2016 IEEE 17th International Symposium on High Assurance Systems Engineering

High-Integrity Multitasking in SPARK: Static Detection of Data Races and Locking Cycles

S. Tucker Taft
AdaCore
Lexington, MA USA
Email: taft@adacore.com

Florian Schanda
Altran UK Limited
Bath, UK
Email: florian.schanda@altran.com

Yannick Moy
AdaCore
Paris, France
Email: moy@adacore.com

***Abstract*—SPARK is a subset of Ada designed to enable formal verification. A new release of SPARK 2014, based on the Ada 2012 standard, incorporates support for multitasking, based on the Ravenscar Profile, which subsets the full Ada tasking model to a relatively static, single-level tasking model. This paper describes the safety requirements relating to multitasking in this version of SPARK, and the corresponding static checks performed by the SPARK 2014 toolset.**

***Index Terms*—formal verification, multitasking.**

I. Introduction

The SPARK language and toolset [1] are designed to support the development and formal verification of software systems providing the highest level of safety and security. The 2014 version of the SPARK language is based on the Ada 2012 programming language standard [2], subsetted to enable formal verification, and augmented using the Ada 2012 annotation syntax to support more complete behavioural specifications. The additional annotations allow the full specification of functional effects and information flow of every component of the software system, which can then be verified fully statically, or using a combination of static and dynamic techniques.

The initial release of SPARK 2014 was limited to the sequential constructs of the language. The latest release incorporates support for real-time multitasking, including explicit declarations of *task types* and *task objects* to provide multiple threads of control, and data-oriented synchronization and coordination capabilities based on the monitor-like *protected type* feature of Ada.

The formal verification of SPARK programs is supported by a set of tools that translate the SPARK programs into an intermediate verification language (Why3 [3]), which is then transformed into a set of verification conditions (VCs) that can be discharged by one or more SMT solvers [4] (including CVC4, Alt-Ergo, and Z3). In addition, the SPARK 2014 programs may be compiled by an Ada 2012 compiler to produce an executable program, allowing the use of run-time checking of assertion expressions, including preconditions, postconditions, and type invariants, for parts of the code not yet statically verified.

As part of supporting concurrency in the new release of SPARK 2014, an additional set of safety requirements with corresponding static checks has been specified to ensure that there are no data races, no locking cycles, and no task suspensions while holding a lock, in the resulting programs. This article makes the following contributions:

- Language features which facilitate static checking of safety requirements,
- and which static checks are performed by the SPARK 2014 toolset.

II. Multitasking safety requirements

The Ada language defines a set of rules for the safe use of data objects shared among multiple tasks [2]. Ada does not generally require that violations of these rules be detected either statically or dynamically, and, if they are violated, the effects are generally not predictable. By contrast, the SPARK 2014 toolset statically detects possible violations of these rules; in other words, the SPARK 2014 toolset performs static data-race detection.

Ada also provides mechanisms to avoid cyclic locking structures, which can lead to deadlock. However, these mechanisms are enforced by a *priority-ceiling* approach, which only provides protection on a monoprocessor, and are based on run-time checks between the priority of the task requesting exclusive access to a resource, and the *ceiling priority* of the resource [5]. The SPARK 2014 toolset extends this protection against cyclic locking to multiprocessor contexts, and enforces the rules statically rather than dynamically.

Finally, Ada has a rule disallowing executing a *potentially blocking* operation while holding a lock. As with data races, this rule is not required to be enforced by a standard Ada compiler nor by the Ada run-time system, though if a violation is detected, the run-time system will raise a run-time exception. By contrast, the SPARK 2014 toolset enforces this rule statically.

III. Static multitasking checks

The multitasking release of SPARK supports both *tasks*, which represent a separate thread of control, and *protected objects*, which are resources with *procedure* and *entry* operations that upon call acquire exclusive read/write access to the object, and *function* operations that upon call acquire shared read-only access to the object.

Data races are eliminated in SPARK by a simple rule: any global object referenced from a task shall be marked as `Part_Of` that task, or be a *synchronized* object. A *synchronized* object is an object that can support simultaneous access by multiple tasks without incurring a data race. This includes protected objects, *atomic* objects (all access is via atomic instructions), and *suspension* objects (a kind of private semaphore). Normal objects, such as integer or floating point variables, are not synchronized, and, if global, may be referenced only by the task they are associated with via a `Part_Of` annotation. If the global variable has no `Part_Of` annotation, then it may only be referenced by the *environment task* of the program, that is the *main* task, the task in which execution of the program begins.

