A Termination Checker for Isabelle Hoare Logic

Jia Meng[1], Lawrence C. Paulson[2], and Gerwin Klein[3]

[1] National ICT Australia jia.meng@nicta.com.au
[2] Computer Laboratory, University of Cambridge lp15@cam.ac.uk
[3] National ICT Australia gerwin.klein@nicta.com

Abstract. Hoare logic is widely used for software specification and verification. Frequently we need to prove the total correctness of a program: to prove that the program not only satisfies its pre- and post-conditions but also terminates. We have implemented a termination checker for Isabelle's Hoare logic. The tool can be used as an oracle, where Isabelle accepts its claim of termination. The tool can also be used as an Isabelle method for proving the entire total correctness specification. For many loop structures, verifying the tool's termination claim within Isabelle is essentially automatic.

1 Introduction

For many critical systems, such as operating systems kernels, testing is not adequate to ensure correctness. Formal methods have become popular in research as well as industry. There are several approaches to verifying program correctness. For example, Hoare logic has been used to specify a program's pre- and post-conditions; using logical deduction, one can prove the program meets its specification. A program that fails to terminate satisfies its post-condition by default, so *total correctness* requires a proof that the program terminates.

Total correctness proofs are complicated and require a lot of human effort. A different approach is to focus on the termination property of a program, which can usually be automated with some human assistance. An example is the `Terminator` program developed at Microsoft Research [3, 8], which checks whether a C program terminates.

A proof of termination is useful, but it does not guarantee that a program does what it is supposed to do. Full specification and verification is still what we are aiming at, and the most important formalism for that purpose is Hoare logic. Our current work is part of a larger project to verify the functional correctness of the L4 operating system micro kernel. An automatic termination checker can reduce manual work. Since `Terminator` is not publicly available, we have implemented a termination checker in the spirit of `Terminator` and have integrated this tool into Isabelle [6]. The tool can be used by Isabelle's Hoare logic to prove total correctness specifications. In this paper, we concentrate on termination of `WHILE` constructs.

Although our termination checker and `Terminator` are based on the same technology, we have a different emphasis. In addition to implementing the termination tool, we also investigate how we can have the tool's results used by

Isabelle's Hoare logic. The termination tool is based on model checking and it returns a set of well-founded (WF) relations for each cyclic path. However, a total correctness specification in Hoare logic requires one single WF relation (the variant) for each looping construct. An explicit variant represents evidence of termination, but this variant is not given as a result by either `Terminator` or our tool, which are both based on the model checking technology. Much of our investigations concern how to make the results from the termination tool usable to Isabelle. As far as we know, our tool is the first integration of a model-checking-based termination tool with an interactive prover, and our work is the first integration of the terminator technology to Hoare logic.

In addition, we invented an optimization for our termination checker (§4.3) to meet our particular requirements on the generated WF relations. Finally, Podelski and Rybalchenko [8] have proved the mathematical theory behind `Terminator`. In order that we can use the tool in Isabelle, we have had to formalize their proofs in Isabelle, and this is the first formalization of the proofs in any interactive prover.

Another method for termination analysis is to translate imperative programs into functional programs so that one can prove the termination property of an imperative program by proving its functional counterpart terminates [1]. However, we decided to adopt another approach, which is based on disjunctively well-founded relations. This approach is comparatively novel and is a promising method that has been applied to full C programs.

Paper outline. We first present background information on Isabelle's Hoare logic and termination requirements of programs (§2). We then describe how we have implemented our termination checker (§3). Subsequently, we illustrate the two approaches we have used to integrate the termination checker into Isabelle (§4 and §5). In order to show how we can use our termination checker with Isabelle, we give some examples (§6). Finally, we conclude the paper (§7).

2 Isabelle and Termination Properties

2.1 Hoare logic in Isabelle

We base our work on Norbert Schirmer's implementation of Hoare logic in Isabelle/HOL [9]. Schirmer has designed a small but expressive imperative language, called SIMPL, with recursive procedures. He defines an operational semantics for SIMPL and derives a sound and complete Hoare logic. The Hoare logic implementation includes an automated verification condition generator (`vcg`). In other work [10], we have provided an Isabelle front-end for mapping C into SIMPL.

