

Termination of Periodically Monotonic Affine Loops

Kevin Durant and Willem Visser

Department of Mathematical Sciences

Stellenbosch University, Victoria Street, Stellenbosch

email: {kevdur, willem}@gmail.com

tel: +27 21 808 4232; fax: +27 21 882 9865

Abstract—We present a technique for locating infinite paths within a program’s loop blocks, in particular examining loops whose defining variables are updated by transformations of an affine nature. Along with a few auxiliary results, we show that this technique will always locate such paths, if they exist, for update transformations which periodically shift the values of the loop’s variables no nearer to their relevant bounds, which are defined in the loop’s guard condition. All single-variable and cyclic affine loops possess this property. Finally we describe this technique’s implementation, which has been performed with the aid of Java Pathfinder, The National Aeronautics and Space Administration’s (NASA) open-source software verifier, and also note how our technique can be adapted to prove that termination holds for affine loops which exhibit periodically decreasing behaviour.

Index Terms—Internet services & end user applications; software design.

I. INTRODUCTION

The development of software, for any purpose, can be a rather intricate process. In and amongst design goals and principles, resource management and quality control lies possibly the most fundamental requirement of all: correctness. The software must fulfill its design requirements, and should handle unforeseen circumstances gracefully. The pursuit of such software leads one to the research field of model checking, and our research meets this parent field upon the topic of program termination — the idea that, unless explicitly desired, a program should not continue to execute infinitely. Termination is itself no small principle — for what might be an enlightening look at the approachability of the problem the reader is directed to [1] — but many of the problems it generates in computer science can be transformed into questions regarding the reachability and stability of loop systems. In addition, due to its simple form, the loop termination problem provides a clear illustration of the practicality of termination proving.

A few recent papers to discuss this subject are [2], [3], [4], [5]. The accepted approach for proving that a system always terminates is the synthesis of a ranking function — a map from the system’s allowed variable values into a well-order such as the natural numbers, which decreases (when the natural numbers are concerned) with each iteration through the system. If such a function exists, then after a finite number of iterations the mapped value must leave the well-order, implying that the variables’ values are no longer valid.

The authors of [6] have shown that the general problem of termination can, when fairness assumptions are present, be

reduced to the termination problem for unnested while loops. This result was then combined with their loop-termination algorithm (based upon the theory of ranking functions; see [4]) to implement a practically useful program termination validator.

The technique described in this paper differs from the normal practice, as, rather than concerning ourselves with finding a ranking function for a loop, we attempt to obtain information which might provide insight into the termination properties of the loop directly from the functions applied to the individual variables. Our primary focus, in fact, is not only to assert the termination of loops, but also to locate infinite paths within the system if they exist, i.e., to generate, if possible, initial values for the loop’s variables which will cause the loop to execute infinitely. Such values should provide valuable test cases for the developer.

The algorithm presented here has been implemented with the aid of the extensible Java Pathfinder¹ (JPF) and its symbolic execution library, *JPF-symbc*².

II. PERIODICALLY NON-DECREASING LOOPS

We consider loops in which variables are integer-valued and updated via the application of an affine transformation (the combination of a linear transformation and an integer shift) over the loop’s variables. We term such loops *affine*; Figure 1 depicts their general form. For the sake of sensibility, we shall

```
while ( $av > b$ ) {  
     $v = Av + c$ ;  
}
```

Fig. 1. The general form of an affine loop.

use terms such as “increasing” and “positive” when referring to the change affected upon the variables by the loop’s transformation. Strictly speaking, these terms are only relevant when the application of a variable’s update transformation increases a variable’s value, and when a guard constraint is of the homogenous form ($x > 0$). Such terms, however, are to be associated with the partial relation \succ , which refers to the difference of a state of the variables v and the bounds b defined in their guard condition ($av > b$). So, for example,

¹<http://babelfish.arc.nasa.gov/trac/jpf>

²<http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc>

for a loop with the guard condition $(x > 0 \wedge y < 0)$ we may write $(x_1, y_1) \succ (x_2, y_2)$ to describe the state where both x_1 and y_1 are no closer to crossing the condition's bounds than x_2 and y_2 , i.e., $x_1 \geq x_2$ and $y_1 \leq y_2$. We may also use the shorthand notation $v \succ 0$, as well as the term “positive”, when v satisfies the loop's guard condition.

$$\begin{array}{l} \text{while } (x > 0) \{ \\ \quad x = ax + c; \\ \} \end{array}$$

Fig. 2. A general homogenous single-variable affine loop.

