A Proof System for MDESL

Jianyu Lu, Wanling Xie, Huibiao Zhu, Yuan Fei
Shanghai Key Laboratory of Trustworthy Computing,
School of Computer and Software Engineering,
East China Normal University, Shanghai, China

Abstract—Hardware description language (HDL) Verilog has been standardized and widely used in industry. To describe the features such as event-driven computation, time and shared-variable concurrency of hardware, a Verilog-like language MDESL (multithreaded discrete event simulation language), has been introduced. In this paper, we put forward a proof system for MDESL which is based on the classical Hoare Logic (precondition, program, postcondition). To deal with the guard statement, we add a new element trace to Hoare triples. We extend the primitives of expression to express the global time of current program, and interpret the triples so that it can verify both terminating and nonterminating computations. To verify a concurrent program, we use a merger method of the trace to combine the traces in our parallel rule. Finally, there is an example about using our proof system to verify the correctness of a program written by MDESL.

I. INTRODUCTION

With the increasing complexity of computer hardware, more and more modern hardware designs choose to use the hardware description language (HDL) to describe the designs at various levels of abstraction. As a high level programming language, HDL not only has the classical programming statements such as skip, assignments, conditionals, loops, but also has some extensions for real-time, concurrency, guard and new data structures appropriate for modelling hardware. The Verilog Hardware Description Language (Verilog HDL) became an IEEE standard in 1995 as IEEE std 1364-1995 [1], [2] due to its simple, intuitive and effective at multiple of abstraction. There are several important features in Verilog, including real-time [3], [4], event-driven computation, shared-variable concurrency and simulator-based interpretation.

MDESL [5] (Multithreaded Discrete Simulation Language) is a Verilog-like language [6], [7], [8]. Parts of the statements and constructs in MDESL are similar to those in C programming language. However, as a hardware level programming language, it also has the statements and constructs which can describe the features of hardware, such as event-driven computation, real-time and shared-variable concurrency. In MDESL, the guard statement @((g)) represents that a new state will be compared to its previous state, if the result satisfies g, then the program will continue to execute its rest statements, otherwise it will be in a state until the guard is triggered. This embodies the feature of the event-driven of MDESL. Time delay statement is introduced in MDESL, the synchronization of different parallel components can be based on time controls, and the parallel mechanism is an interleaving model.

In this paper, we put forward a proof system for MDESL in order to verify the correctness of the programs written by MDESL. Our proof system is based on classical Hoare Logic [9]. According to the semantic model of MDESL, we have added a data structure trace in the front of the triple and extended the assertion languages to form a new triple which is convenient for the compositional verification of MDESL. Trace [10] is used to record the new time and values when an atomic action raises a data update. Thus it can help us to deal with the shared-variable feature. As usual, the precondition can specify the set of initial or input states at the start of the execution, and the postcondition describes the set of final or output states at termination. To verify the property of time, we add a special variable time (similar to [11]) which represents the beginning time of the program in the precondition and the terminating time in the postcondition. We can specify the execution time of a program and merge the trace between different parallel components by the global clock time. In the classical Hoare Logic, we can only deal with the terminating program, but in our proof system, we can specify the terminating time by the use of time. If time ∈ [0, ∞) we can deduce that the program will terminate or if time = ∞ this means that the process will run forever. We can use the rules for nonterminating in our proof system to deal with this situation.

The remainder of this paper is organized as follows. In Section II, we introduce the program language and the semantic model of MDESL, and give the specifications of the assertions and some definitions used in our proof system. In Section III, we provide the proof system for MDESL, including the rules for sequential programs, parallel composition and nontermination computations. In addition, we introduce some auxiliary axioms and rules which are useful for the verification of the programs. In Section IV, we apply our proof system to verify one example. Section V concludes the paper.

