Verification Condition Generation and Variable Conditions in Smallfoot

Josh Berdine[1], Cristiano Calcagno[2], and Peter W. O'Hearn[3]

[1] Microsoft Research
[2] ETH Zurich, Imperial College London, and Monoidics Ltd
[3] University College London

Abstract. These notes are a companion to [1] which describe
- the variable conditions that Smallfoot checks,
- the analysis used to check them,
- the algorithm used to compute a set of verification conditions corresponding to an annotated program, and
- the treatment of concurrent resource initialization code.

2012 Introduction

This document presents the variable conditions and checking algorithms as implemented in Smallfoot 0.1 of late 2005. These conditions on the proof rules for concurrency rely on some of the relaxations introduced by Brookes [3] relative to O'Hearn's system [7], and so originally soundness was via Brookes's result. Ian Wehrman and Berdine have since found some cases where these relaxed conditions are unsound, prompting a revisitation of this topic. Brookes [4] and Reddy and Reynolds [9] have recently introduced systems which address these issues while admitting encodings of proofs in O'Hearn's more restrictive system (among other improvements).

In hindsight, while Smallfoot needs more than O'Hearn's system, it does not use the full relaxation of Brookes's original system, in particular retaining concurrency condition 5 below. As a result, the proofs found by Smallfoot appear to be embeddable into either of the recently proposed sound systems, although we do not claim to have a formal proof at this time.

The condition checking algorithms remain non-compositional, and hence uses a whole-program analysis, despite the compositionality of both Brookes's revised system and that of Reddy and Reynolds. This is a result of the necessity of guiding the search for a proof to one which satisfies the occurrence conditions, as opposed to the distinct problem of checking whether a given candidate proof outline satisfies the conditions, or of inferring a valid permissions annotation as in [9].

1 Checking Variable Conditions

1.1 Annotated Programs

Each Smallfoot program determines a resource environment Γ which contains the resource declarations

$$r_i(\vec{x}_i)R_i$$

where $\vec{x}_i$ and R_i are resource r_i's protected variables and invariant; and a procedure environment Δ which contains the procedure declarations

$$f(\vec{p}\,;\vec{v})[P_f]\,C_f\,[Q_f]$$

where procedure f's parameters $\vec{p}$ are passed by reference and $\vec{v}$ by value, and assertions P_f and Q_f are f's pre- and post-conditions. We assume that Γ and Δ are given.

Commands are generated by:

$$\begin{aligned}
E &::= x \mid \mathrm{nil} \mid c \mid E\,\texttt{xor}\,E \\
B &::= E{=}E \mid E{\neq}E \\
S &::= x{:=}E \mid x{:=}E{\rightarrow}t \mid E{\rightarrow}t{:=}E \mid x{:=}\texttt{new}() \mid \texttt{dispose}(E) \\
C &::= S \mid C\,;C \mid \texttt{if}(B)\ \{C\}\ \texttt{else}\ \{C\} \mid \texttt{while}(B)\ [I]\ \{C\} \\
&\quad \mid\ f(\vec{x}\,;\vec{E}) \mid f(\vec{x}\,;\vec{E}) \parallel f(\vec{x}\,;\vec{E}) \mid \texttt{with}\ r\ \texttt{when}(B)\ \{C\}
\end{aligned}$$

1.2 Legal Annotated Programs

Using the following notation

$$\begin{aligned}
\mathit{owned}(\vec{r}) &\stackrel{\text{def}}{=} \bigcup_{i.r_i \in \vec{r}} \vec{x}_i \\
\mathit{var}(\vec{r}) &\stackrel{\text{def}}{=} \mathit{owned}(\vec{r}) \cup \bigcup_{i.r_i \in \vec{r}} \mathit{fv}(R_i)
\end{aligned}$$

where $\vec{r}$ denotes a set of resources, the set of legal annotated programs is restricted by the following constraints:

