transactions on Programming Languages and Systems, 5(4):596-619, 1983.
30. JSR 308 Expert Group. *Annotations on Java types.* Java specification request, Java Community Process, 2007.
31. K.R.M. Leino, P. Müller, and J. Smans. *Verification of concurrent programs with Chalice*. In Lecture notes of FOSAD, LNCS 5705. Springer, 2009.
32. J. Manson, W. Pugh, and S.V. Adve. *The Java memory model*. In Principles of Programming Languages, pp. 378-391, 2005.
33. C. Marché, C. Paulin-Mohring, and X. Urbain. *The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations.* Journal of Logic and Algebraic Programming, 58:89-106, 2004.
34. M. Moir and N. Shavit. *Concurrent Data Structures*. In Handbook of Data Structures and Applications, D. Metha and S.Sahni, eds., Chapman and Hall/CRC Press, Chapter 47, pp. 47-1-47-30, 2004.
35. P.W. O'Hearn. *Resources, concurrency and local reasoning*. Theoretical Computer Science, 375(1(3):271-307, 2007.
36. S. Owicki and D. Gries. An *axiomatic proof technique for parallel programs*. Acta Informatica Journal, 6:319-340, 1975.
37. J. Pang, W.J. Fokkink, R.F.H. Hofman and R. Veldema, *Model checking a cache coherence protocol for a Java DSM implementation.* Journal of Logic and Algebraic Programming 71(1):1-43
38. M. Parkinson. *Local reasoning for Java.* PhD thesis, appeared as Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.
39. J.C. Reynolds. Separation *logic: a logic for shared mutable data structures*. In Logic in Computer Science. IEEE Press, 2002.

Ethical issues table

	Research on Human Embryo/ Foetus	YES	NO
	Does the proposed research involve human Embryos?		√
	Does the proposed research involve human Foetal Tissues/ Cells?		√
	Does the proposed research involve human Embryonic Stem Cells (hESCs)?		√
	Does the proposed research on human Embryonic Stem Cells involve cells in culture?		√
	Does the proposed research on Human Embryonic Stem Cells involve the derivation of cells from Embryos?		√
	DO ANY OF THE ABOVE ISSUES APPLY TO MY PROPOSAL?		√

	Research on Humans	YES	NO
	Does the proposed research involve children?		√
	Does the proposed research involve patients?		√
	Does the proposed research involve persons not able to give consent?		√
	Does the proposed research involve adult healthy volunteers?		√
	Does the proposed research involve Human genetic material?		√
	Does the proposed research involve Human biological samples?		√
	Does the proposed research involve Human data collection?		√
	DO ANY OF THE ABOVE ISSUES APPLY TO MY PROPOSAL?		√

	Privacy	YES	NO
	Does the proposed research involve processing of genetic information or personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		√
	Does the proposed research involve tracking the location or observation of people?		√
	DO ANY OF THE ABOVE ISSUES APPLY TO MY PROPOSAL?		√

	Research on Animals	YES	NO
	Does the proposed research involve research on animals?		√
	Are those animals transgenic small laboratory animals?		√
	Are those animals transgenic farm animals?		√
	Are those animals non-human primates?		√
	Are those animals cloned farm animals?		√
	DO ANY OF THE ABOVE ISSUES APPLY TO MY PROPOSAL?		√

	Research Involving Developing Countries	YES	NO
	Does the proposed research involve the use of local resources (genetic, animal, plant, etc)?		√
	Is the proposed research of benefit to local communities (e.g. capacity building, access to healthcare, education, etc)?		√
	DO ANY OF THE ABOVE ISSUES APPLY TO MY PROPOSAL?		√

	Dual Use	YES	NO
	Research having direct military use		√
	Research having the potential for terrorist abuse		√

	DO ANY OF THE ABOVE ISSUES APPLY TO MY PROPOSAL?		√

Other Ethical Issues	YES	NO
Are there **OTHER** activities that may raise **Ethical Issues**?		√
If **YES** please specify:		

If you have answered "YES" to any of the above questions you are required to complete and upload the "B2_Ethical Issues Annex" (template provided).

Without this Annex, your application cannot be properly evaluated and the granting process will not proceed if successful.

Please see Annex 2a and 2b of the Guide for Applicants – ERC Grant Schemes for further details and CORDIS http://cordis.europa.eu/fp7/ethics_en.html for further information on how to deal with Ethical Issues in your proposal.

Section 3: Research Environment

a. PI's Host institution

University of Twente

The PI is hosted in the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente in Enschede, Netherlands. The University of Twente distinguishes itself from the other Dutch universities as an *entrepreneurial research* university. It is the only campus university in the Netherlands and provides academic education and research in a wide variety of fields, ranging for example from psychology to public administration and from applied physics to biomedical technology. The university is part of the 3TU.Federation in which it cooperates with Delft University of Technology and Eindhoven University of Technology. The University of Twente is also a member of the European Consortium of Innovative Universities.

