A Structural Approach to Prophecy Variables

Zipeng Zhang[1], Xinyu Feng[1], Ming Fu[1], Zhong Shao[2], and Yong Li[1]

[1] University of Science and Technology of China
[2] Yale University

Abstract. Verifying the implementation of concurrent objects essentially proves the fine-grained implementation of object methods refines the corresponding abstract atomic operations. To simplify the specifications and proofs, we usually need auxiliary history and prophecy variables to record historical events and to predict future events, respectively. Although the meaning of history variables is obvious, the semantics of prophecy variables and the corresponding auxiliary code is tricky and has never been clearly spelled out operationally.

In this paper, we propose a new language construct, future blocks, that allows structural use of prophecy variables to refer to events in the future. The semantics of the construct is simple and easy to understand, without using any form of oracle or backward reasoning. Our language also separates auxiliary states from physical program states. With careful syntactic constraints, it ensures the use of history and prophecy variables would not affect the behaviors of the original program, which justifies the verification method based on the use of auxiliary variables.

1 Introduction

One of the major challenges to verify shared-state concurrent programs is the very fine-grained interleaving between threads. Usually the correctness argument goes in the following way: thread t knows it could do A because it (or someone else) has done B before. However, Hoare-style reasoning uses assertions specifying only the current state, so we cannot refer to the historical events directly in our assertions. To address this issue, a well-known technique is to introduce history variables and code that assigns proper values to them when certain events occur. In our assertions, instead of saying "event B has occurred before", we only need to say something like "the variable v_B has value 1". Usually history variables are write-only, so that their use would not affect the behavior of the original program.

On the other hand, we may also need to refer to events that may occur in a future point of the program execution, especially when we verify optimistic concurrent algorithms. Optimistic algorithms usually access shared states without first acquiring exclusive ownership of them. They will validate the access in a later time. If the validation validation succeeds, the access finishes the task. Otherwise the algorithm rolls back and retry the same process. As the dual of history variables, we may need prophecy variables to predict what would happen in the future [1].

```
1   push(x){
2     ...
3     < C;
4       absSt:= (x::absSt);
5     >
6     ...
    }
```

(a) the use of abstract state

```
1   push(x){
1'    guess b;
2     ...
3     < C;
4       if (succeeds at line 8)
4'      if (b)
5         absSt:= (x::absSt);
6     >
7     ...
8     validation here
8'    assert(b <=> line 8 succeeds);
9     ...
    }
```

(b) the need of prophecy variables

Fig. 1. The need of prophecy variables for linearizability verification. Note (b) is made up by the authors to demonstrate the idea only. It does not come from Vafeiadis [9].

Using Prophecy Variables to Verify Linearizability. Vafeiadis [9] proposed to verify the linearizability of concurrent objects by inserting the corresponding abstract operations at the program's linearization point, the point where the effect of the operation becomes visible to other threads. As shown in Fig. 1(a), the linearization point of the push method is at line 3, where the command C finishes the push operation over the concrete data structure. For proof purpose, we insert line 4, which pushes x onto an abstract stack `absSt`. Such an abstract atomic operation can be viewed as a specification of the push method. Here $\langle C \rangle$ means C will be executed atomically. By enforcing as program invariant some consistency relation between the concrete data and the abstract version, we essentially proved that the concrete code is a refinement of the abstract operation, therefore the method is linearizable.

However, for some smart and complex algorithms, it is not easy to determine the linearization point. In Fig. 1(b) (let's first ignore lines 1', 4' and 8'), whether line 3 is a linearization point or not may depends on whether the validation at line 8 succeeds or not, therefore whether the abstract operation at line 5 should be executed is conditional. Vafeiadis [9] used a prophecy variable to refer to the result of validation at line 8. Line 1' declares the variable b, and uses an oracle to initialize it such that the `assert` command at line 8' holds. Then we *replace* line 4 with 4', which tests the value of b to decide whether line 5 should be executed.

The use of prophecy variables here is very intuitive and natural, but it is difficult to give operational semantics of the `guess` command, which needs an oracle to do the initialization so that the `assert` command will hold in the future. There is no formal semantics given by Vafeiadis.

Other Related Works. Since the idea of prophecy variables was proposed by Abadi and Lamport [1], it has been used for concurrency verification in various settings. However, most of the works are for refinement [7, 6] or model checking [2], and the semantics is given based on execution traces, which is not suitable for Hoare-style verification. Sezgin et al. [8] introduced tressa assertions to express properties about the future of an execution. The semantics is modeled in terms of backward reachability from an end state of the program. The work is proposed for reduction based proof of atomicity.