Firstly, consider the homogenous single-variable loop, as depicted in Figure 2. Our goal is to find an initial value for x for which $T^l(x) > 0$ for all $l \geq 0$. In search of such an x , we consider paths within the loop defined by the update transformation T which, when executed, do not cause the termination of the loop, and leave x no closer to its (constraint's) bound than before:

$$Q_T = \{x: \exists k \text{ s.t. } T^l(x) > 0 \forall 0 \leq l < k \wedge T^k(x) \geq x\}.$$

Obviously, due to the finite number of integers between any initial x and its bound, $T^l(x) > 0 \forall l \geq 0 \Rightarrow x \in Q_T$. In fact, this increasing behaviour must be exhibited infinitely many times by x values which cause infinite loops, with a possibly arbitrary number of iterations separating each increase. The loops with which we are concerned are those for which some constant period k exists, such that applying the loop's transformation T k times will always leave the loop's variables no closer to exiting the loop than before. We shall refer to these loops as *periodically non-decreasing*, or more simply, *periodic*.

Periodically non-decreasing loops can be identified by checking whether the following condition holds for some period k :

$$v \in R_T(k) \Rightarrow T^k v \in R_T(k),$$

where

$$R_T(k) = \{v: T^l v > 0 \wedge T^{l+k} v \geq T^l v \forall 0 \leq l < k\}.$$

Stated more simply, the set $R_T(k)$ must be *recurrent*. If it is, then, for any $v \in R_T(k)$, we have

$$\begin{array}{ll} T^l v > 0 & \text{if } 0 \leq l < k \text{ and} \\ T^l v \geq v & \text{if } l \geq k, \end{array}$$

and v initiates an infinite path through the loop T , providing Lemma 1.

Lemma 1. *If $R_T(k)$ is recurrent for some k , then every $v \in R_T(k)$ generates a non-terminating path within the loop described by T . In particular, if $R_T(k)$ is recurrent and non-empty, the loop does not always terminate.*

$$\begin{array}{l} \text{while } (x > 0) \{ \\ \quad x = -x + 10; \\ \} \end{array}$$

Fig. 3. A simple, possibly non-terminating affine loop example.

Consider the loop described in Figure 3. The loop is terminating if $x \geq 10$, and non-terminating if $x \in (1, 9)$. Let $T(x) = -x + 10$. Then

$$R_T(1) = \{x: x > 0 \wedge T(x) \geq x\}$$

is not recurrent, as $1 \in R_T(1)$ but $T(1) = 9 \notin R_T(1)$, as $T(9) = 1 \not\geq 9$. However, $T^2(x) = -(-x + 10) + 10 = x$, so

$$\begin{aligned} R_T(2) &= \{x: x > 0 \wedge T(x) > 0 \\ &\quad \wedge T^2(x) = x \geq x \wedge T^3(x) = T(x) \geq T(x)\} \\ &= \{x: x > 0 \wedge T(x) > 0\} \end{aligned}$$

is recurrent, and one can deduce that $R_T(2) = (1, 9)$. We now show that the technique based upon recurrent sets is complete for periodically non-decreasing functions.

Lemma 2. *If, for some $k > 0$, the affine transformation T^k is a non-decreasing function, then $R_T(k)$ is recurrent.*

Proof: Assume $T^k v$ is a non-decreasing function, i.e., $v \succ w \rightarrow T^k v \succ T^k w$. Let $v \in R_T(k)$. To prove the recurrency of $R_T(k)$ we must show $T^k v \in R_T(k)$, i.e., $T^l(T^k v) \succ 0 \wedge T^{l+k}(T^k v) \geq T^l(T^k v) \forall 0 \leq l < k$. The validity of the first set of constraints follows from the assumption that $v \in R_T(k)$, as $v \in R_T(k) \Rightarrow T^l(T^k v) = T^{l+k} v \succ T^l v \succ 0$. Considering the remaining constraints, we obtain $T^{l+k}(T^k v) = T^k(T^{l+k} v) \geq T^k(T^l v) = T^{k+l} v \succ v$. Thus $T^k v \in R_T(k)$, and $R_T(k)$ is a recurrent set. ■

From Lemma 1, the recurrence and non-emptiness of $R_T(k)$ for some k implies that the loop does not always terminate. We now show that the converse (if an infinite path exists, $R_T(k)$ will not be empty) holds for periodically increasing loops and, in the absence of limiting preconditions, non-decreasing loops.