- In any procedure call $f(\vec{y};\vec{E})$ or region $\texttt{with}\ r\ \texttt{when}(B)\ \{C\}$ the variable f/r must be defined in a procedure/resource declaration.
- In every procedure declaration $f(\vec{p}\,;\vec{v})[P_f]\,C_f\,[Q_f]$, the formal parameters $\vec{p},\vec{v}$ are all distinct.
- The resources $\vec{r}$ must be distinct.
- The protection lists must all be disjoint: $\mathit{owned}(r_i) \cap \mathit{owned}(r_j) = \emptyset$ when $i \neq j$.
- No resource's invariant may have a free occurrence of a variable in a distinct resource's protection list: $\mathit{fv}(R_i) \cap \vec{x}_j = \emptyset$ when $i \neq j$.

1.3 Simplifying Assumptions

Additionally, we assume a pre-processing phase which renames bound variables to satisfy the following simplifying assumptions:

- Bound variables (formal parameters and variables bound by `local`) are distinct from one another, and from global variables.
- For each procedure declaration, variables in the postcondition are not bound (by `local`) in the body.

1.4 Variable Conditions

We define functions $\mathit{var}(-), \mathit{mod}(-), \mathit{req}(-)$ on commands C and procedure names f. Intuitively:

- $\mathit{var}(C)$ is set of variables that C mentions (in the program or specifications, recursively) without protection;
- $\mathit{mod}(C)$ is the set of variables that C may modify without protection (acquiring r protects $\mathit{owned}(\{r\})$);
- $\mathit{req}(C)$ is the set of resources required to be acquired before executing C.

The auxiliary function $\mathit{er}(M, A)$, returning the set of resources that need to be acquired before modifying variables in M and accessing variables in A, is defined as:

$$\mathit{er}(M, A) = \left\{ r \;\middle|\; \begin{array}{l} r(\vec{x})R \in \Gamma \text{ and} \\ (A \cap \vec{x} \neq \emptyset \text{ or } M \cap (\vec{x} \cup \mathit{fv}(R)) \neq \emptyset) \end{array} \right\}$$

The definition proceeds by performing a simple fixpoint calculation to determine the least solution (which is best) of the following equations, ordering by pointwise subset inclusion of functions:

$$\begin{aligned} \mathit{var}(f) &= (\mathit{var}(C) \cup \mathit{fv}(P, Q)) - (\vec{p} \cup \vec{v}) \\ \mathit{mod}(f) &= \mathit{mod}(C) - (\vec{p} \cup \vec{v}) \\ \mathit{req}(f) &= \mathit{req}(C) \cup \mathit{er}(\emptyset, \mathit{fv}(P, Q) - (\vec{p} \cup \vec{v})) \end{aligned}$$

where $f(\vec{p}\,; \vec{v})[P]\, C\, [Q] \in \Delta$, and for commands:

$$\begin{aligned} \mathit{var}(x{:=}\, E) &= \{x\} \cup \mathit{fv}(E) \\ \mathit{var}(x{:=}\, E{\to}t) &= \{x\} \cup \mathit{fv}(E) \\ \mathit{var}(E{\to}t{:=}\, F) &= \mathit{fv}(E) \cup \mathit{fv}(F) \\ \mathit{var}(x{:=}\, \texttt{new}()) &= \{x\} \\ \mathit{var}(\texttt{dispose}(E)) &= \mathit{fv}(E) \\ \mathit{var}(C\,; C') &= \mathit{var}(C) \cup \mathit{var}(C') \\ \mathit{var}(\texttt{if}(B)\ \{C\}\ \texttt{else}\ \{C'\}) &= \mathit{fv}(B) \cup \mathit{var}(C) \cup \mathit{var}(C') \\ \mathit{var}(\texttt{while}(B)\ [I]\ \{C\}) &= \mathit{fv}(I, B) \cup \mathit{var}(C) \end{aligned}$$