Our Work. In this paper we propose a new language construct for structural use of prophecy variables with clearly defined semantics. With careful syntactic constraints in the language, we also ensure that adding prophecy (and history) variables do not affect the behaviors of the original program. This justifies the soundness of this proof method. Our work makes the following new contributions:

– We introduce a novel language construct **future**(B) **do** C' **on** C. It means whether we execute C' before C or not depends on the value of B at the end of C. As we will show in Section 2, the operational semantics could be very simply defined without using an oracle or backward reasoning.
– We give Hoare-style logical rules for the future block, which allows us to apply the same idea of Vafeiadis to verify linearizability of concurrent objects. We show how the RDCSS example in [9] can also be verified in our logic.
– Our language separates auxiliary program state from physical program state. It also uses stratified syntax to distinguish the original program from the auxiliary code inserted for verification only. Then we formally prove that adding auxiliary code would not change the behavior of the original program. This gives a formal justification of the verification method using history and prophecy variables.

2 The Language

We present the syntax of our language in Fig. 2, which is stratified into several levels. The first level (E, B and C) is the syntax of the original programs to be verified. It is a simple WHILE language with parallel composition and memory operations. The command $\langle\widehat{C}\rangle$ means $\widehat{C}$ is executed atomically. Here $\widehat{C}$ is a sequential subset of C, which does not contain parallel composition. Expressions E and B are pure in that they do not update variables and do not access memory.

The lifted syntax ($\widetilde{E}$, $\widetilde{B}$ and D) is for writing auxiliary code, which can be inserted into the original program to get a program K. We use α to represent auxiliary history variables and prophecy variables in D. The lifted expressions $\widetilde{E}$ and $\widetilde{B}$ could use both program variables (x) and auxiliary ones. It is important to note that D does not update program variables, not update physical memory and not contain **while** loops (so it must terminate).

The statement K is a mix of C and D with two new language constructs. **future** $\widetilde{B}$ **do** $\langle C;\ D_1 : D_2\rangle$ **on** $\widehat{K}$ would execute either $\langle C; D_1\rangle; \widehat{K}$ or $\langle C; D_2\rangle; \widehat{K}$, depending on whether $\widetilde{B}$ is true or false at the end of $\widehat{K}$. Here $\widehat{K}$ is a special

$$
\begin{array}{rrl}
(\mathit{Expr}) & E & ::= x \mid n \mid E+E \mid E-E \mid \ldots \\
(\mathit{Bexp}) & B & ::= \mathrm{true} \mid \mathrm{false} \mid E=E \mid E\neq E \mid \neg B \mid \ldots \\
(\mathit{Cmd}) & C & ::= x := E \mid x := [E] \mid [E] := E \mid \textbf{skip} \mid \langle \widehat{C} \rangle \mid C;\ C \\
 & & \quad \mid \textbf{if } B \textbf{ then } C \textbf{ else } C \mid \textbf{while } B \textbf{ do } C \mid C \parallel C \\
(\mathit{Lexpr}) & \widetilde{E} & ::= \alpha \mid E \mid \widetilde{E}+\widetilde{E} \mid \widetilde{E}-\widetilde{E} \mid \ldots \\
(\mathit{Lbexp}) & \widetilde{B} & ::= B \mid \widetilde{E}=\widetilde{E} \mid \widetilde{E}\neq\widetilde{E} \mid \neg\widetilde{B} \mid \ldots \\
(\mathit{LCmd}) & D & ::= \alpha := \widetilde{E} \mid \alpha := \lfloor\widetilde{E}\rfloor \mid \lfloor\widetilde{E}\rfloor := \widetilde{E} \mid \textbf{skip} \mid \langle D \rangle \mid D;\ D \\
 & & \quad \mid \textbf{if } \widetilde{B} \textbf{ then } D_1 \textbf{ else } D_2 \\
(\mathit{Stmts}) & K & ::= C \mid D \mid K;\ K \mid \textbf{future } \widetilde{B} \textbf{ do } \langle C;\ D_1 \!:\! D_2 \rangle \textbf{ on } \widehat{K} \mid \langle \widehat{K} \rangle \\
 & & \quad \mid K_1 \parallel K_2 \mid \textbf{if } B \textbf{ then } K \textbf{ else } K \mid \textbf{while } B \textbf{ do } K \mid \textbf{assume}(\widetilde{B})
\end{array}
$$