Lemma 3. *If T^k is an increasing function and an infinite path generator v_∞ exists, then $R_T(k)$ is recurrent and not empty. Furthermore, if T^k is a non-decreasing function with no limiting preconditions on v (except the loop block's guard constraints), then the existence of an infinite path generator v_∞ implies the recurrence and non-emptiness of $R_T(k)$.*

Proof: Assume T^k is a non-decreasing function, v_∞ is an infinite path generator, and $v_\infty \notin R_T(k)$. Then, for some $0 \leq l < k$, either $T^l v_\infty \preceq 0$ or $T^{l+k} v_\infty \prec T^l v_\infty$. The former case, however, cannot be true due to our assumption that v_∞ generates an infinite path, that is, $T^l v_\infty \succ 0 \forall l \geq 0$. Thus, $T^{l+k} v_\infty \prec T^l v_\infty$ for a suitable l .

If T^k is in fact monotonically increasing, then repeated applications of T^k will cause this decreasing behaviour to continue: $T^{l+k} v_\infty \prec T^l v_\infty \Rightarrow T^{dk}(T^{l+k} v_\infty) \prec T^{dk}(T^l v_\infty)$ for all $d \geq 0$. From some point onwards, we must have $T^{dk}(T^{l+k} v_\infty) \preceq 0$, contradicting the infinite nature of the path generated by v_∞ .

When dealing with a function T^k which is only non-decreasing, we also begin with the constraint which implies that $v_\infty \notin R_T(k)$: $T^{l+k}v_\infty \prec T^l v_\infty$, and note that then $T^{dk}(T^{l+k}v_\infty) \preceq T^{dk}(T^l v_\infty)$ for all $d \geq 0$. If, for v_∞ 's path, equality never holds for any $d \geq 0$, then the infinite nature of the path is contradicted as it was above, and if equality does hold, we have $T^{dk}(T^{l+k}v_\infty) = T^{dk}(T^l v_\infty)$ for some d , and the path becomes cyclic from some iteration index less than or equal to $l + dk$ onwards. Because $T^{l+dk}v_\infty$ forms part of the cycle of length k , it must be contained in $R_T(k)$. Thus, for periodically non-increasing loops with no preconditions on the value of v which would prevent $T^{l+dk}v_\infty$ from being an initial value, the existence of an infinite path implies the non-emptiness of $R_T(k)$. ■

$$\text{while } (x > 0) \{ \\ \quad x = -3x + 20; \\ \}$$

Fig. 4. A single-variable affine loop example.

Consider the example in Figure 4 and notice that if $T(x) = -3x + 20$, then $T^2(x) = -3(-3x + 20) + 20 = 9x - 40$, a monotonically increasing function. Thus any $x \in R_T(2)$ will generate an infinite path. In this case, $R_T(2) = \{5\}$, which contains the only infinite path generator for T , generating the cyclic path $5 \rightarrow 5 \rightarrow \dots$. This periodically monotonic behaviour is a general property of single-variable transformations, as detailed in the following lemma. Our technique is thus complete (let $k = 2$) for single-variable affine loops.

Lemma 4. *A single-variable transformation $T(x)$ is non-decreasing over a period of 2.*

Proof: The proof is simple, as all that is required is the repeated application of the single-variable transformation $T(x) = ax + c$:

$$T^2(x) = a(ax + c) + c \\ = a^2x + c(a + 1),$$

which is non-decreasing. ■

This property of single-variable loops, which allows us to limit our search to a finite number of periods, is also possessed by the cyclic loops described in Section III, but whether a similar bound exists for all periodically non-increasing affine loops is, to the authors' knowledge, yet unknown.

It is perhaps important to note that although the set R_T naturally lends itself towards non-decreasing functions, it is not limited to them in its applicability. Rather, it attempts to locate initial values for v which create a periodically non-decreasing path under repeated application of the transformation T . It is complete in the case of periodically non-decreasing functions because their paths inherently possess this property.

$$\text{while } (x > 0 \wedge y > 0) \{ \\ \quad x = -8x + 6y; \\ \quad y = 2y; \\ \}$$

Fig. 5. A transformation which is not periodically monotonic.

The example provided in Figure 5 is not periodically monotonic, as the coefficients of iterative values of x and y alternate between positive and negative. However, our technique finds an infinite path within the loop: $(x, y) = (3, 5)$, which is an eigenvector for one of the transformation's eigenvalues: 2.