$$
\begin{aligned}
\mathit{var}(f(\vec{x}\,;\vec{E})) &= \mathit{var}(f) \cup \vec{x} \cup \mathit{fv}(\vec{E}) \\
\mathit{var}(f(\vec{x}\,;\vec{E}) \parallel f'(\vec{x}'\,;\vec{E}')) &= \mathit{var}(f(\vec{x}\,;\vec{E})) \cup \mathit{var}(f'(\vec{x}'\,;\vec{E}')) \\
\mathit{var}(\texttt{with}\ r\ \texttt{when}(B)\ \{C\}) &= ((\mathit{fv}(B) \cup \mathit{var}(C)) - \mathit{fv}(R_r)) \cup (\mathit{mod}(C) - \mathit{owned}(\{r\}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{mod}(x{:=}\,E) &= \{x\} \\
\mathit{mod}(x{:=}\,E{\rightarrow}t) &= \{x\} \\
\mathit{mod}(E{\rightarrow}t{:=}\,F) &= \emptyset \\
\mathit{mod}(x{:=}\,\texttt{new}()) &= \{x\} \\
\mathit{mod}(\texttt{dispose}(E)) &= \emptyset \\
\mathit{mod}(C\,;C') &= \mathit{mod}(C) \cup \mathit{mod}(C') \\
\mathit{mod}(\texttt{if}(B)\ \{C\}\ \texttt{else}\ \{C'\}) &= \mathit{mod}(C) \cup \mathit{mod}(C') \\
\mathit{mod}(\texttt{while}(B)\ [I]\ \{C\}) &= \mathit{mod}(C) \\
\mathit{mod}(f(\vec{x}\,;\vec{E})) &= \mathit{mod}(f) \cup \vec{x} \\
\mathit{mod}(f(\vec{x}\,;\vec{E}) \parallel f'(\vec{x}'\,;\vec{E}')) &= \mathit{mod}(f(\vec{x}\,;\vec{E})) \cup \mathit{mod}(f'(\vec{x}'\,;\vec{E}')) \\
\mathit{mod}(\texttt{with}\ r\ \texttt{when}(B)\ \{C\}) &= \mathit{mod}(C) - \mathit{owned}(\{r\})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{req}(S) &= \mathit{er}(\mathit{mod}(S), \mathit{var}(S)) \\
\mathit{req}(C\,;C') &= \mathit{req}(C) \cup \mathit{req}(C') \\
\mathit{req}(\texttt{if}(B)\ \{C\}\ \texttt{else}\ \{C'\}) &= \mathit{req}(C) \cup \mathit{req}(C') \cup \mathit{er}(\emptyset, \mathit{fv}(B)) \\
\mathit{req}(\texttt{while}(B)\ [I]\ \{C\}) &= \mathit{req}(C) \cup \mathit{er}(\emptyset, \mathit{fv}(I, B)) \\
\mathit{req}(f(\vec{x}\,;\vec{E})) &= \mathit{req}(f) \cup \mathit{er}(\vec{x}, \mathit{fv}(\vec{E})) \\
\mathit{req}(f(\vec{x}\,;\vec{E}) \parallel f'(\vec{x}'\,;\vec{E}')) &= \mathit{req}(f(\vec{x}\,;\vec{E})) \cup \mathit{req}(f'(\vec{x}'\,;\vec{E}')) \\
\mathit{req}(\texttt{with}\ r\ \texttt{when}(B)\ \{C\}) &= (\mathit{req}(C) \cup \mathit{er}(\emptyset, \mathit{fv}(B))) - \{r\}
\end{aligned}
$$

The definitions of *var* and *mod* are as expected, except that CCR statements $\texttt{with}\ r\ \texttt{when}(B)\ \{C\}$ hide accesses and modifications of the variables $\mathit{owned}(r)$ in C. Note that variables in $\mathit{fv}(R_r)$ can only be modified in a critical region for r. Therefore, when computing the external effect of a command $\texttt{with}\ r\ \texttt{when}(B)\ \{C\}$ we can ignore the reads to $\mathit{fv}(R_r)$ since they can never happen in parallel with a write. For *req*, any mention of a variable protected by r, or modification of a variable in r's invariant, causes r to be required; and a CCR for r discharges the requirement of r.