Fig. 2. Syntax of the language

$$
\begin{array}{llll@{\qquad}llll}
(\mathit{Store}) & s & \in & \mathit{PVar} \to \mathit{Int} & (\mathit{Heap}) & h & \in & \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{LStore}) & \mathit{ls} & \in & \mathit{LVar} \to \mathit{Int} & (\mathit{LHeap}) & \mathit{lh} & \in & \mathit{Nat} \rightharpoonup \mathit{Int} \\
(\mathit{State}) & \sigma & ::= & (s, h, \mathit{ls}, \mathit{lh}) & (\mathit{Trans}) & \mathcal{R}, \mathcal{G} & \in & \mathcal{P}(\mathit{State} \times \mathit{State})
\end{array}
$$

Fig. 3. Program states

K with no parallel composition and **future** blocks. We also require implicitly D_1 and D_2 do not update the auxiliary variables in $\widetilde{B}$. The other new construct **assume**($\widetilde{B}$) blocks the execution if $\widetilde{B}$ is false, and does nothing otherwise. We do not allow users to write the **assume** statement directly. As we will show below, it is a run-time command which is automatically inserted during execution. All the implicit syntax constraints could be enforced with a simple syntactic sanity check. Note that the boolean expression in **if** and **while** statements at this level can only use program variables. This is to ensure that the value of auxiliary variables would not affect the execution of the original program.

We model program states in Fig. 3. A program state σ consists of a physical store s, a heap h, an auxiliary store ls and an auxiliary heap lh. The store s (auxiliary store ls) maps program variables (auxiliary variables) to integers. The heap h (auxiliary heap lh) is a finite partial mapping from natural numbers to integers. State transitions $\mathcal{R}$ and $\mathcal{G}$ are binary relations of states.

We show operational semantics to the key language constructs in Fig. 4. The complete definition is given in the companion technical report [11]. Transitions between closed program configurations (the pair (K, σ)) are modeled by the binary relation $\leadsto$. We use $\leadsto^*$ as its reflexive and transitive closure. Transitions with environment's interference $\mathcal{R}$ are modeled as $\_ \overset{\mathcal{R}}{\longmapsto} \_$. Semantics for the **future** block is straightforward. We nondeterministically choose to execute either

$$\frac{[\![E]\!]_s = n}{(x := E, (s, h, ls, lh)) \rightsquigarrow (\mathbf{skip}, (s\{x \rightsquigarrow n\}, h, ls, lh))}$$

$$\frac{[\![\widetilde{E}]\!]_{(s,ls)} = n}{(\alpha := \widetilde{E}, (s, h, ls, lh)) \rightsquigarrow (\mathbf{skip}, (s, h, ls\{\alpha \rightsquigarrow n\}), lh)}$$

$$\frac{(\widehat{K}, \sigma) \rightsquigarrow^* (\mathbf{skip}, \sigma')}{(\langle \widehat{K} \rangle, \sigma) \rightsquigarrow (\mathbf{skip}, \sigma')} \qquad \frac{(\widehat{K}, \sigma) \rightsquigarrow^* \mathbf{abort}}{(\langle \widehat{K} \rangle, \sigma) \rightsquigarrow \mathbf{abort}} \qquad \frac{\sigma = (s, h, ls, lh) \quad [\![\widetilde{B}]\!]_{(s,ls)} = \text{true}}{(\mathbf{assume}(\widetilde{B}), \sigma) \rightsquigarrow (\mathbf{skip}, \sigma)}$$

$$\frac{}{(\mathbf{future}\ \widetilde{B}\ \mathbf{do}\ \langle C;\ D_1 : D_2 \rangle\ \mathbf{on}\ \widehat{K}, \sigma) \rightsquigarrow (\langle C; D_1 \rangle; \widehat{K}; \mathbf{assume}(\widetilde{B}), \sigma)}$$

$$\frac{}{(\mathbf{future}\ \widetilde{B}\ \mathbf{do}\ \langle C;\ D_1 : D_2 \rangle\ \mathbf{on}\ \widehat{K}, \sigma) \rightsquigarrow (\langle C; D_2 \rangle; \widehat{K}; \mathbf{assume}(\neg\widetilde{B}), \sigma)}$$

$$\frac{(K_1, \sigma) \rightsquigarrow (K_1', \sigma')}{(K_1 \parallel K_2, \sigma) \rightsquigarrow (K_1' \parallel K_2, \sigma')} \qquad \frac{(K_2, \sigma) \rightsquigarrow (K_2', \sigma')}{(K_1 \parallel K_2, \sigma) \rightsquigarrow (K_1 \parallel K_2', \sigma')}$$