Since SIMPL treats procedures, the Hoare triple format that we show in examples later also mentions procedure environments, Γ. Partial correctness

is written $\Gamma \vdash \{\!|P|\!\} C \{\!|Q|\!\}$; total correctness is written with subscript t, as $\Gamma \vdash_t \{\!|P|\!\} C \{\!|Q|\!\}$. Partial correctness means if C is executed in a state where P is true, and *if* C terminates, then it will end in a state where Q is true. Total correctness includes a termination requirement: if C is executed in a state satisfying P, then it will terminate in a state satisfying Q.

For proving total correctness, there are two approaches. One approach is to separate partial correctness from termination, as with the following rule

$$\frac{\Gamma \vdash \{\!|P|\!\}\ C\ \{\!|Q|\!\} \quad \Gamma \vdash_t \{\!|P|\!\}\ C\ \{\!|\top|\!\}}{\Gamma \vdash_t \{\!|P|\!\}\ C\ \{\!|Q|\!\}}$$

where $\top$ is logical truth. The second premise says that the program C started in a state satisfying P will terminate. If we assert it without giving any evidence, we have implemented an oracle that accepts an external claim of termination. Here we follow the convention that unmentioned variables do not change their values.

The second approach is to use Hoare logic rules for total correctness. These rely on the concept of a well-founded (WF) relation: one that has no infinite descending chains. In this paper we do not consider the recursive procedure call of SIMPL, which makes `WHILE` the only looping construct. Schirmer uses sets of states to formalize all predicates, such as pre- and post-conditions and the loop's boolean expression. For the verification condition generator, a typical `WHILE` statement is annotated with an invariant, which is again a set of states, and a variant, which is some WF relation. This means the `WHILE`-rule for total correctness in Schirmer's Hoare logic is the following:

$$\frac{\begin{array}{c}\forall \sigma.\ \Gamma \vdash_t \{\!|\ \{\sigma\} \cap I \cap b\ |\!\}\ C\ \{\!|\ \{t \mid (t,\sigma) \in V\} \cap I\ |\!\} \\ P \subseteq I \qquad I \cap -b \subseteq Q \qquad \texttt{wf}\ V\end{array}}{\Gamma \vdash_t \{\!|P|\!\}\ \texttt{WHILE}\ b\ \texttt{INV}\ I\ \texttt{VAR}\ V\ \texttt{DO}\ C\ \{\!|Q|\!\}}$$

The first premise fixes the pre-state σ and requires that the loop body C decreases the variant while maintaining the invariant I. The set $\{t \mid (t,\sigma) \in V\}$ consists of all post-states t such that (t,σ) are in the variant V. If V is a WF relation, the loop must terminate. This approach is good for generating full termination proofs, because the variant gives explicit evidence for termination.

The rest of this paper shows how we can integrate externals tools into Isabelle/HOL to produce the variants mentioned above and how to prove them well-founded. This reduces the manual proof effort for total correctness goals, either by oracle or by proof fully verified in Isabelle/HOL.

2.2 Termination Properties

Before we explain the termination properties, it is helpful to have a brief review of the transition system that is usually used as an abstraction of programming languages. For more information, we refer readers to the book by Manna [5].

Each program has an associated *transition system*. A transition system consists of four parts: Π, Σ, R and Θ.

- Π is a finite set of *state variables*, which consists of program variables that appear in the program statements and also control variables, such as the program counter (PC).
- Σ is a set of *states*. Each state s is an interpretation of Π, which assigns values to each variable in Π. For example, $x\ s$ is the value of variable x in the state s.
- R is a finite set of *transitions*. For a deterministic program, a transition τ is a function $\Sigma \to \Sigma$. If a transition τ leads a state s to another state s' (written as $s \xrightarrow{\tau} s'$), then s' is reachable from s. We say that s and s' are pre- and post-states of τ.
- Finally Θ is the *initial condition*, which is a set of initial states

Each transition τ is characterized by its *transition relation* ρ_τ, where $(s', s) \in \rho_\tau$ if and only if s' is reachable from s by τ. In addition, we can extract one or more transition relations from each program statement. For example, if we have a `WHILE` statement at program location L as `WHILE(x > 0){x = x -1;}`, then its transition relations are $\{(s', s) \mid\ PC\ s = L \wedge x\ s > 0 \wedge x\ s' = x\ s - 1 \wedge PC\ s' = L\}$ and $\{(s', s) \mid\ PC\ s = L \wedge x\ s \leq 0 \wedge x\ s' = x\ s \wedge PC\ s' = M\}$, where M is the exit location of the `WHILE` construct.