III. CYCLIC LOOPS

Our definition of a periodic loop characterises loops which display a form of repetitive behaviour. The most elementary representation of this behaviour appears in cyclic loops — those whose values repeat after a certain number of iterations. It is obvious that our technique can be applied to this form of loop, because a cyclic loop of period k must depict the compound transformation T for which $T^k v = v$, a non-decreasing function. What is interesting, however, is that there is an upper bound on the period of such a loop, dependent on its number of variables.

Theorem 1 (Minkowski). *If G is a finite subgroup of $GL(n, \mathbb{Z})$, the group consisting of invertible $n \times n$ matrices with integer entries, then G is isomorphic to a subgroup of $GL(n, \mathbb{Z}_p)$ for all primes $p \neq 2$.*

Because $GL(n, \mathbb{Z}_p)$ is finite, Theorem 1 implies that the group $GL(n, \mathbb{Z})$ contains, up to isomorphism, finitely many subgroups (a detailed presentation of the mentioned theorem and its corollaries awaits the interested reader in [7]). Note the implication on the current topic: a cyclic affine loop transformation in n variables is represented as an $n \times n$ integer matrix T for which $T^k = I$ for some $k \geq 0$. This T is invertible ($TT^{k-1} = I$) and generates a finite cyclic subgroup of $GL(n, \mathbb{Z})$. Since $GL(n, \mathbb{Z})$ has only finitely many subgroups, there can be only finitely many possible values for the period of a cyclic affine loop in n variables.

Specifically, the possible periods of an affine loop in n variables are those k which satisfy $W(k) \leq n$, where, if the prime power decomposition of k is $k = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_t^{\alpha_t}$,

$$W(k) = \begin{cases} \sum_{i=1}^t (p_i - 1) p_i^{\alpha_i - 1} - 1 & \text{if } p_1^{\alpha_1} = 2, \\ \sum_{i=1}^t (p_i - 1) p_i^{\alpha_i - 1} & \text{otherwise.} \end{cases}$$

For the sake of clarity, Table I gives the largest possible period $K(n)$ for cyclic loops with $n = 1, \dots, 10$ variables.

The existence of an upper bound $K(n)$ implies that if $R_T(k)$ is not recurrent for any $k \leq K(n)$, then the loop described by T is not cyclic. Thus the technique described in Section II is complete for cyclic loops, and only a finite number of sets need be checked for recurrence. Practically, upper bounds are explicitly defined for small values of n , and a primitive value of n^2 or $(n + 1)^2$ is used for relatively small n , as the upper

n	$K(n)$
1	0
2	6
3	6
4	12
5	12
6	30
7	30
8	60
9	60
10	120

TABLE I
MAXIMAL POSSIBLE PERIOD $K(n)$ FOR A CYCLIC LOOP WITH n
VARIABLES.

bound for general n , $K(n) \leq n!2^n$, as described in [8], is not feasible for our iterative technique.

Consider the cyclic loop presented in Figure 6: it gives rise to a 3×3 integer transformation matrix, and thus has a period of at most 6. Executing our algorithm returns the infinite path generated by the input value $(1, -2)$, with period 4:

$$\begin{aligned} (1, -2) &\rightarrow (1, -1) \\ &\rightarrow (2, -2) \rightarrow (2, -3) \\ &\rightarrow (1, -2). \end{aligned}$$

```
while (x > 0) {
  x = x + y + 2;
  y = -2x + y + 3;
}
```

Fig. 6. A cyclic loop, with period 4.

One final note on this topic is that if more complex guard constraints are allowed in the loop’s guard condition, infinite paths may often appear as cyclic paths. Specifically, a condition of the form $(x > a \wedge x < b)$ implies that an infinite path through the loop may only traverse a finite number of values for the x variable. Thus, such a path must at some point return to its original value.

IV. JPF IMPLEMENTATION

Java Pathfinder (JPF) is something of a changeling when one considers model checking software. Designed to be an extendable, modular verifier of Java bytecode, it’s skeletal nature gives rise to a powerful, practical tool with a relatively steep learning curve. Our search for infinite paths has been implemented as a feature of JPF’s symbolic execution library, *JPF-symbc*.