Variable Aliasing Conditions The variable conditions for aliasing follow those of [6] and [5]. The conditions needed to avoid variable (not heap) aliasing are enforced by checking, for every procedure call $f(\vec{x}\,;\vec{E})$ in the program:

- The actual reference parameters $\vec{x}$ are distinct.
- If a global variable z is passed by reference, then f and procedures f calls, recursively, must not read or modify z, or mention it in specifications: $\vec{x} \cap \mathit{var}(f) = \emptyset$.

Variable Conditions for Concurrency The variable conditions for concurrency follow those of [8, 7, 3]. The first two concurrency conditions:

1. Protected variables of r appear only within CCRs for r.
2. Variables appearing in a resource r's invariant can only be modified within CCRs for r.

are checked using the computed $req(-)$. A violation of one of these conditions results in "too large" a required resources set for the offending code, which eventually propogates to the main procedure. Hence, we check that $req(\texttt{main}) = \emptyset$, if it appears. Note that this analysis ignores deadlock due to acquiring an already-held resource, as Smallfoot only proves safety.

The third and fourth concurrency conditions:

3. Only protected variables can be modified in one parallel process and read or mentioned in specifications in another.
4. For each parallel composition $f(\vec{x}\,;\vec{E}) \parallel f'(\vec{x}'\,;\vec{E}')$, $f(\vec{x}\,;\vec{E})$ and the specification of f cannot mention variables modified by $f'(\vec{x}'\,;\vec{E}')$, and vice versa.

are checked using the computed $var(-)$ and $mod(-)$:

$$\begin{aligned} & mod(f(\vec{x}\,;\vec{E})) \cap (fv(P', Q') \cup var(f'(\vec{x}'\,;\vec{E}'))) = \emptyset \\ \text{and} \quad & mod(f'(\vec{x}'\,;\vec{E}')) \cap (fv(P, Q) \cup var(f(\vec{x}\,;\vec{E}))) = \emptyset, \\ \text{where} \quad & f(\vec{p}\,;\vec{v})[P]\,C\,[Q], f'(\vec{p}'\,;\vec{v}')[P']\,C'\,[Q'] \in \Delta. \end{aligned}$$

The final concurrency condition is a property of program proofs, not of annotated programs themselves:

5. Whenever a CCR is symbolically executed, the pre and post states cannot mention variables modified by other processes.

Note that according to the inference rule for CCRs, entering a CCR adds the resource invariant to the current precondition. Also, it may be that processes running in parallel with the one executing the CCR under consideration modify variables appearing in the added invariant. For this reason, we introduce a further analysis which computes, for each procedure f, the set $par(f)$ of procedures that might run in parallel with f:

1. $par(f) \supseteq \{f'\}$ for all occurrences of $f(\vec{x};\vec{E}) \parallel f'(\vec{x}';\vec{E}')$ or $f'(\vec{x}';\vec{E}') \parallel f(\vec{x};\vec{E})$ in the program;
2. $par(f') \supseteq par(f)$ for all occurrences of $f'(\vec{x}'\,;\vec{E}')$ or $f'(\vec{x}'\,;\vec{E}') \parallel f''(\vec{x}''\,;\vec{E}'')$ or $f''(\vec{x}''\,;\vec{E}'') \parallel f'(\vec{x}'\,;\vec{E}')$ in C_f.

(As before, we take the smallest set satisfying the above conditions.) The results of this analysis are then used in VCGen to instrument the VCs for CCRs so that any such variables modified by processes in parallel are quantified out of the post states of CCRs during symbolic execution, thereby avoiding bad proofs.

Checking the concurrency conditions essentially classifies each variable into one of the following five classes:

Local variables are declared by `local`, or are procedure value parameters. Their use is unrestricted within their scope.