$$\frac{(K_i, \sigma) \rightsquigarrow \mathbf{abort} \quad i \in \{1, 2\}}{(K_1 \parallel K_2, \sigma) \rightsquigarrow \mathbf{abort}} \qquad \frac{}{(\mathbf{skip} \parallel \mathbf{skip}, \sigma) \rightsquigarrow (\mathbf{skip}, \sigma)}$$

$$\frac{(\sigma, \sigma') \in \mathcal{R}}{(K, \sigma) \xmapsto{\mathcal{R}} (K, \sigma')} \qquad \frac{(K, \sigma) \rightsquigarrow (K', \sigma')}{(K, \sigma) \xmapsto{\mathcal{R}} (K', \sigma')} \qquad \frac{(K, \sigma) \rightsquigarrow \mathbf{abort}}{(K, \sigma) \xmapsto{\mathcal{R}} \mathbf{abort}}$$

Fig. 4. Selected rules for operational semantics

$\langle C; D_1 \rangle; \widehat{K}$ or $\langle C; D_2 \rangle; \widehat{K}$. For each choice, we append at the end **assume**($\widetilde{B}$) and **assume**($\neg\widetilde{B}$) respectively. The semantics for **assume** is standard.

3 The Program Logic

In this section, we extend the rely-guarantee logic [5] to reason about programs with future blocks. Then we show that inserting auxiliary code D into the original program C would not affect the behavior of C, which justifies the validity of this verification method.

3.1 Reasoning about Future Blocks

Figure 5 shows the syntax and semantics of selected assertions. We use separation logic assertions to specify program states, whose semantics is standard. An action a specifies a state transition, i.e., a binary relation over states. Rely (R) and Guarantee (G) conditions are both actions.

Due to the limit of space, we only show the inference rules for the future block and the assume statement and leave other rules in the companion technical report[11]. The FUT rule for future blocks is straightforward, which simply

$$\begin{array}{rrcl}
(\mathit{StateAssert}) & p, q & ::= & \widetilde{B} \mid \mathit{emp} \mid E \mapsto E \mid p * q \mid \widetilde{E} \rightarrowtail \widetilde{E} \mid \cdots \\
(\mathit{Action}) & a, R, G & ::= & p \ltimes q \mid [p] \mid a \wedge a \mid a \vee a \mid \cdots
\end{array}$$

$$\begin{array}{ll}
(s, h, \mathit{ls}, \mathit{lh}) \models \widetilde{B} & \text{iff } [\![\widetilde{B}]\!]_{(s,\mathit{ls})} = \text{true} \\
(s, h, \mathit{ls}, \mathit{lh}) \models \mathit{emp} & \text{iff } h = \emptyset \text{ and } \mathit{lh} = \emptyset \\
(s, h, \mathit{ls}, \mathit{lh}) \models E_1 \mapsto E_2 & \text{iff there exist } \ell \text{ such that} [\![E_1]\!]_s = \ell, [\![E_2]\!]_s = n, \\
& \qquad \mathit{dom}(h) = \{\ell\} \text{ and } h(\ell) = n \\
(s, h, \mathit{ls}, \mathit{lh}) \models \widetilde{E}_1 \rightarrowtail \widetilde{E}_2 & \text{iff there exist } \ell \text{ such that} [\![\widetilde{E}_1]\!]_{(s,\mathit{ls})} = \ell, [\![\widetilde{E}_2]\!]_{(s,\mathit{ls})} = n, \\
& \qquad \mathit{dom}(\mathit{lh}) = \{\ell\} \text{ and } \mathit{lh}(\ell) = n \\
(s, h, \mathit{ls}, \mathit{lh}) \uplus (s', h', \mathit{ls}', \mathit{lh}') \stackrel{\text{def}}{=} & \begin{cases} (s, h \cup h', \mathit{ls}, \mathit{lh} \cup \mathit{lh}') & \\ \qquad \text{if } s = s', \mathit{dom}(h) \cap \mathit{dom}(h') = \emptyset, & \\ \qquad \mathit{ls} = \mathit{ls}', \mathit{dom}(\mathit{lh}) \cap \mathit{dom}(\mathit{lh}') = \emptyset & \\ \mathit{undef} \qquad \text{otherwise} & \end{cases} \\
\sigma \models p * q & \text{iff exist } \sigma_1 \text{ and } \sigma_2, \sigma_1 \uplus \sigma_2 = \sigma, \sigma_1 \models p \text{ and } \sigma_2 \models q \\
(\sigma, \sigma') \models p \ltimes q & \text{iff } \sigma \models p \text{ and } \sigma' \models q \\
(\sigma, \sigma') \models [p] & \text{iff } \sigma = \sigma' \text{ and } \sigma \models p
\end{array}$$

