# Using Regression Verification to Revalidate Real-Time Software on Multicore Computers

Sagar Chaki[1], Arie Gurfinkel[1], and Ofer Strichman[2]

[1] SEI/CMU
[2] Technion

## 1 Position

Technological innovation is the hallmark of the computer hardware industry. Keeping pace with this innovation is a major challenge for software engineering: new hardware makes new resources available, but to take advantage of them the software must be migrated (or ported) to the new hardware platform. This is not trivial. An idealistic approach of developing and validating new software from scratch for every (significant) improvement in hardware is impractical in view of deadlines and budgetary constraints. For example, it is infeasible to rewrite all of the existing sequential software to take full advantage of the new multi-core CPUs. At the same time, it is dangerous to directly reuse software written for one platform on another. A new platform changes the underlying assumptions and requires the software to be revalidated, at great cost. This is particularly a problem in safety-critical domains that depend on complex software with lifetimes spanning decades and several major technological changes. In the rest of this position, we call the problem of deciding whether an existing system behaves correctly under a new set of environmental assumptions the *revalidation problem*. Note that by environmental assumptions we broadly mean the assumptions on the environment in which the program is executing. This includes assumptions on the actual physical environment as well as assumptions on the hardware platform, the architecture of the system, etc.

We believe that automated program verification can play a crucial role in ameliorating the revalidation problem. By analyzing the source code of the program statically, it is possible to infer what happens in all possible run-time executions without ever executing a single line of code. By varying the semantics used by the analysis, it is possible to capture different environmental assumptions. Thus, a single program analysis pass reveals how specific changes in environmental assumptions affect the execution of the software.

One technique, *regression verification* [**?**,**?**,**?**] – deciding the behavioral equivalence of two closely related programs – stands out as a way to improve usability of automated verification for re-validation. First, it completely sidesteps the need for formal specifications – the base program takes the place of the specification. Second, there are various opportunities for abstraction and decomposition that only apply when establishing an equivalence of similar programs – hence, improving scalability. In the best possible scenario, the effort required for regression

verification is proportional only to the difference between programs being compared, and not to their absolute sizes. This makes the approach tractable in practice. Third, often, invariants strong enough to prove equivalence of recursive functions and loops can be generated automatically. Regression verification is now supported by two research tools, RVT [**?**] and Microsoft's SymDiff[3].

The original definition of regression verification applies to proving equivalence of similar software. We believe that it extends naturally to revalidation. Although, in this case, the effort required must also take into account the difference in the assumptions.

While the challenge of migrating software between platforms occurs in many contexts, we believe that the problem is particularly severe in the case of migration of real-time embedded (RTE) systems from single-core to a multi-core platforms. There are several reasons for concentrating on this domain. First, RTE systems are often safety-critical. Their validation is necessary and costly. Second, they are tightly integrated in our daily life. Arguably, human existence as we know it today depends on RTE systems operating correctly. Third, the RTE platforms are often very restricted (a prerequisite for predictable real-time behavior). This might be leveraged for a scalable analysis.

In summary, we argue that, today, there is a lack of usable automated verification techniques to aid in migrating RTE systems from single-core to multi-core environments. We believe that applying regression verification to this problem is a promising direction of future research that address the two major usability challenges – formal specification of requirements and scalability. In the rest of this position, we elaborate on some of the problems of migrating from single-core to multi-core, give a brief background on regression verification, and conclude with an outline of open challenges.

## 2 Migrating to Multicore

At a first glance, it appears that from safety and security perspectives, porting of an RTE software to multi-core platforms is trivial. After all, the software has been extensively validated and possibly even formally verified on single-core platforms, and is not being modified. It must, therefore, be functionally unchanged and remain as safe and secure on multi-core hardware. But, this is not so! The key insight is that software's behavior depends not only on the source code itself, but also on the underlying hardware (i.e., on the architectural assumptions provided by the hardware). Catastrophic failures have occurred when legacy software has been used without due consideration to changes in the environment in which the software is operating. A well known example is the infamous Ariane 5 disaster [**?**].

In the context of multi-core platforms, the crucial change in assumption is the switch from virtual to real concurrency. In a single-core system, concurrency is "virtual" in the sense that only one instruction is executing at any given time.

[3] `http://research.microsoft.com/en-us/projects/symdiff`

The tasks (or threads) are executed concurrently, but the instructions (or atomic blocks) are not. In a multi-core system, concurrency is "real" – multiple threads run on multiple cores with multiple instructions (or atomic blocks) executing concurrently.

Real concurrency has real consequences. For example, RTE systems exploit virtual concurrency in the widely used priority ceiling mutual exclusion protocols (PCP) [**?**]. A PCP ensures an exclusive access to a shared resource (e.g., memory, peripheral devices, etc.) by allowing the highest-priority threads to access shared resources at will. The platform ensures that no other thread is able to preempt (and execute) while the highest-priority thread is accessing the resource. The priority of a thread changes dynamically at run time according to the shared resource it attempts to access. Specifically, each resource has a priority ceiling, and whenever a thread $t$ locks a resource $r$, its priority is raised to the ceiling of $r$. This way, no other thread that wants to access $r$ is a able to get scheduled. PCP is widely used in embedded software owing to its simplicity. It requires no special mutual-exclusion primitives (e.g., locks, semaphores, monitors etc.), and (in combination with priority inheritance) reduces chances of concurrency-related errors like races and deadlocks.

However, priority-ceiling breaks down on multi-core platforms! Here, having the highest priority does not guarantee exclusive access to the hardware. This leads to more thread interleavings, and, therefore, to new races and deadlocks. Therefore, software that runs safely and securely on a single-core platform might misbehave on a multi-core hardware. A major challenge for architecture migration is to detect such problems as early as possible (and prevent them as early as possible).