Process-local variables appear, and are mentioned in specifications, in only one process, and do not appear in any resource invariants. In that process, their use is unrestricted.

Global-constant variables appear in some function and are not local. They cannot be written to but can be read or appear in specifications, including resource invariants, in any process.

Protected variables are those which appear in one resource's protection list, and can be modified, accessed, or mentioned in specifications in any process, but only within critical regions for the associated resource.

Process-protected variables appear in at least one resource invariant and in only one process. In that process they are modified only within critical regions for all the resources in whose invariants they appear. Also in that process, they can be read and appear in specifications outside of critical regions. Variables which appear free in some resource invariant but are not protected are either process-protected or global-constant, depending on whether they are ever written to.

2 Verification Condition Generation

2.1 Verification Conditions

A verification condition is a triple $[P]\, SI\, [Q]$ where SI is a "symbolic instruction":

$$SI ::= \epsilon \mid S \mid [P]\, \mathsf{jsr}_{\vec{x}}\, [Q] \mid \mathsf{if}\ B\ \mathsf{then}\ SI\ \mathsf{else}\ SI \mid SI\,;SI$$

A symbolic instruction is a piece of loop-free sequential code where all procedure calls have been instantiated to jsr instructions of the form $[P]\, \mathsf{jsr}_{\vec{x}}\, [Q]$. This form plays a central role in Smallfoot. We use it not only to handle procedure calls, but also for concurrency and for entry to and exit from a critical region.

Semantically, $[P]\, \mathsf{jsr}_{\vec{x}}\, [Q]$ is a "generic command" in the sense of [10]. It is the greatest relation satisfying the pre- and post-condition, and subject to the constraint that only the variables in $\vec{x}$ are modified.

The symbolic execution rule for the jsr instruction is:

$$\frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma' * \Sigma_F \qquad [(\Pi[\vec{x}'/\vec{x}]) \wedge Q * (\Sigma_F[\vec{x}'/\vec{x}])]\, C\, [R]}{[\Pi \wedge \Sigma]\, ([\Pi' \wedge \Sigma']\, \mathsf{jsr}_{\vec{x}}\, [Q]\,;C)\, [R]}\ \vec{x}' \text{ fresh}$$

To apply this rule we have to discover a frame axiom Σ_F which describes the portion of heap unchanged by a procedure call, and [2] describes a proof-theoretic method for obtaining them.

2.2 VCGen

For each procedure declaration $f(\vec{p}\,;\vec{v})[P]\, C\, [Q]$ we generate a set of verification conditions $vcg(f, [P]\, C\, [Q])$, which is itself defined using a helper function *chop*

that takes a command and produces a symbolic instruction together with a set of verification conditions. *vcg* just runs *chop* on the body C, tacks the pre and post onto the resulting symbolic instruction, and adds that to the verification conditions returned by *chop*.