II. BASIC FRAMEWORK

In this section we introduce the basic framework. We first introduce the syntax of MDSEL in subsection A. Then the semantic model to describe the shared-variable concurrency and real-time computations is given in subsection B. The formalism to specify a system which is described by MDESL is presented in subsection C.
The following forms are instantaneous:

1. \( x := e \) is an atomic assignment, whereas \( x := e \) is not.
2. \( \#n \) is time delay which suspends the execution for \( n \) time units, \( n \) is an integer.
3. An event guard \( \#(v) \) is triggered by the change of \( v \), and \( \#(\downarrow v) \) is triggered by a decrease in \( v \), however \( \#(\uparrow v) \) is triggered by an increase in \( v \).
4. \( \#(g_1) \text{ or } g_2 \) is triggered if \( \#(g_1) \) or \( \#(g_2) \) is triggered.
5. \( \#(g_1) \text{ and } g_2 \) is triggered if \( \#(g_1) \) and \( \#(g_2) \) are triggered simultaneously.
6. \( \#(g_1) \text{ and } \neg g_2 \) is triggered if \( \#(g_2) \) remains untriggered and \( \#(g_1) \) are triggered.

We describe the execution of a statement is instantaneous if the execution of a statement lasts zero time. In MDESL, the following forms are instantaneous:

1. \( x := e, \text{Skip}, \#(x := e) \) are instantaneous.
2. If \( P \) and \( Q \) are instantaneous, \( P; Q \) is also instantaneous.
3. The transition from \( \text{if } b \text{ then } P \text{ else } Q \) to \( P \text{or } Q \) is instantaneous.
4. The transition from \( \text{while } b \text{ do } P \text{ while } b \text{ do } P \) (or to Skip) is instantaneous.

### B. The Semantical Model

In this subsection, we will introduce the semantical model of MDESL. MDESL processes communicate with each other by shared variables. In order to record communications among them during execution, we use a trace of snapshots. When a process executes an atomic action, a snapshot will be added to the end of the trace. We use \( tr \) to denote that trace.

#### Definition 2.1 (Snapshot)

We use a triple \((t, \sigma, \mu)\) to denote a snapshot, which is used to specify the behaviour of an atomic action, where:

1. \( t \) represents the time when the atomic action happens.
2. \( \sigma \) represents the values of program variables when an atomic action is completed.
3. \( \mu \) denotes which process provides the status update. When \( \mu = 1 \), it represents the process itself performs the atomic action, \( \mu = 0 \) states the environment engages an atomic action.

We use the following projections to choose the components of a snapshot:

\[ \pi_1(t, \sigma, \mu) = t, \pi_2(t, \sigma, \mu) = \sigma, \pi_3(t, \sigma, \mu) = \mu \]

#### Definition 2.2 (Operators of Trace)

We present the following main operators \( OP \) among the trace. Let \( tr_1 \) and \( tr_2 \) be two traces, \( s \) and \( t \) be two snapshots.

\[ OP ::= \neg | \text{last} | \leq | \preceq | len \]

1. \( \neg tr_1 \) represents that \( tr_1 \) and \( tr_2 \) are connected.
2. \( \text{last}(tr_1) \) denotes the last snapshot of \( tr_1 \).
3. \( tr_1 \preceq tr_2 \) indicates that \( tr_1 \) is a prefix of \( tr_2 \), and we have \( \forall tr, tr \preceq tr \).
4. \( tr_2 - tr_1 \) indicates that the remain of removing all snapshots in \( tr_1 \) from \( tr_2 \) when \( tr_1 \preceq tr_2 \). Combined with the definition of \( tr_1 \neg tr_2 \), we can conclude that \( tr_2 = tr_1 \neg (tr_2 - tr_1) \).
5. \( len(tr_1) \) stands for the length of \( tr_1 \), i.e., if \( tr_1 \) contains two snapshots, then \( len(tr_1) = 2 \).

The Fig. 1. shows the trace behaviour of a process and its environment. Here, we use "\( s \)" to represent the process’s atomic action and "\( e \)" to stand for the environment’s atomic action. The numbers on the vertical line stand for the snapshots sequences in the process’s trace, the numbers on the horizontal line indicate the time when the atomic action happens.

#### Example 2.1

Let \( P = \#(x := 1; \#1; x := 2) \) and the initial trace of \( P \) is \( tr_1 \). We assume that the initial time is \( time_0 \).

When \( P \) completes its first atomic action \( x := 1 \), a snapshot \((time_0, \sigma(x := 1), 1)\) will be added to the end of \( tr_1 \). And we use \( tr_2 \) to denote the new trace \( tr_1 \neg (time_0, \sigma(x := 1), 1) \).
After one time unit, the atomic action $x := 2$ takes place, which generates a snapshot $(\text{state}_0 + 1, \sigma(x := 2), 1)$ attached to the end of the trace $tr_2$. And we use $tr_3$ to stand for the new trace $tr_2 \cap (\text{state}_0 + 1, \sigma(x := 2), 1)$. According to Definition 2.2, we have

- the starting time of $S$,
- the initial values of the common variables of $S$.