## 3 Current Approaches

A number of known techniques can be used to address revalidation problems, including testing, static analysis and software model checking. Each of these complementary methods has its advantages and disadvantages compared to regression verification. Testing is of course relatively easy to do, but is not exhaustive. Critical errors have escaped detection even after years of rigorous state-of-the-art testing effort. This problem is even more acute for concurrent programs where testing is able to explore only a minute fraction of the enormous number of inter-thread interactions. Exhaustive approaches, like static analysis and model checking are expected to be more vulnerable to scalability issues, because unlike regression verification, it cannot use the fact that most of the code (or all, in this case) hasn't change in order to simplify the verification problem. More importantly, they require a target specification to verify. Writing down appropriate specifications is known to be difficult and time-consuming. In the absence of good specifications, exhaustive approaches provide only limited guarantees. It seems that there is no "silver bullet" for this problem, and novel solutions must be explored to complement and aid existing ones. In particular, we propose to explore the applicability of program equivalence techniques in this context.

*Regression Verification.* The problem of proving the equivalence of two successive, closely related programs $P_{old}$ and $P_{new}$ is called *regression verification* [**?**,**?**,**?**]. It is potentially easier in practice than applying functional verification to $P_{new}$ against a user-defined, high-level specification. There are three reasons for this claim, as was briefly mentioned in the introduction. First, it circumvents the complex and error-prone problem of crafting specifications. In some sense, regression verification uses $P_{old}$ as the specification of $P_{new}$. Second, there are various opportunities for abstraction and decomposition that are only relevant to the problem of proving equivalence between similar programs, and these techniques reduce the computational burden of regression verification [**?**]. Specifically, the computational effort is proportional to the change, rather than to the size of the original program. This is in stark contrast to testing and functional verification: in testing, a change in the program requires the user to rerun the whole (system) test suite; in formal verification, depending on the exact system being used, reusing parts of the previous proof may be possible, but it is far from simple and in general not automated. Third, loops and recursion are typically not a problem, because as suggested in [**?**], the statement that the two compared functions are equivalent is typically an inductive invariant. It is simple to test this invariant by using uninterpreted functions.

Both functional verification and program equivalence of general programs are undecidable problems. Coping with the former was declared in 2003 by Tony Hoare as a "grand challenge" to the computer science community [**?**]. Program equivalence can be thought of as a grand challenge in its own right, but there are reasons to believe, as indicated above, that it is a "lower hanging fruit". The observation that equivalence is easier to establish than functional correctness is supported by past experience with two prominent technologies: (i) regression testing – the most popular automated testing technique for software, and (ii) equivalence checking – the most popular formal verification technique for hardware. In both cases the reference is a previous version of the system.

Although, as listed above, regression verification has its advantages, the notion of correctness guaranteed by equivalence checking is weaker than complete verification: rather than proving that $P_{new}$ is "correct", we prove that it is "as correct" as $P_{old}$. However, equivalence checking is still able to expose functional errors since failing to comply with the equivalence specification indicates that something is wrong with the assumptions of the user. In practice, this is of tremendous benefit. In addition, due to its lower complexity, equivalence checking is often feasible in cases where the alternative of complete functional verification is not.

## 4 The Road Ahead

In this section, we list some of the key challenges in the areas of *semantics*, *scalability*, and *evaluation*.

*Semantics.* All the prior research in regression verification has focused on checking equivalence between two syntactically different *sequential* (i.e., one thread

of control) programs running on identical hardware. In contrast, revalidation requires checking equivalence between two syntactically identical *concurrent* (i.e., multiple threads) programs running on different hardware. Therefore, existing notions of program equivalence used in regression verification today do not apply directly. This is further complicated by the reactive nature of most RTE systems.

*Scalability.* Most of the research on verification of concurrent systems does not exploit any knowledge of a scheduler – a scheduler is assumed to be completely non-deterministic. This is a reasonable assumption for systems that are expected to work under a variety of scheduling disciplines. However, RTE systems operate in restricted environments, and the exact knowledge of the scheduler is exploited in their timing analysis. Exploiting this knowledge for achieving scalability in functional verification remains a challenge.

*Evaluation.* It is important to establish a re-validation challenge problem for the research community motivated by industrial needs. We hope that we can leverage an interaction with experts in real-time embedded systems to construct examples that highlight the problems faced in the industry. This must include a description of the old and new architectures and a formal model (perhaps a C/C++ code) of the software.

To summarize, we argue that revalidation of real-time embedded code on multicore computers is an important problem, and that regression verification has a reasonable chance to cope with it.

## References


1. Godlin, B., Strichman, O.: "Regression Verification". In: Proceedings of the 46th Design Automation Conference (DAC'2009). pp. 466–471. San Francisco, California, USA (2009)
2. Hoare, C.: "The Verifying Compiler: A Grand Challenge for Computing Research". Journal of the ACM (JACM) 50(1), 63–69 (2003)
3. Sha, L., Rajkumar, R., Lehoczky, J.P.: "Priority Inheritance Protocols: An Appoach to Real-Time Synchronization". IEEE Transactions on Computers 39(9), 1175–1185 (1990), doi:10.1109/12.57058
4. Strichman, O.: "Regression Verification: Proving the Equivalence of Similar Programs". In: Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09). p. 63. Grenoble, France (2009)
5. Strichman, O., Godlin, B.: "Regression Verification – A Practical Way to Verify Programs". In: Proceedings of First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments (VSTTE'05). pp. 496–501. Zurich, Switzerland (2005)
6. Wikipedia: Ariane 5 Flight 501, `http://en.wikipedia.org/w/index.php?title=Ariane_5_Flight_501&oldid=388509601`