This set notation for transition relations is sometimes abbreviated by a logical formula: the value of a variable in the pre-state is represented directly by the variable name and the value in the post-state is represented by the primed version of the variable name. When the PC value is understood from the context, its pre- and post-values are also ignored. For example, the first transition relation above can be abbreviated to $x > 0 \wedge x' = x - 1$.

A *computation* is a possibly infinite sequence of states $s_0, s_1 \ldots$, such that s_0 is in an initial state and each s_{i+1} is reachable from s_i via some transition.

A *path* in a transition system is a sequence of transitions $\pi = \tau_1 \ldots \tau_n$, such that the PC's value in the post-state of τ_i is the same as the value in the pre-state of τ_{i+1}. There is a path transition relation for each path, which is just the relational composition of each consecutive transition relation involved in this path. The path above is cyclic if the PC value in the pre-state of τ_1 is the same as the value in the post-state of τ_n.

A program is terminating if there is no infinite computation. Theoretically, this can be proved by showing that there is a WF relation T, such that each consecutive pair of states s_i and s_{i+1} has $(s_{i+1}, s_i) \in T$. However, it is often

too difficult to find one single WF relation T for this purpose. Podelski and Rybalchenko [8] have proved that it is sufficient to find a finite set of WF relations $T = \{T_1, \ldots, T_n\}$ to show the program is terminating, provided we can prove $R_I^+ \subseteq T_1 \cup \ldots \cup T_n$, where R is the program's transition relation, R^+ is the transitive closure of R and R_I^+ is a reachable subset of R^+. This means for each reachable path, its path transition relation must be a member of some T_i of T.

Since each non-cyclic path induces a WF relation, to show the program is terminating, we only need to show there is a WF relation T_i for each reachable cyclic-path. Cook et al. have shown [3] that program termination checking can be translated into program reachability analysis. For this to work, auxiliary program variables are introduced and the relations between them and the original variables are established to mimic the transition relations of the program. A designated program location `ERROR` is used: it is only reachable if there is no WF relation T_k such that the post-state s' and pre-state s of a cyclic path's transition relation has $(s', s) \in T_k$. If this happens, one can try to find a WF relation for that cyclic path. If no WF relation can be found, then the program is reported as possibily non-terminating. If a WF relation can be generated for the path, then the process continues with other cyclic paths, until `ERROR` is really not reachable, which means all cyclic paths are covered by some WF relations.

2.3 Using a Termination Tool for Generating Variants

Although we can use the termination tool to generate WF relations for all cyclic paths, these relations cannot be used as variants for Hoare logic.

First, there may be nested `WHILE` constructs. The variant V for the outer `WHILE` has to be one single WF relation so that the pre-state s and the post-state s' of the body transition relation satisfy $(s', s) \in V$, regardless how many times the inner `WHILE` are entered and whether they are entered at all. This means V has to satisfy multiple cyclic paths. We can use the termination tool to generate WF relations to cover all these cyclic paths, but the variant must be one single WF relation, and we believe that automatically combining multiple WF relations into one is impossible in general.

Second, even if a program has no nested `WHILE`s, there may be more than one (cyclic) path and hence more than one path transition relation between the start and the end of the `WHILE` loop. Each transition relation has to be a conjunction of positive atomic formulae, and each atomic formula is a mathematical assertion over state variables. This is because we use an external ranking function generator to synthesis WF relations and the tool only accepts a transition relation expressed as a conjunctive formula without negations. Consequently, if the guard of the `WHILE` has disjunctions or negations or the body has `IF` constructs, then we will have multiple path transition relations. We again need to combine multiple WF relations into one.

Instead of generating a single WF relation, we could investigate another approach that tries to construct a termination argument from these WF relations. However, our work aims to use the termination checker to support proofs performed in Isabelle Hoare logic, and the total correctness rule requires the variant as the termination argument.

3 Implementing a Termination Checker

We have implemented a termination checker that checks C programs involving integer variables, in the spirit of `Terminator` [3], in the sense that we check the termination of the entire program by generating one or more WF relations for its cyclic paths. Currently, we only use the tool to check cyclic paths produced by `WHILE` constructs and the tool does not handle pointers. Moreover, we use two external tools: `Blast` and `Rankfinder`.