Centre for Telematics and Information Technology

Research within the University of Twente has been organised in six research institutes. Computer science research is organised within the Centre for Telematics and Information Technology (CTIT). CTIT is one of the largest academic ICT research institutes in Europe, with a budget of 28.5 million Euros. Over 475 researchers, distributed over 28 research groups, actively participate in the research programme. Integration of technology-based research and its application in specific domains is a clear focus of CTIT. The institute maintains an extensive international network of contacts and working relations with academia and industry. This network includes ICT and manufacturing companies, universities and research institutes, health care organisations, financial institutes, governmental organisations, and logistics service providers. Since CTIT covers a broad range of different ICT research, it stimulates scientists to collaborate and to explore multidisciplinary topics. It also enables students to follow courses in new fields and domains that cross the boundaries of traditionally structured schools in the computer science.

Strategic Research Orientations

Research within CTIT is bundled into six Strategic Research Orientations (SRO) that allow to bring together a sufficiently large number of researchers on attractive topics. Because of the different backgrounds of the different groups, this allows to combines scientific challenges with potential applicability. The PI's research group Formal Methods and Tools (FMT) participates in two of these SROs: Dependable Systems and Networks (DSN) and Integrated Security and Privacy in a Networked World (ISTRICE). For ISTRICE, the PI is the contact person within the FMT group. Participation in the SRO also is beneficial for the participation in other (national) projects and networks, and provides more visibility to establish industrial contacts.

Centre for Dependable ICT Systems

CTIT collaborates with the other Dutch technical universities in the Netherlands Institute for Research on ICT (3TU.NIRICT). The creation of NIRICT has led to the creation of a Centre of Excellence, called Centre for Dependable ICT Systems (CeDICT), in which the FMT group is also involved. CeDICT has a budget of 12 million Euros for research on dependable systems (of which 1.75 million Euros are for the FMT group). The participation of the FMT group in CeDICT provides an excellent opportunity to collaborate with other researchers in the Netherlands working on program verification of concurrent and distributed software.

Graduate school on Dependable and Secure Computing

Further, the FMT group is involved in the new graduate school on Dependable and Secure Computing (DeSC). This enables the PhD students within the VerCors project to enrol in the last part of the graduate school program. Enrolling in the graduate school program provides the PhD students with the opportunity to follow extra courses, which can be research-related, to deepen the knowledge or widen the perspective of the PhD student, but also to support personal development of the student (*e.g.* writing and communication skills, career orientation, project or time management). The graduate school will also be a possible source to find well-qualified PhD students.

Institute for Programming research and Algorithmics

In addition, the FMT group is also involved in the Institute for Programming research and Algorithmics (IPA); a national research school, whose goal is to educate researchers in the field of programming research

and algorithmics, comprising the study and development of formalisms, methods, and techniques to design, analyze, and construct software systems and components. IPA provides a curriculum for the education of PhD students that consists of three basic courses and two advanced courses on new topics every year. Further, IPA sponsors events organised by its members that contribute to the education of its PhD students, and they work together with research schools from other countries in the European Educational Forum to organize educational activities such as summer schools. In addition, IPA strives for the transfer of knowledge from its researchers to people in industry. For this they organise feedback days, where people from academia and industry meet to discuss a hot topic of mutual interest, and they publish an IPA news letter, which contains articles on joint projects between IPA research groups and Dutch companies. All members of the VerCors team will participate in IPA, and will be able to attend events organised by IPA.

Formal Methods and Tools Group

Finally, the work in the VerCors project will bring new expertise to the FMT group, but it will also benefit from available expertise within the FMT team. In particular, within the group a broad knowledge exists on program verification and specification, but more from a model checking point of view (Dr.Ir. Arend Rensink, Dr.Ir. Theo Ruys).

Further, the team is also working on the development of multi-core model checking algorithms (Prof.Dr. Jaco van de Pol, Dr.Rer.Nat. Michael Weber). For the correctness of the model checking algorithm, the correctness of concurrent or distributed hash tables is essential. Therefore, Van de Pol and Weber will be able to provide useful feedback on the practical applicability of the verification technique being developed. We also expect that their implementations can serve as the basis for one of the larger case studies within the VerCors project.

Prof. van de Pol will act as the formal supervisor for the PhD students (this is in line with Dutch regulations).

Finally, the FMT group owns a special cluster with large computing power. This cluster can also be used for experiments, *e.g.*, with the automatic verification of proof obligations, within the VerCors project.

b. Additional institutions (additional participants)

There are no additional institutions involved in the project.