Assertion $p$ expresses precondition described as below:

- if the starting time of $S$ ($\infty$ if $S$ does not terminate),
- the final values of the common variables of $S$ if $S$ terminates.

We can use $tr$ to record the second behaviour, as described above, it’s a sequence of snapshots to record communications among them during execution. In order to record the global clock, we use a special variable time, ranging over $\text{TIME} \cup \{\infty\}$, here $\text{TIME}$ is a time domain which is discrete and $\text{TIME} = \{x | x \in \mathbb{N}\}$. We use $\sigma$, $\sigma_0$, $\sigma_1$, ... to represent states, assigns a value from $\mathbb{R}$ to a common variable and assigns a value from $\text{TIME} \cup \{\infty\}$ to time variable.

A set of pairs of the form $(\sigma, tr)$ represents the semantics of a program $P$ starting in a state $\sigma_0$ denoted by $\mathcal{M}(P)(\sigma_0)$. Here $\sigma$ is a state and $tr$ is the trace of $P$, as we defined above, $\sigma_0(x)$ is the value of common variable $x$ at the start of the $P$ and $\sigma_0(\text{time})$ represents the starting time, $tr_0$ denotes the initial trace when the program $P$ begins to execution. If $P$ terminates and a pair $(\sigma, tr)$ is in $\mathcal{M}(P)(\sigma_0)$, then the value of $\sigma(\text{time})$ represents the termination time. If $P$ does not terminate then $\sigma(\text{time}) = \infty$ and $\sigma(x)$ is an arbitrary value, here $x$ is a common variable.

### C. Specifications

Our specifications are based on classical Hoare triples $\{p\} S \{q\}$, it has the following meaning: if $S$ is executed in a state satisfying precondition $p$ and $S$ terminates then the final state satisfies postcondition $q$. In MDESL, $\delta(g)$ statement needs to compare current state with the earlier state, the classical Hoare triples are not suitable for it. According to the semantic model of MDESL mentioned in subsection B, we can use the trace to help us to solve the problem. Thus the formula has the new form $tr : \{p\} S \{q\}$ where $tr$ represents the initial trace before $S$ executes its first statement, $p$ and $q$ are assertions and $S$ is a program.

Assertion $p$ expresses precondition described as below:

- the starting time of $S$,
- the initial values of the common variables of $S$.

Assertion $q$ expresses the postcondition described as follows:

- the terminating time of $S$ ($\infty$ if $S$ does not terminate),
- the final values of the common variables of $S$ if $S$ terminates.

Compared with classical Hoare triples, we add a special variable time in assertions, so our proof system can deal with total correctness as well as partial correctness. Then we will give some useful notations which will be used in our following proof system.

**Definition 2.3** If a guard $g$ of a program $S$ is trigged and the current trace of $S$ is $tr$, we can denote it as $\mathcal{M}(g)(tr)$. Let $tr = \{p\} S \{q\}$ and $tr_0 = \{p_0\} S_0 \{q_0\}$. Let $\mathcal{M}(g)(P)(\sigma_0)$ be the semantics of $P$ starting in state $\sigma_0$. If $\sigma_0$ is an initial state of $P$, then $\mathcal{M}(g)(P)(\sigma_0)$ is the lemma of $\mathcal{M}(g)(P)(\sigma_0)$.

**Definition 2.4** If a guard $g$ of a program $S$ is not trigged until the trace of $S$ is $tr$, and the beginning trace of $S$ is $tr_0$. During this period, denoted by $\mathcal{M}(g)(tr_0, tr_1)$.

**Definition 2.5** If a guard $g$ of a program $S$ is trigged during $tr_0$ and $tr_1$, this period is represented by $\mathcal{M}(g)(tr_0, tr_1)$.