Fig. 5. Assertions and their semantics

$$\frac{\begin{array}{c} R; G \vdash \{p\}\ \langle C; D_1\rangle; \widehat{K}; \textbf{assume}(\widetilde{B})\ \{q\} \\ R; G \vdash \{p\}\ \langle C; D_2\rangle; \widehat{K}; \textbf{assume}(\neg\widetilde{B})\ \{q\} \end{array}}{R; G \vdash \{p\}\ \textbf{future}\ \widetilde{B}\ \textbf{do}\ \langle C;\ D_1 : D_2\rangle\ \textbf{on}\ \widehat{K}\ \{q\}}\ (\textsc{fut})$$

$$\frac{p \wedge \widetilde{B} \Rightarrow q \quad \mathrm{sta}(q, R)}{R; G \vdash \{p\}\ \textbf{assume}(\widetilde{B})\ \{q\}}\ (\textsc{asm})$$

$$\text{where } \mathrm{sta}(p, a) \stackrel{\text{def}}{=} \forall \sigma, \sigma'.\, (\sigma \models p) \wedge ((\sigma, \sigma') \models a) \Rightarrow \sigma' \models p$$

Fig. 6. Selected inference rules

requires that both execution paths of the statement satisfy the specification. In the ASM rule for the assume statement, we know $\widetilde{B}$ holds if the execution falls through. However, we might need to weaken $p \wedge \widetilde{B}$ to get a stable post-condition q. Here stability of an assertion with respect to the rely condition means that the validity of the assertion would be preserved by state transitions in the rely condition.

Semantics and Soundness. We give semantics of the judgment $R; G \vdash \{p\}\ K\ \{q\}$ below, which ensures the following non-interference property $(K, \sigma, \mathcal{R}) \Longrightarrow (\mathcal{G}, q)$. It says that, with an environment whose state transitions in $\mathcal{R}$, (K, σ) would not abort, every step of its execution satisfies the guarantee $\mathcal{G}$, and the post-condition q holds at the termination state if the execution terminates.

$$
\begin{array}{ll}
\mathit{Er}(C) = C & \mathit{Er}(D) = \textbf{skip} \\
\mathit{Er}(\langle K \rangle) = \langle \mathit{Er}(K) \rangle & \mathit{Er}(K_1;\ K_2) = \mathit{Er}(K_1);\ \mathit{Er}(K_2)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Er}(\textbf{if } B \textbf{ then } K_1 \textbf{ else } K_2) & = & \textbf{if } B \textbf{ then } \mathit{Er}(K_1) \textbf{ else } \mathit{Er}(K_2) \\
\mathit{Er}(\textbf{while } B \textbf{ do } K) & = & \textbf{while } B \textbf{ do } \mathit{Er}(K) \\
\mathit{Er}(\textbf{future } \widetilde{B} \textbf{ do } \langle C;\ D_1 \!:\! D_2 \rangle \textbf{ on } \widehat{K}) & = & \langle C \rangle;\ \mathit{Er}(\widehat{K}) \\
\mathit{Er}(K_1 \parallel K_2) & = & \mathit{Er}(K_1) \parallel \mathit{Er}(K_2)
\end{array}
$$

Fig. 7. Erasure of auxiliary code

Definition 3.1 (Non-interference). *$(K, \sigma, \mathcal{R}) \Longrightarrow^0 (\mathcal{G}, q)$ always holds; $(K, \sigma, \mathcal{R}) \Longrightarrow^{n+1} (\mathcal{G}, q)$ holds iff $(K, \sigma) \not\rightsquigarrow$ **abort**,and,*

1. *for all σ',if $(\sigma, \sigma') \in \mathcal{R}$, then for all $k \leq n$, $(K, \sigma', \mathcal{R}) \Longrightarrow^k (\mathcal{G}, q)$;*
2. *for all σ', if $(K, \sigma) \rightsquigarrow (K', \sigma')$, then $(\sigma, \sigma') \in \mathcal{G}$ and $(K', \sigma', \mathcal{R}) \Longrightarrow^k (\mathcal{G}, q)$ holds for all $k \leq n$;*
3. *if $K =$ **skip**, then $\sigma \models q$, and $(K, \sigma, \mathcal{R}) \Longrightarrow^k (\mathcal{G}, q)$ holds for all $k \leq n$.*

We say $(K, \sigma, \mathcal{R}) \Longrightarrow (\mathcal{G}, q)$ if $\forall k.\, (K, \sigma, \mathcal{R}) \Longrightarrow^k (\mathcal{G}, q)$

We define $[\![a]\!]$ as $\{(\sigma, \sigma') \mid (\sigma, \sigma') \models a\}$. Then we can define $R; G \models \{p\}\ K\ \{q\}$ as the following, and give the soundness theorem.