3.1 Using `Blast` and `Rankfinder`

`Blast` [4] is a model checker for C programs. It checks that software behavior meets its specification using lazy abstraction. For our purpose, we use `Blast` to check if a designated location called `ERROR` is reachable. If `ERROR` is not reachable, then `Blast` reports the program is safe. If `ERROR` is reachable, then `Blast` returns a trace: a sequence of locations from the start of the program to `ERROR`.

Here, we are using the location `ERROR` to signal a possibly non-terminating cyclic path. If `ERROR` is not reachable, then there is no non-terminating cyclic path. However, if `Blast` reports a trace that leads to `ERROR`, then we can extract a cyclic path from it, and then we can examine if we can generate a WF relation to show the path is terminating, using `Rankfinder`.

`Rankfinder` [7] is a ranking function (a.k.a. measure function) generator. A ranking function is a decreasing function, with a lower bound. Given a transition relation τ, `Rankfinder` tries to synthesize a decreasing ranking function, with two parameters: an integer bound b and a positive integer d that is the minimum decrease of the ranking function during the transition relation. For example, if the transition relation is $x \geq 0 \wedge x' = x - 1$, then the ranking function from `Rankfinder` is x, the bound is 0 and minimum decrease is 1. Each ranking function F induces the well-founded relation

$$\{(s', s) \mid b\ s \leq F\ s \wedge F\ s' \leq F\ s - d\}$$

where s' and s are the post- and pre-states of the transition.

3.2 The Termination Checker

Our tool is closely integrated with Isabelle and it is called via an Isabelle invocation. It works as follows.

1. The C program embedded in SIMPL is extracted to generate a control flow graph. For better performance, we "compact" the flow graph so that only `WHILE` locations and the program entrance point are kept in the graph. All the remaining program locations are removed from the graph by joining the path transition relations. If the pre-condition P of the Hoare specification is non-empty (i.e. there are initial conditions on program variables), then we modify the flow graph to include the initial conditions. Subsequently, we examine each `WHILE` construct in turn, and the order in which we examine the `WHILE` constructs does not matter.
2. For each `WHILE` construct, writing its program location as L, check the termination of all cyclic paths that start from and finish at L:
   (a) Insert the already-generated WF relations for L into the C program and generate a text file, then call `Blast`. Initially no WF relation is generated, so nothing is inserted.
   (b) If `Blast` reports the program is safe, i.e. `ERROR` is not reachable, then move to the next `WHILE` construct. If `Blast` reports an error trace, then we extract the cyclic paths from the trace and calculate the reachable transition relations. We then call `Rankfinder` to generate a ranking function for each transition relation. If `Rankfinder` succeeds, then use the newly generated WF relations to modify the C program and re-run `Blast`. If `Rankfinder` cannot generate a well-founded relation for a transition relation, then the program is reported as possibly non-terminating.
3. If `ERROR` is no longer reachable, `Blast` will report the program is safe. We can then move on to the next `WHILE` construct if available.
4. If for each `WHILE` construct, its cyclic paths are reported to be terminating, then the entire program is terminating; the generated WF relations are also reported. Otherwise, the program is reported as possibly non-terminating.

4 Integrating the Termination Checker into Isabelle

Our termination checker has been used as a tool for Isabelle, both as an oracle and as a proof method. Isabelle's oracle mechanism accepts an external tool's result without verifying it. When used as a proof method, the result is used to create an Isabelle proof that is verified through Isabelle's kernel.

4.1 Integration as an Oracle

Recall that a total correctness goal can be proved separately as a partial correctness goal and a termination goal (§2.1). When used as an oracle, we only use the tool to prove the second subgoal, namely the program is terminating, if started from a state satisfying P. We do not need to generate variants in this case. Therefore, if the tool reports the program is terminating, the second subgoal is removed from the proof state.

4.2 Integration as an Isabelle Proof Method

Using the termination checker as an Isabelle oracle gives us a quick answer to whether the program is terminating. Of course, using the tool as an Isabelle proof method would yield greater confidence. This requires us to use the tool to generate a variant for each `WHILE` construct. Moreover, we would like the variant to be as simple as possible. The form of the variant generated depends on the complexity of the program, as well as the WF relations generated for `WHILE` constructs.