The JPF-symbc extension explores possible execution paths through a given Java program, and can provide example values for each path it traverses. A loop block, however, lacks an explicit path description, and thus cannot be handled by JPF-symbc in the same manner as simple choice branches. By making use of the algorithm generated from the results in Section II, the authors have successfully implemented an extension which can often provide example values, if applicable,

for the two basic branches of a loop: terminating and non-terminating. At the very least the loop can be bypassed during symbolic execution and its guard condition, which must hold for the program to exit the loop block, added to the relevant branch’s symbolic path description. This section provides a brief overview of this implementation.

Due to the fact that JPF is in itself a Java virtual machine, which executes on bytecode, loop blocks must be reconstructed from their machine instruction representations. The authors’ extension was created with the ability to identify encoded loops, irrespective of their form, allowing *for*, *while* and *do while* loops with complex guard conditions to be decoded, as well as nested combinations thereof. The ‘jump’ instructions which create the loop’s behaviour can be algorithmically parsed to reconstruct the loop’s guard condition; a simple example of an encoded guard condition is presented in Figure 7.

```
1: push x
2: if x ≤ 0 → inst. 5
3: push y
4: if y > 0 → inst. 7
5: push z
6: if z ≤ 0 → inst. 9
7: success
8: ...
9: failure
```

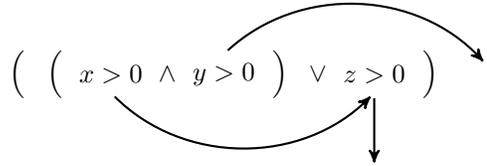


Fig. 7. The bytecode generated by the guard condition $((x > 0 \wedge y > 0) \vee z > 0)$, as well as its graphical representation. An arrow is drawn above or below a constraint to represent the jump which occurs upon its satisfaction or failure respectively; in this case only the arrows relating to the bytecode are shown.

The metadata of an interpreted loop, which describe the indices which border sections of the loop block, then allow JPF to detour around the loop block, if desired. The interpreted loop can also be matched to a concrete Java object and examined in more detail; in our case, the affine loop object. An example of a reconstructed affine loop object, which contains the information gathered from the loop’s bytecode, is provided in Figure 8. Any form of loop can be handled similarly. Because the termination of a loop is dependent upon only the variables present in the loop’s guard condition and those which affect their values, a set of symbolic *loop variables* must be built from the loop block. This process creates a complete symbolic representation of the affine loop (of course, a similar representation can be obtained for any loop, symbolic library permitting), containing both the variables’ guard constraints and update transformations. The latter is simply stored as the integer matrix which describes the loop’s transformation,

```

while (x > 0 ∧ y > 0 ∨ z > 0) {
  x = x - 2y + 4;
  y = x + 2y;
  z = z - 2;
}

```

JPF Reconstruction:

```
(x > CONST_0 && y > CONST_0) || z > CONST_0
```

Loop variables:

```
x = ((x + (y * CONST_-2)) + CONST_4)
```

```
y = (x + CONST_4)
```

```
z = (z + CONST_-2)
```

T :

```
[ 1, -2,  0,  4]
```

```
[ 1,  0,  0,  4]
```

```
[ 0,  0,  1, -2]
```

```
[ 0,  0,  0,  1]
```

Fig. 8. A loop's reconstruction via JPF. Note how the transformation matrix T is built by parsing the variables' update expressions sequentially.

upon which linear algebra can be performed. This algebraic representation simplifies some calculations, such as that of composite transformations such as T^k .

It is worth noting the practical form of the algorithm obtained from Section II:

$$\mathbf{v} \in R_T(k) \Rightarrow T^k(\mathbf{v}) \in R_T(k),$$

where

$$R_T(k) = \{\mathbf{v} : T^l \mathbf{v} > 0 \wedge T^{l+k} \mathbf{v} \geq T^l \mathbf{v}, \forall 0 \leq l < k\},$$

is transformed into a condition which can be understood by a satisfiability checking library linked to `jpf-symbc`:

$$\begin{aligned} \{\mathbf{v} : T^l \mathbf{v} > 0 \wedge T^{l+k} \mathbf{v} \geq T^l \mathbf{v} \forall 0 \leq l < k \\ \wedge (T^{2k} \mathbf{v} < T^k \mathbf{v} \vee T^{1+2k} \mathbf{v} < T^{1+k} \mathbf{v} \\ \vee \dots \\ \vee T^{3k-1} \mathbf{v} < T^{2k-1} \mathbf{v})\} = \emptyset. \end{aligned}$$

The above form searches attempts to locate a \mathbf{v} such that $\mathbf{v} \in R_T(k)$ and $T^k(\mathbf{v}) \notin R_T(k)$.