Definition 3.2. *$R; G \models \{p\}\ K\ \{q\}$ iff, for all σ, if $\sigma \models p$, then $(K, \sigma, [\![R]\!]) \Longrightarrow ([\![G]\!], q)$.*

Theorem 3.3 (Soundness). *If $R; G \vdash \{P\}\ K\ \{Q\}$,then $R; G \models \{P\}\ K\ \{Q\}$.*

We also want to point out that it should be easy to develop rules similar to the FUT rule in other program logic [10, 3]. Here we pick rely-guarantee reasoning mainly for simplicity.

3.2 Auxiliary Code Erasure

Theorem 3.3 just shows the logic ensures the partial correctness of the annotated code K, but what we want ultimately is that the original program C is well-behaved. We show in this section that this is indeed guaranteed by our verification method. In Fig. 7 we show the erasing process *Er* that removes the auxiliary code in K to get the original program C. It simply removes D in K, or replace occurrence of D with **skip**.

The following theorem says for all safe K, it has no less behaviors than the original program $\mathit{Er}(K)$. Therefore we can verify the annotated program instead of the original one.

Theorem 3.4. *If $R; G \models \{p\}\ K\ \{q\}$, $C = \mathit{Er}(K)$ and $(s, h, \mathit{ls}, \mathit{lh}) \models p$, the following are true:*

1. *for all s' and h', if $(C,(s,h)) \leadsto^* (\textbf{\textit{skip}},(s',h'))$, then $\exists ls'\, lh'.\,(K,(s,h,ls,lh) \leadsto^* (\textbf{\textit{skip}},(s',h',ls',lh')))$;*
2. $(C,(s,h)) \not\leadsto^*$ ***abort***.

Here the transition $(C,(s,h)) \leadsto (C',(s',h'))$ can be derived easily from the transition $(C,\sigma) \leadsto (C',\sigma')$, since C does not access logical states. The complete definition is given in the technical report [11].

We define semantics of *auxiliary-state-independent assertions* $\widetilde{p}$ below.

$$(s,h,ls,lh) \models \widetilde{p} \ \text{ iff } \ \text{there exists } ls' \text{ and } lh' \text{ such that } (s,h,ls',lh') \models p \text{ holds}\,.$$

Then the following corollary trivially follows Theorem 3.4. That is, given $R;G \models \{p\}\ K\ \{q\}$, we can get the partial correctness of $Er(K)$ with empty environment.

Corollary 3.5. *If $R;G \models \{p\}\ K\ \{q\}$ and $C = Er(K)$, then $R_0;G_0 \models \{\widetilde{p}\}\ C\ \{\widetilde{q}\}$ where $R_0 = [true]$ and $G_0 = true \ltimes true$.*

4 Example

We use an example to show how our logic supports verification with prophecy variables. More examples can be found in our technical report [11].

4.1 The RDCSS Algorithm

Restricted double-compare single-swap(RDCSS) is defined by Harris[4] in the implementation of multiple compare-and-swap(MCAS). The code is shown in Fig. 8. RDCSS takes five arguments a_1, a_2, o_1, o_2 and n_2, which are stored in the descriptor d. Here a_1 and a_2 are memory addresses and o_1 and o_2 are

```
class Descriptor {
    address_t $a_1$, $a_2$;
    word_t $o_1$, $o_2$, $n_2$;
    single word_t AbsResult;
    single word_t $r_2$;  }

Complete(Descriptor d) {
        local r;
C1:     ⟨ r:= [d.$a_1$]; ⟩
        if (r=d.$o_1$)
C2:         CAS1(d.$a_2$, d, d.$n_2$);
        else
C3:         CAS1(d.$a_2$, d, d.$o_2$);
}
```

```
RDCSS(Descriptor d) {
    local r;
    r:=CAS1(d.$a_2$, d.$o_2$, d);
    while (IsDesc(r)) {
        Complete(r);
        r:=CAS1(d.$a_2$, d.$o_2$, d);
    }
    if (r=d.$o_2$) Complete(d);
    return r;
}
```

Fig. 8. RDCSS implementation

their expected values. If both addresses contain their expected values, then the new value n_2 is stored a_2. The function returns the current value stored at a_2. Addresses are split into two disjoint domains, A and B. a_1 and a_2 belong to A and B respectively. A-type addresses could be accessed using standard memory operations, while B-type can be accessed only through the RDCSS function.