For this purpose, we divide the WF relations for each `WHILE` construct (W) into two sets: T_{in} is generated for cyclic paths that do not leave W and T_{out} is generated for cyclic paths that leave W and re-enter. This is because a variant for W is essentially a set of transition relations of paths that do *not* leave W. Therefore, if we define a variant using a relation that does not include any path that leaves W, we only need T_{in} for WF proofs. T_{out} is needed when we try to prove the entire program R is WF, since we need to prove R^+ is WF and R^+ contains paths that leave W.

In this section, we describe the form of the variants generated and will informally explain why they are variants and why they are WF. We will show some formal proofs in the next section.

Programs with No Nested Loops If a program has a single loop, then the variant generated only depends on the number of WF relations in T_{in} because we are not concerned about the paths leaving and re-entering the `WHILE` (W). There are two cases to consider.

First, if there is only one ranking function F generated, then we generate an Isabelle measure function M from it. The difference between F and M is that F involves integers whereas M uses natural numbers only, but this is easily dealt with.

Hoare logic requires us to prove that the loop body decreases the variant. More formally, V must be a set containing all the post- and pre-states pairs of the loop. We can indeed prove that the transition relations of each cyclic path starting and finishing at location W form a subset of V.

Second, if there is more than one ranking function in T_{in}, then we define the variant to be the intersection $R_L \cap I$ of the transition relation of the loop and the invariant of the loop. Frequently we can use R_L as the variant, which is weaker. The use of the invariant[1] is important when reachability becomes a concern (§4.4). As there is only one `WHILE` construct, R_L does not need to mention its PC value. As an example, consider the following C program.

```
WHILE (x > 0 || p > 2){x = x - 1; p = p - 2;}
```

[1] Currently, invariants are generated manually, but we plan to incorporate automatic invariant generation in the future.

Its R_L is

$$\{(s', s) \mid x\ s > 0 \wedge x\ s' = x\ s - 1 \wedge p\ s' = p\ s - 2 \vee \\ p\ s > 2 \wedge x\ s' = x\ s - 1 \wedge p\ s' = p\ s - 2\}$$

Obviously, the transition relation is a variant, and we can prove (see §5) that R_L is well-founded.

Programs with Nested Loops This is the complicated case. We generate the variant for each `WHILE` in turn. The form of the generated variant also depends on the complexity of the `WHILE` construct.

If there is only one ranking function F in T_{in}, then we generate its corresponding Isabelle measure function M as above.

If there are multiple ranking functions, then the variants are defined in terms of transition relations. Since there are multiple `WHILE` loops, the transition relation must mention PC values. There are two cases to consider.

First, if the `WHILE` construct with location L has no nested inner loop, then we define the variant to be

$$V = \mathit{fix\_pc}\ L\ (R_L \cap\ I)$$

where R_L, I are transition relation and invariant of the `WHILE` construct and the definition of *fix_pc* is

```
definition
  fix_pc :: "int ⇒((α * int) * (α * int)) set ⇒(α * α) set"
  where "fix_pc pc R = {(s',s). ((s',pc),(s,pc)) ∈R}"
```

The function *fix_pc* removes the dependence on the PC by restricting a relation to the given PC value. Again, R_L can replace $R_L \cap\ I$ sometimes.

Second, if the loop has inner nested loops, then we define its variant V as

$$V = \mathit{fix\_pc}\ L\ R^+$$

where R is the transition relation of the entire program. The formula on the right hand side is indeed a variant, because any path that starts from and finishes at the `WHILE` with location L must have its corresponding transition relation ρ as a subset of R^+, i.e. $\rho \subseteq R^+$. In addition, ρ must be a set containing tuples of the form $((s', pc'), (s, pc))$, where $pc = L$ and $pc' = L$. We will discuss the well-foundedness property of V in section 5.