**Definition 2.6 (Validity)** For a program $S$, the beginning trace is $tr_0$, the program $S$ and assertions $p$ and $q$, if the correctness formula $\mathcal{M}(P)(\sigma_0)$ is true, we can write $\vdash tr_0 : \{p\} S \{q\}$, iff for the initial state $\sigma_0$, and any $\sigma, tr$ with $(\sigma, tr) \in \mathcal{M}(P)(\sigma_0)$, we have that $(\sigma_0, tr_0) \models \mathcal{M}(g)(tr_0, tr_1)$.
III. The Proof System

In this section, we will introduce a compositional proof system for MDESL. First we give the proof rules of the sequential program and some axioms that are generally applicable to each statement in subsection A. Then in subsection B, the rules for parallel composition are presented. Last, we will introduce some auxiliary axioms and rules in subsection C.

A. Axioms and Sequential Program Rules

A skip statement means the program does nothing and terminates immediately, it will have no effect on itself and the environment.

**Axioms 1. Skip**

\[ tr : \{ p \} \text{Skip} \{ p \} \]

The Chaos statement means that the behaviour of the program is totally unpredictable and the global clock will not stop, we use a notion \( time = \infty \) to represent that the program is divergence.

**Axioms 2. Chaos**

\[ tr : \{ p \} \text{Chaos} \{ q \land time = \infty \} \]

The nontermination axiom represents that a program following a Chaos computation has no effect.

**Axioms 3. Nontermination**

\[ tr : \{ p \land time = \infty \} S \{ p \land time = \infty \} \]

The rule for an assignment \( x := e \) is same as the classical rule because the assignment statement takes 0 time unit to complete.

**Axioms 4. Assignment**

\[ tr : \{ q[x := e] \} x := e \{ q \} \]

About the rule for delay statement \( \#e \), which means that the global clock \( time \) delays \( e \) time units and no change takes place in common variables. We give the postcondition \( q \), then the precondition \( q = e + \) is required.

**Axioms 5. Delay**

\[ tr : \{ q[time = time + e]\} \#e \{ q \} \]

About the rule for the \( @ (g) \) statement, there are two possibilities in sequential program. One possibility is that the guard is triggered by the execution of its prior atomic action (or it may be triggered by its environment and we will discuss it in subsection B). In this case, the notation \( trig(g) \) at \( tr \) is true, due to no variables are updated, the postcondition and the precondition are same. The other possibility is that the execution of the program can not trig the guard \( @ (g) \), and the guard will be in a waiting state to be fired endlessly, which means the state of the program becomes Chaos. We use a notion \( q_{\infty} \) to represent a nonterminating computation of infinite waiting.

**Rule 1. Guard -1**

\[ (p \land time < \infty) \land trig(g) \land tr \rightarrow p \]

\[ (p \land time < \infty) \land \neg trig(g) \land tr \rightarrow q_{\infty} \]

**Rule 2. Conditional**

\[ tr : \{ p \land b \} S_1 \{ q \}, tr : \{ p \land \neg b \} S_2 \{ q \} \]

\[ tr : \{ p \} \text{if } b \text{ then } S_1 \text{ else } S_2 \{ q \} \]

About the rule for the while construct, it has two parts. The first part is related to the classic rule of Hoare Logic. The second part is to handle the nonterminating statements.

**Rule 3. While**

\[ tr : \{ I \land b \land time < \infty \} S \{ I \} \]

\[ (\forall tr_1, \exists tr_2, tr \leq tr_1 \leq tr_2) \rightarrow q_{\infty} \]

\[ S := \text{Skip} \rightarrow q_{\infty} \]

\[ I \rightarrow I_1, \]

\[ (\forall tr_1, \exists tr_2 > t_1 : I_1[t_2/time]) \rightarrow q_{\infty} \]

\[ tr : \{ I \} \text{while } b \text{ do } S \text{ od } \{ (I \land \neg b) \lor (q_{\infty} \land time = \infty) \} \]

We will give an informal description of the soundness of the While rule. For a while program \( b \text{ do } S \text{ od } \), we assume that it starts in a state satisfying \( p \). There are four cases.

The first is the same as the classic Hoare Logic. Program \( S \) is a terminating computation, and the loop is terminated. Except the last one, for all these computations of \( S, b \) is always true. So the condition \( tr : \{ I \land b \land time < \infty \} S \{ I \} \) holds in the case, this means that the last computation \( \neg b \) must be true, which leads to \( (I \land \neg b) \).