The implementation of RDCSS uses a variant of CAS. As shown below, it returns the old values stored in the address instead of a boolean.

```
value_t CAS1(value_t *addr, value_t exp, value_t new) {
     value_t v;
    ⟨ v := [addr];  if (v=exp) [addr]:=new; ⟩
    return v;
}
```

RDCSS first attempts to place its descriptor at the memory location a_2, which means to 'lock' the location. If it succeeds, then continues to attempt to update the memory location a_2 with the new value. If a thread A reads a_2 and finds it contains a descriptor, it means that there is another thread B trying to update a_2 through RDCSS. In this case thread A should call the helper function *Complete(d)* to help thread B complete the operation. The function *IsDesc* tests whether its parameter points to a descriptor.

4.2 Proofs

Vafeiadis [9] verified the algorithm using auxiliary prophecy variables. The reason we need prophecy variables is that whether the linearization point is at line C1 or not (see Fig. 8) depends on the comparison result at lines C2 and C3. Here we follow the same idea in his proof, except that we rewrite the method *complete(d)* using our **future** block instead of using *guess* and *assert* statements. The instrumented code of the *complete(d)* function and the proof sketch is shown in Fig. 9. The proof for the RDCSS function is the same with the one by Vafeiadis, thus omitted here.

Following Vafeiadis [9], two auxiliary variables are added in the *Descriptor* in Fig. 8. *AbsResult* represents the value of a_2 when RDCSS is called, and r_2 represents its value when RDCSS returns. Some important assertions taken from Vafeiadis [9] are shown on top of Fig. 9. $D_d(a_1, a_2, o_1, o_2, n_2, a, r_2)$ describes a valid descriptor pointed by d. $U_d(a_1, a_2, o_1, o_2, n_2)$ describes a descriptor whose *AbsResult* and r_2 are undefined. The abstraction assertion $K(x)$ maps the concrete value of x to the abstract value. The overall invariant *RDCSS_Inv* asserts that all locations in A exist, all locations in B have matching concrete values and abstract values, and there are some used garbage RDCSS Descriptors.

In Fig. 9, the texts in yellow background are the inserted auxiliary code. We add the auxiliary variable $lda2$ to capture the value of $d.a_2$ in the future at lines C2 and C3. Its value is assigned by D_3. The boolean expression $\widetilde{B}$ $(lda2 = d)$ tests whether the comparison at C2 and C3 would succeed. If it holds, we know the line $C1$ is the linearization point, and update the abstract results correspondingly in D_1. Here we just want to demonstrate the use of the **future** block in the proof. More details can be found in Vafeiadis [9] or our TR [11].

$$\begin{aligned} D_d(a_1, a_2, o_1, o_2, n_2, a, r_2) \overset{\text{def}}{=} \; & d \mapsto \{.a_1 = a_1; .o_1 = o_1; .a_2 = a_2; .n_2 = n_2; .AbsResult = a; .r_2 = r_2\} \\ & \wedge (a_1 \in A) \wedge (a_2 \in B) \wedge \text{IsDesc(d)} \\ & \wedge \neg\text{IsDesc}(o_2) \wedge \neg\text{IsDesc}(n_2) \end{aligned}$$

$$U_d(a_1, a_2, o_1, o_2, n_2) \overset{\text{def}}{=} D_d(a_1, a_2, o_1, o_2, n_2, undef, undef)$$

$$K(x) \overset{\text{def}}{=} (x \in B) \wedge \exists d,\ v,\ w. \left\{ \begin{array}{l} (Abs[x] \mapsto v * x \mapsto v \wedge \neg\text{IsDesc(v)}) \\ \vee(Abs[x] \mapsto v * x \mapsto d * U_d(\_, x, \_, v, \_)) \\ \vee(Abs[x] \mapsto w * x \mapsto d * D_d(\_, x, \_v, \_, v, w)) \end{array} \right\}$$