The data race rule is enforced relatively easily in SPARK because all subprograms must identify the global variables they manipulate with a `Global` annotation. Hence, the main body of a task must not call a subprogram whose `Global` annotation identifies an object which is not synchronized and not `Part_Of` the calling task. The `Global` annotation of a subprogram includes all indirect references as well as direct references, so no hidden side effects are possible. Note that, as a convenience to the user, the SPARK 2014 toolset can also *infer* `Global` annotations automatically, but the rule remains effectively the same, based on the inferred `Global` annotations.

Cyclic locking, potential violations of ceiling priority rules, and use of potentially blocking operations while holding a lock, are all checked for by the SPARK 2014 toolset by propagating across calls information about the invocation of operations on protected objects. In a later release, programmers will be able to declare their usage of protected operations and potentially blocking operations with explicit annotations on the specification of a subprogram. But in the current release, the toolset propagates the information automatically, rather than relying on user-provided annotations.

For cyclic locking, the check is simply that a protected procedure or entry of a given object must not make an *external* call on a protected operation of the same object, directly or indirectly. For the ceiling priority rule, the check is that the priority of a task must be less than or equal to the ceiling priority of any protected object it operates upon, directly or indirectly. Similarly, a protected operation whose object has a given ceiling priority must not call a protected operation on an object with a lower ceiling priority, directly or indirectly. Finally, for the potentially blocking rule, the static check is that a protected operation does not invoke, directly or indirectly, a potentially blocking operation.

The data-race rule is checked in a *modular* fashion, in that it can be performed without access to the complete program, thanks to the user-provided `Global` annotations. By contrast, these locking-related checks all rely on a propagation of information across calls, globally throughout the program. This means that these checks are *not* modular and cannot be performed without access to the entire program. Once the SPARK toolset supports the explicit specification on a subprogram of the protected and potentially blocking operations it performs, these checks can be performed in a modular fashion. Nevertheless, the toolset will still provide support for *inferring* these annotations, so the programmer can still have the convenience of omitting these explicit annotations, presuming they do not have a requirement for modular checking.

IV. Related work

Detection of race conditions and deadlocks has a relatively long history, and both static [6] and dynamic [7] approaches have been used. A unique feature of the SPARK detection approach is that it can provide a guarantee that no data races, no cyclic locking, and no violations of the ceiling priority and potentially-blocking rules remain in the program, because the language has itself been subsetted to enable static detection of all such violations. This guarantee is important to users of the SPARK language, as the applications are often at the highest level of criticality. In other contexts, identifying race conditions or deadlocks is seen as simply finding another kind of program defect. In SPARK, if multitasking is used, then eliminating possible data races, cyclic locking, and other sources of potential deadlocks is considered part of ensuring the overall safety of the software. The work presented in this article was based on the RavenSPARK [8] profile of earlier SPARK toolsets; a more restricted language with less support for the standard Ada run-time and implementation that only supported modular analysis as it could not infer contracts.

V. Future Work

Some incremental extensions to the language will be natural, such as annotations for announcing tasks and relaxing the Ravenscar profile. Automatic abstraction to communicating sequential processes (CSP) may be one way to achieve this. Other work could include support for the proposed parallel programming extensions [9] for Ada.

Acknowledgements

This work was performed as part of a joint development project of AdaCore and Altran UK.

References

[1] J. W. McCormick and P. C. Chapin, *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[2] International Standards Organization, "ISO IEC 8652:2012," Programming Languages and their Environments – Programming Language Ada, 2012.

[3] J.-C. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *Programming Languages and Systems*. Springer, 2013, pp. 125–128.

[4] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[5] J. W. McCormick, F. Singhoff, and J. Hugues, *Building parallel, embedded, and real-time applications with Ada*. Cambridge University Press, 2011.

[6] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 237–252.

[7] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 221–234.

[8] *The SPARK Ravenscar Profile*, Altran UK Limited, April 2004.

[9] B. Moore, L. M. Pinho, S. Michell, and T. Taft, "Safe parallel programming in ada with language extensions," in *High Integrity Language Technology ACM SIGAdas Annual International Conference*, 2014.