4.3 An Optimization

The complexity of the WF proofs for variants largely depends on the number of WF relations our tool generates for the `WHILE` constructs. If a `WHILE` construct

is shown terminating using a set of WF relations, then it may also be possible to find another smaller set of WF relations that does the job. Suppose we have two WF relations T_1 and T_2, which show the termination of two cyclic paths π_1 and π_2 using $\rho_1 \subseteq T_1 \wedge \rho_2 \subseteq T_2$, where ρ_1 and ρ_2 are the path transition relations for the two paths. Then if we can generate a weaker WF relation S such that $\rho_1 \subseteq S \wedge \rho_2 \subseteq S$, then we can replace T_1 and T_2 by S. Please note, in general we cannot derive S by simply making a union of the two WF relations, since the result of the union may not be well-founded.

For each `WHILE` construct, our tool attempts to generate a WF relation when a possibly non-terminating cyclic path is found. Therefore, the order in which the WF relations are generated depends on the order in which these cyclic paths are detected. Since we use `Blast` to detect these cyclic paths, we have no control over its searching strategy and so we cannot ask for any specific paths to be reported first.

For a `WHILE` construct W that has one or more inner loops, some of W's cyclic paths (i.e. those start from and finish at W) enter inner loops (call them P_1) while some do not (call them P_2). It may happen that there is something decreasing along the execution of W, regardless whether any of W's inner loops are entered. More precisely, there may be a set T with one or more WF relations that cover paths from both P_1 and P_2. This means, we may be able to generate T without entering any of W's inner loops. We call this set of WF relations the *global* WF relations for W, since it exists regardless of the inner loops' behaviour.

Suppose there is one inner `WHILE` U of W, and the path transition relation from W to U is ρ_1, the path transition of U loop is ρ_u and the path transition relation from U back to W is $\neg b \wedge \rho_2$, where b is the guard of U, i.e. the condition when U is entered. The path transition relation from W back to W is

$$\rho = (\neg b \wedge \rho_2) \; o \; \rho_u^+ \; o \; \rho_1.$$

To generate the required T, we generate WF relations for $\rho_A = \rho_2 \; o \; \rho_1$. This path corresponds to a program W', which is W with U completely commented out. We do not generate WF relations for $\rho_B = (\neg b \wedge \rho_2) \; o \; \rho_1$. since this path simulates the effect that the guard of U is not true. Clearly ρ_A is weaker than ρ_B and if a WF relation can be used for ρ_A, then it can be used for ρ_B. Nevertheless, our aim is to have the generated WF relations to work for ρ as well; if the WF relation for ρ_B is too strong, then it may not work for ρ.

Of course, a given `WHILE` construct W may not have this global set T of WF relations. As a result, we still need to generate WF relations for all cyclic paths that enter inner loops. The advantage of generating T is that some relations in T may make it unnecessary to generate new WF relations for some cyclic paths, thereby reducing the number of relations generated. We have implemented this optimization in our termination checker.

4.4 An Issue of Reachability

As we have mentioned, the technique of termination checking works by ensuring all *reachable* cyclic paths are terminating. When we use `Blast` to check the reachability of the `ERROR` location, the notion of reachable cyclic path is already present implicitly with `Blast`. However, sometimes we need to express reachability explicitly, for two reasons.

The first one is for `Rankfinder` to generate WF relations. For example, we may have a program

```
y = 2; WHILE (x > 0){x = x - y;}
```

Without knowing $y > 0$, the transition relation of `WHILE`'s cyclic path is not well-founded and so `Rankfinder` will fail to generate a WF relation for it. To *strengthen* the transition relation, we note that $y > 0$ is an invariant and by adding it to the transition relation of the path, the new relation is indeed WF.

The second reason for including the reachability condition is to have a strong enough variant. There may be non-terminating cyclic paths that do not concern `Blast` since they are deemed to be unreachable, and so `Rankfinder` will not have to generate WF relations for them. However, when defining the variant, which are effectively the transition relations of paths, we need to incorporate in it the reachability condition so that unreachable paths are removed from it.

We have tried several ways of expressing this reachability requirement of loop variants. A simple way is to include the loop's invariant in the variant. For single-looped program, we define the variant as $R_L \cap I$. For an innermost loop, we define its variant as $\mathit{fix\_pc}\ L\ (R_L \cap I)$. We have used this method to prove problems that were not provable otherwise.