$$RDCSS\_Inv \overset{\text{def}}{=} \exists T.\ \circledast_{x \in A}\ .x \mapsto \_ * \circledast_{x \in B}.K(x) * \circledast_{d \in T}.\exists o_2, D_d(\_, \_, \_, o_2, \_, o_2, \_)$$

```
{$RDCSS\_Inv \wedge D_d(a_1, a_2, o_1, o_2, n_2, \_, \_) * true$}
future ($\widetilde{B}$)
do
⟨
C1:    v:=[d.$a_1$];
{ $(\exists n.RDCSS\_Inv \wedge D_d(a_1, a_2, o_1, o_2, n_2, \_, \_) \wedge d.a_1 \mapsto n \wedge v = n) * true$ }
        {$D_1$:
$\left\{ \begin{array}{l} (RDCSS\_Inv \wedge v \neq d.o_1 \wedge D_d(a_1, a_2, o_1, o_2, n_2, o_2, o_2) * true) \\ \vee(RDCSS\_Inv \wedge v = d.o_1 \wedge D_d(a_1, a_2, o_1, o_2, n_2, o_2, n_2) * true) \end{array} \right\}$
        $D_2$}
⟩
on {
        if (v = d.$o_1$)
        ⟨   $D_3$;
C2:           CAS1(d.$a_2$, d, d.$n_2$);         ⟩
        else
        ⟨   $D_3$;
C3:           CAS1(d.$a_2$, d, d.$o_2$);         ⟩
}
$\left\{ \begin{array}{l} (lda2 \neq d \wedge RDCSS\_Inv \wedge D_d(a_1, a_2, o_1, o_2, n_2, \_, \_) * true) \\ \vee(lda2 = d \wedge RDCSS\_Inv \wedge D_d(a_1, a_2, o_1, o_2, n_2, o_2, n_2) * true) \\ \vee(lda2 = d \wedge RDCSS\_Inv \wedge D_d(a_1, a_2, o_1, o_2, n_2, o_2, o_2) * true) \end{array} \right\}$
{ $RDCSS\_Inv \wedge D_d(a_1, a_2, o_1, o_2, n_2, \_, \_) * true$ }
```

where

```
$D_1 \triangleq$ (
    d.AbsResult := d.$o_2$;
    if ([d.$a_1$] = d.$o_1$) {
        d.$r_2$ := d.$n_2$;
        Abs[d.$a_2$] := d.$n_2$;
    }
    else
        d.$r_2$ := Abs[d.$a_2$];
)
```

$$\begin{array}{lcl} D_2 & \triangleq & \textbf{skip} \\ D_3 & \triangleq & \text{lda2} := [\text{d}.a_2] \\ \widetilde{B} & \triangleq & \text{lda2} = \text{d} \end{array}$$

Fig. 9. Proof outline of `Complete(d)`

Acknowledgments. We thank the anonymous reviewers for their comments and suggestions. Zipeng Zhang, Xinyu Feng, Ming Fu and Yong Li are supported in part by National Natural Science Foundation of China (Grant No. 61073040, 61003043, 61170018 and 61103023), by Program for New Century Excellent Talents in Universities (NCET), and by the Fundamental Research Funds for the Central Universities. Zhong Shao is supported in part by NSF grants CNS 0915888 and CNS 0910670, and DARPA CRASH grant FA8750-10-2-0254.

References

[1] Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82, 253–284 (1991)

[2] Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proc. 38th ACM Symp. on Principles of Prog. Lang. (POPL 2011), pp. 399–410 (2011)

[3] Feng, X.: Local rely-guarantee reasoning. In: Proc. 36th ACM Symp. on Principles of Prog. Lang. (POPL 2009), pp. 315–327. ACM (2009)

[4] Harris, T., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: 16th International Symposium on Distributed Computing, pp. 265–279 (October 2002)

[5] Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. 5(4), 596–619 (1983)

[6] Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.D.: Network Invariants in Action. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 101–115. Springer, Heidelberg (2002)

[7] Marcus, M., Pnueli, A.: Using Ghost Variables to Prove Refinement. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 226–240. Springer, Heidelberg (1996)

[8] Sezgin, A., Tasiran, S., Qadeer, S.: Tressa: Claiming the Future. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 25–39. Springer, Heidelberg (2010)

[9] Vafeiadis, V.: Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory (July 2008)

[10] Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)

[11] Zhang, Z., Feng, X., Fu, M., Shao, Z., Li, Y.: A structural approach to prophecy variables. Technical report, University of Science and Technology of China (March 2012), `http://kyhcs.ustcsz.edu.cn/projects/concur/struct_prophecy`