$$
\begin{aligned}
vcg(g, [P]\,C\,[Q]) &= \{[P]\,SI\,[Q]\} \cup L \\
&\quad \text{where } SI\,, L = chop(g, C) \\
chop(g, S) &= S\,, \emptyset \\
chop(g, C\,;C') &= SI\,;SI'\,, L \cup L' \\
&\quad \text{where } SI\,, L = chop(g, C) \text{ and } SI'\,, L' = chop(g, C') \\
chop(g, \texttt{if}(B)\ \{C\}\ \texttt{else}\ \{C'\}) &= \mathsf{if}\ B\ \mathsf{then}\ SI\ \mathsf{else}\ SI'\,, L \cup L' \\
&\quad \text{where } SI\,, L = chop(g, C) \text{ and } SI'\,, L' = chop(g, C') \\
chop(g, \texttt{while}(B)\ [I]\ \{C\}) &= [I]\,\mathsf{jsr}_{mod(C)}\ [\neg B \wedge I]\,, vcg([B \wedge I]\,C\,[I]) \\
chop(g, f(\vec{x}\,;\vec{E})) &= ([\mathsf{emp}]\,\mathsf{jsr}_{\emptyset}\ [\vec{v}'{=}\vec{E} \wedge \mathsf{emp}]\,;[P[\vec{x}/\vec{p}, \vec{v}'/\vec{v}]]\,\mathsf{jsr}_{mod(C_f)[\vec{x}/\vec{p}, \vec{v}'/\vec{v}]}\ [Q[\vec{x}/\vec{p}, \vec{v}'/\vec{v}]])\,, \emptyset \\
&\quad \text{where } f(\vec{p}\,;\vec{v})[P]\,C\,[Q] \text{ and } \vec{v}' \text{ fresh} \\
chop(g, f(\vec{x}\,;\vec{E}) \parallel f'(\vec{x}'\,;\vec{E}')) &= ([\mathsf{emp}]\,\mathsf{jsr}_{\emptyset}\ [\vec{v}{=}\vec{E} \wedge \vec{v}'{=}\vec{E}' \wedge \mathsf{emp}]\,;[P * P']\,\mathsf{jsr}_{\vec{z} \cup \vec{z}'}\ [Q * Q'])\,, \emptyset \\
&\quad \text{where } [\mathsf{emp}]\,\mathsf{jsr}_{\emptyset}\ [\vec{v}{=}\vec{E} \wedge \mathsf{emp}]\,;[P]\,\mathsf{jsr}_{\vec{z}}\ [Q]\,, \emptyset = chop(g, f(\vec{x}\,;\vec{E})) \\
&\qquad [\mathsf{emp}]\,\mathsf{jsr}_{\emptyset}\ [\vec{v}'{=}\vec{E}' \wedge \mathsf{emp}]\,;[P']\,\mathsf{jsr}_{\vec{z}'}\ [Q']\,, \emptyset = chop(g, f'(\vec{x}'\,;\vec{E}')) \\
chop(g, \texttt{with}\ r\ \texttt{when}(B)\ \{C\}) &= \big([\mathsf{true} \wedge \mathsf{emp}]\,\mathsf{jsr}_{\emptyset}\ [B \wedge R]\,;SI\,;[\mathsf{true} \wedge R]\,\mathsf{jsr}_{\vec{x} \cup \vec{u}}\ [\mathsf{true} \wedge \mathsf{emp}]\big)\,, L \\
&\quad \text{where } SI\,, L = chop(g, C) \text{ and } r(\vec{x})R \text{ and } \vec{u} = fv(R) \cap \textstyle\bigcup_{f \in par(g)} mod(f)
\end{aligned}
$$

The definition of *chop* for primitive statements, sequential composition, conditionals and loops is mostly as expected, except that for loops we generate a jsr instruction that allows invariants to be smaller than they might otherwise be, because of framing.

For procedure call, we rename the value parameters and use two jsr's: the first to initialize the renamed parameters and the second to abstract the body of the procedure, using only its spec. This renaming allows the postcondition to refer to the initial value of the parameters which are not modified by the body. The composition of the two jsr's satisfies a spec $[A] - [B]$ iff the second one satisfies $[A \wedge \vec{v}'{=}\vec{E}] - [B]$. In the definition, $mod(C_f)$ is the set of variables modified by C_f (or one of the procedures that C_f calls) except for protected variables modified within a CCR.

For parallel composition we emit two jsr's that combine the initializations of the two procedure calls, and take the $*$-combinations of the respective preconditions and postconditions, following the parallel proof rule

$$\frac{[P]\,C\,[Q] \qquad [P']\,C'\,[Q']}{[P * P']\,C \parallel C'\,[Q * Q']}\ .$$