V. TERMINATION ASSERTION

The approach described in Section II was developed to search for infinite paths within loops. When viewed from a different perspective, however, it can be used to attempt to prove the termination of an affine loop, and is an effective heuristic for common terminating loops.

If the loop described by the affine transformation T continues to execute as long as the variable vector $\mathbf{v} = [v_1, \dots, v_n]^T$ satisfies $\mathbf{v} \succ 0$, then proving T terminating is equivalent to

proving that, for any valid initial value of \mathbf{v} , some v_i will eventually violate its guard constraint.

Note how this approach differs from that taken when attempting to prove non-termination. Most importantly, non-termination requires the continuous satisfaction of constraints which encompass every loop variable, while to prove termination one need only prove that some property holds for a single variable which causes it to violate the iteration conditions of a system eventually.

In our particular case, we attempt to show that a loop variable is periodically decreasing, i.e., that some $k > 0$ exists such that $T_i^k v_i \prec v_i$ for all possible initial values of \mathbf{v} , where T_i depicts the restriction of T to the i 'th variable. Converted to a form which can be parsed by a satisfiability checker, we obtain

$$\begin{aligned} \exists k > 0 \text{ s.t., for some } i \in [1, n] : \\ \{v_i : \mathbf{v} \succ 0 \wedge T_i^k v_i \succ v_i\} = \emptyset. \end{aligned}$$

This check is sufficient for those common loops for which some index is iteratively decremented to ensure the termination of the loop; take as example the loop displayed in Figure 9, which decreases over a period of 1.

```

while (x > 0 ∧ y > 0) {
  x = x - y;
  y = y + 1;
}

```

Fig. 9. An affine loop whose decreasing behaviour over a period of 1 can be asserted by the technique described in Section V.

VI. FUTURE RESEARCH

There are two avenues of further research which the authors wish to pursue: firstly, to make use of the algebraic foundation awarded by the current JPF implementation to enhance the termination-checking algorithm, using principles from linear algebra and dynamical systems; and secondly, to apply an approach inspired by the one presented in this text to program blocks of different forms, which may include non-deterministic updates.

VII. CONCLUSION

We have presented a technique for finding infinite paths within affine loops; this technique is complete for periodically non-decreasing loops, in that if an infinite path within such a loop exists, it will be located. In the case of cyclic and single-variable loops, an upper bound exists on the number of possible periods which should be checked. We have implemented this technique with the aid of JPF and its symbolic execution library, `JPF-symbc`, and the implementation can assert the termination of or generate infinite paths for a number of practical affine loops.

VIII. ACKNOWLEDGEMENTS

This work was supported by the Stellenbosch Unit of the Telkom-Nokia-Siemens-Networks Centre of Excellence in ATM and Broadband Networks and their Applications.

REFERENCES

- [1] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Commun. ACM*, vol. 54, pp. 88–98, May 2011.
- [2] A. Tiwari, "Termination of linear programs," in *Computer Aided Verification*, 2004, pp. 70–82.
- [3] M. Braverman, "Termination of integer linear programs," in *Computer Aided Verification*, 2006, pp. 372–385.
- [4] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Verification, Model Checking and Abstract Interpretation*, 2004, pp. 239–251.
- [5] A. R. Bradley, Z. Manna, and H. B. Sipma, "Termination analysis of integer linear loops," in *International Conference on Concurrency Theory*, 2005, pp. 488–502.
- [6] A. Podelski and A. Rybalchenko, "Software Model Checking of Liveness Properties via Transition Invariants," Max-Planck-Institut für Informatik, Tech. Rep., 2003.
- [7] J. Kuzmanovich and A. Pavlichenkov, "Finite groups of matrices whose entries are integers," *Amer. Math. Monthly*, vol. 109, pp. 173–186, 2002.
- [8] S. Friedland, "The Maximal Orders of Finite Subgroups in $GL_n(\mathbf{Q})$," *Proc. Amer. Math. Soc.*, vol. 125, pp. 3519–3526, 1997.

Kevin Durant received his honours degree in 2010 and is currently studying towards his Master of Science degree in Computer Science at Stellenbosch University.

Willem Visser holds a PhD in Computer Science from the University of Manchester (1998) and is currently a Full Professor at Stellenbosch University. He is on the Editorial Board of *ACM Transactions on Software Engineering and Methodology* and on the steering committee of the Automated Software Engineering conference.