If the `WHILE` construct has nested inner loops, its variant can also be strengthened by adding invariants. To discover an invariant can require much thought, and ideally we would use an automatic tool for generating invariants. At present we are using no such tool and have decided to use $\mathit{fix\_pc}\ L\ R^+$ as the variant. This heuristic choice does not affect the soundness of our tool and will not affect the way the tool is used as an oracle. When the tool is used as a proof method, if a non-reachable non-terminating cyclic path is included, then a user will fail to prove that a (non-WF!) path transition is well-founded. This is a signal that the path may be in fact not reachable. The user can then make another attempt: either trust the oracle or try to strengthen the variant by finding a strong invariant of the entire program.

5 Proving Variants being Well-Founded

Having generated the required variants, we need to show their well-foundedness. Showing that the relation defined for V is a variant is a separate task, which requires the users to have found the correct invariants.

When the generated V has the form *measure* M, we can apply Isabelle's existing methods to show it is WF automatically. Otherwise, there are several possibilities:

- A single-loop program: the variant has the form $R_L \cap I$ or R_L.
- A multi-loop program: the variant has the form *fix_pc* L R, where R is either $R_L \cap I$ or R_L.
- In the most complicated case, the variant has the form *fix_pc* L R^+.

The proofs are based on *disjunctively well-founded relations* of Podelski and Rybalchenko [8]. A relation is disjunctively well-founded, if it is the union of finitely many well-founded relations. We need to formalize them for Isabelle proofs to work.

5.1 Proving that R and *fix_pc* L R are Well-Founded

These are the simpler of the three cases. In order to show that R is WF, we need to prove R^+ is WF. We have proved the following two essential theorems in Isabelle:

```
theorem union_wf_disj_wf1:
  "⟦⋀s. s ∈ R ⟹ wf s; r ⊆ ⋃R; finite R⟧ ⟹ disj_wf r"

theorem trans_disj_wf_implies_wf:
  "⟦trans r; disj_wf r⟧ ⟹ wf r"
```

The first theorem characterizes what it means for a relation r to be disjunctively well-founded (*disj_wf*). The second theorem states the crucial result that if a relation is both transitive and disjunctively WF, then it is WF. The Isabelle proof follows the informal argument [8] in using Ramsey's theorem. Using these two theorems, we can prove that R^+ is WF by proving $R^+ \subseteq \bigcup T$, where T is the set of WF relations the tool generates. We can prove that R is well-founded using another Isabelle lemma:

```
"wf (r⁺) ⟹ wf r"
```

Finally, we have used another theorem to prove variants of the form *fix_pc* L R.

```
theorem fix_pc_wf:
  "wf R ⟹ wf (fix_pc pc R)"
```

5.2 Proving that *fix_pc* L R^+ is Well-Founded

This is the complicated case and we have tried two approaches.

In the first approach, we tried to prove R^+ is WF by restricting attention to cyclic paths. In order to restrict R^+ to the transitions of cyclic paths, we have defined the constant *same_pc*:

```
definition
  same_pc :: "((α * int) * (α * int)) set"
  where "same_pc = {((s',pc'),(s,pc)). pc' = pc}"
```

Now $R^+ \cap$ *same_pc* denotes the subset of R^+ concerning cyclic paths. We need to prove that the relation *fix_pc* L R^+ is well-founded. It suffices to show

$$R^+ \cap \textit{same\_pc} \subseteq \bigcup T,$$

where T is the set of generated WF relations. We cannot prove this by induction because the *same_pc* property is not preserved from one transition to the next. To make this approach work, we need to identify an invariant S of R^+, such that we can prove $R^+ \subseteq S$ by induction and then prove $S \cap \textit{same\_pc} \subseteq \bigcup T$.

In the second approach, we attempted to prove that $R^+ \subseteq \bigcup T$ directly. Since R^+ includes both cyclic and non-cyclic paths, we tried modifying the tool to generate WF relations for non-cyclic paths as well. However, we found that $R^+ \subseteq \bigcup T$ still could not be proved by induction, apparently because the induction hypothesis was too weak: the set $\bigcup T$ was too large. We suspect that it is not practical to generate sufficiently strong WF relations for all non-cyclic paths because there are simply too many such paths.

5.3 Automation in WF Proofs

We have implemented several Isabelle proof methods to invoke the termination checker. When the tool generates all the required WF relations and shows the program is terminating, the goal will be modified with variants inserted. The generated WF relations are also proved automatically to be WF. There is also an option to insert the WF relations as theorems to the assumption of the proof goal for users to inspect.