Entry to, and exit from, CCRs is modeled by jsr instructions. The entry jsr adds the resource invariant and boolean condition to the symbolic state. Since

this represents adding *any* concrete heap satisfying the invariant and condition, upon entry to a CCR the body cannot assume or depend on anything further about the acquired heap. This is how we handle potential interference from parallel processes, which may change one concrete heap satisfying the invariant to another. Additionally, outside interference is prevented in code following a CCR since the exit jsr removes the resource invariant from the symbolic state and forgets the values of variables which are protected, $\vec{x}$, or might be modified by processes $par(f)$ running in parallel, $\vec{u}$. The net result is that correctness of a parallel program is reduced to several sequential triples, and no interleaving needs to be considered. This VC definition follows the description of the CCR proof rule

$$\frac{[(P * R_r) \wedge B]\, C\, [Q * R_r]}{[P]\, \texttt{with}\ r\ \texttt{when}(B)\ \{C\}\, [Q]}$$

(where R_r is an invariant formula associated with resource r) and both occurrences of jsr make use of the frame axiom inference capability; the precondition P of a CCR is maintained after the entry jsr, and an appropriate Q part for the postcondition in the rule is discovered as a frame axiom for the exit jsr. The $r(\vec{x})R$ in the where clause indicates that R is the declared invariant of r in the program.

To tie all of this together there is one further check that must be made. The *init* procedure must establish all of the resource invariants, separately, and the precondition of *main* if a main procedure is included. So given $\texttt{init}()[P]\, C\, [Q]$ and $\texttt{main}()[P']\, C'\, [Q']$ we check the entailment $Q \vdash R_1 * \cdots * R_n * P'$. We also require C to not contain procedure calls, CCRs, or parallel compositions. All told, the property that this establishes (following the rule for complete programs [7, 3]) for a program is

$$[P]\, C\, ;\mathrm{RESDECLS}\, ;\texttt{let}\ \mathrm{PROCDECLS}\ \texttt{in}\ C'\, [Q' * R_1 * \cdots * R_n]$$

where PROCDECLS consists of those procedure declarations other than *main* and *init*. (We could also include a finalization procedure that disposes of the R_i at the end.)

3 Resource Initialization

Resource initializers are subject to the following constraints:

1. No resource's initializer modifies a variable mentioned by a distinct resource: $mod(C_i) \cap var(r_j) = \emptyset$ when $i \neq j$.
   This is performed by checking, for all i

$$\begin{aligned} & mod(C_i) \cap var(C_1, \ldots, C_{i-1}) = \emptyset \\ \text{and} \quad & var(C_i) \cap mod(C_1, \ldots, C_{i-1}) = \emptyset \end{aligned}$$

2. The `init` procedure, if it appears, and the resource initializers C_i contain no procedure calls or CCRs.

These constraints ensure that the order in which the initializers are executed is immaterial. Therefore, we have an additional verification condition:

$$vcg([P]\, C\, [Q * R_1 * \cdots * R_n])$$

where $R_1, \ldots, R_n$ are all the resource invariants and

$$P, C, Q = \begin{cases} P', C', Q' & \text{if } \texttt{init}()[P']\, C'\, [Q'] \in \Delta \\ \mathsf{emp}, C_1\,; \cdots; C_n, \mathsf{emp} & \text{otherwise} \end{cases}$$

where $C_1, \ldots, C_n$ are all the resource initializers (in some unspecified order).

Also, in case `init` appears, the precondition of procedure `main`, if it appears, is taken to be the postcondition of `init`, irrespective of what appears in the file.

References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
3. S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 2007. Preliminary version in *CONCUR'04*.
4. S. Brookes. A revisionist history of concurrent separation logic. *Electr. Notes Theor. Comput. Sci.*, 2011.
5. S. A. Cook. Soundness and completeness of an axiomatic system for program verification. *SIAM J. on Computing*, 1978.
6. C. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on the Semantics of Algorithmic Languages*, 1971.
7. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 2007. Preliminary version in *CONCUR'04*.
8. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 1976.
9. U. S. Reddy and J. C. Reynolds. Syntactic control of interference for separation logic. In *POPL*, 2012.
10. J. Schwarz. Generic commands—A tool for partial correctness formalisms. *The Computer Journal*, 1977.