After this step, we can use *vcg* followed by *auto* to finish the proof, if the variants are measure functions. For the variants of the forms R, $R \cap I$, *fix_pc* L R or *fix_pc* L $(R \cap I)$, we have implemented proof methods *check_wf_sw* and *check_wf_mw* to prove their well-foundedness automatically.

6 Examples and Experiments

Our termination tool is still in the early stage of development. At the moment, it does not support pointers or data structures, such as arrays. However, based on our current development, we can easily add in support for arrays, though pointers require more effort.

Users invoke the termination tool via Isabelle methods: *check_termination_oracle* uses the tool as an oracle; *check_termination* and *check_terminationH* construct variants and the latter uses the optimization(§4.3). If variants are generated, users

will need to apply a a few more Isabelle methods to prove the variants being WF. For this step to work, users usually also need to construct invariants. For example, in order to prove the lemma

```
lemma "Γ⊢t {|True|}
  WHILE (x >= 0) INV {|True|}
  DO
  x :== x + 1;; y :== 1;;
  (WHILE (y  <= x ∨ p > 0)  FIX X. INV {|x = X|}
  DO y :== y + 1;; p :== p - 2 OD);;
  x :== x - 2
  OD
  {|True|}"
```

we first apply *check_terminationH* to generate variants and then finish the proof with *vcg*, *auto* and *check_wf_mw*. *check_wf_mw* is an Isabelle method that we have implemented to automatically prove relations WF.

We carried out several experiments on our tool, with the results shown in Table 1. The experiments we ran mainly involved nested `WHILE` loops. The last example is terminating, but because of the lack of invariants, our tool reported it as non-terminating.

Result	*Remark*
proved by oracle	Fibonacci series with two nested `WHILE`s, no invariants
proved by oracle	Factorial with two nested `WHILE`s, no invariants
proved by oracle	Arithmetic exponentiation with two nested `WHILE`s, no invariants
proved by oracle	Example from [2]. Two nested `WHILE`s, no invariants
proved by method	Example lemma shown above
proved by method	Artificial example with two nested `WHILE`s, with invariants
proved by oracle	Artificial example with three nested `WHILE`s, no invariants
`Blast` failed to terminate	Arithmetic exponentiation with three nested `WHILE`s
`Blast` failed to terminate	Artificial example with three nested `WHILE`s
reported as non-terminating	A non-terminating program
reported as non-terminating	A terminating Euclid algorithm, no invariants

Table 1. Termination Tool Experiments

7 Conclusions

Automatic termination checking is too valuable a tool to reserve for the field of automated program analysis. Interactive program verifiers would like to benefit from as much automation as possible. We have shown how techniques designed for automatic termination checking can, in many cases, be incorporated in a Hoare logic proof, with the termination argument made explicit as variant functions in `WHILE` loops. The resulting subgoals can often be proved automatically,

using dedicated proof methods that we have written. In the most complicated loop structures, the information returned by the automated analysis does not appear to be detailed enough to allow the proof to be reproduced in Isabelle/HOL. To handle those cases, we have also implemented an interface to the tool that accepts the termination as an oracle.

In order to meet our objectives, we have formalized Podelski and Rybalchenko's theory of disjunctive well-foundedness [8] in Isabelle/HOL[2], and we have optimized the termination tool to eliminate redundant outputs that would complicated the proofs.

Acknowledgements. The research was funded by the L4.verified project of National ICT Australia.[3] Norbert Schirmer answered our questions about his implementation of Hoare logic, and Ranjit Jhala answered questions about `Blast`. In formalizing the theory of disjunctive well-foundedness, we used an Isabelle proof of Ramsey's theorem contributed by Tom Ridge.

References

1. Jürgen Brauburger and Jürgen Giesl. Approximating the domains of functional and imperative programs. *Sci. Comput. Program.*, 35(2):113–136, 1999.
2. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2005.
3. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 415–426, New York, NY, USA, 2006. ACM Press.
4. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, Lecture Notes in Computer Science 2648, pages 235–239. Springer-Verlag, 2003.
5. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.
6. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.
7. Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.
8. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In Harald Ganzinger, editor, *Proceedings of the Nineteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2004*, pages 32–41. IEEE Computer Society Press, July 2004.
9. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.
10. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.

[2] The proofs will appear in a future technical report.

[3] National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.