In the second case, we assume that it starts in a state satisfying \( I \) and nonterminating, i.e., for the initial state \( \sigma_0 \), \( \sigma_0(time) = \infty \). Then model is the same as the nonterminating model (the property has been expressed in the Nontermination Axiom). \( time = \infty \) and \( I \rightarrow I_1 \land time = \infty \) hold in this model, so \( (\forall tr_1, \exists tr_2 > t_1 : I_1[t_2/time]) \) is true. And it leads to \( q_{\infty} \).

In the third case, we assume \( S \) is a nonterminating computation. Then the computation becomes a nonterminating computation, and as in the first case, \( I \land time = \infty \) holds for this model. Thus, since \( I \rightarrow I_1 \), condition \( (\forall tr_1, \exists tr_2 > t_1 : I_1[t_2/time]) \) holds in the case, and it will lead to \( q_{\infty} \).

The last case represents a nonterminating computation which program \( S \) is a terminating computation and the loop is infinite. It means that the boolean condition \( b \) will be true forever. In this case, program \( S \) has three situations.

- If program \( S \) is a \( \text{Skip} \) statement, then it will lead to \( q_{\infty} \).
- If it doesn’t contain any time delay, the trace of the while program will be infinite. Hence, we obtain \( (\forall tr_1, \exists tr_2, tr \leq tr_1 \leq tr_2) \rightarrow q_{\infty} \), and then it leads to \( q_{\infty} \).
- If \( S \) contains any time delay \( \#e \), each computation of \( S \) takes at least \( e \) time units, so we have \( I \rightarrow I_1 \), hence, \( (\forall tr_1, \exists tr_2 > t_1 : I_1[t_2/time]) \rightarrow q_{\infty} \), and we obtain \( q_{\infty} \).
B. Parallel Composition

Before we give the rule of parallel composition, we will first introduce the merge of traces. Now consider the following example.

Example 3.1. Let \( P =_{df} x := y + 2; \#1; y := x + 1 \) and \( Q =_{df} y := y + 2 \). Assume that \( P \mid Q \) is activate with \( x = y = 0 \) and \( \text{time} = 0 \). If \( P \) is scheduled to execute first, then the sequence of snapshots of \( P \) is:

\[
\text{seq}_P = \langle 0, \{ x = 2, y = 0 \}, 1 \rangle, \\
0, \{ x = 2, y = 2 \}, 1, \\
1, \{ x = 2, y = 3 \}, 1 >
\]

where the first and the third snapshots are produced by the atomic action \( x := y + 2 \) and \( y := x + 1 \) of \( P \). And the second one is engaged by the environment of \( P \). In this example, the environment of \( P \) and the computation of \( Q \) yields the following sequence:

\[
\text{seq}_Q = \langle 0, \{ x = 2, y = 0 \}, 0 \rangle, \\
0, \{ x = 2, y = 2 \}, 1, \\
1, \{ x = 2, y = 3 \}, 0 >
\]

Due to \( \text{seq}_P \) and \( \text{seq}_Q \) are built from the same initial state, they are comparable. In addition, all of their snapshots are made by both \( P \) and \( Q \). So their merge rises a trace of \( P \mid Q \):

\[
\text{seq}_{PQ} = \langle 0, \{ x = 2, y = 0 \}, 1 \rangle, \\
0, \{ x = 2, y = 2 \}, 1, \\
1, \{ x = 2, y = 3 \}, 1 >
\]

If \( Q \) is executed first, then the traces of \( P \) and \( Q \) are:

\[
\text{seq}_P = \langle 0, \{ x = 0, y = 2 \}, 0 \rangle, \\
0, \{ x = 2, y = 1 \}, \\
1, \{ x = 2, y = 3 \}, 1 >
\]

\[
\text{seq}_Q = \langle 0, \{ x = 0, y = 2 \}, 1 \rangle, \\
0, \{ x = 2, y = 2 \}, 0, \\
1, \{ x = 2, y = 3 \}, 0 >
\]

Their trace of \( P \mid Q \) is their merge. The trace is:

\[
\text{seq}_{PQ} = \langle 0, \{ x = 0, y = 2 \}, 1 \rangle, \\
0, \{ x = 2, y = 2 \}, 1, \\
1, \{ x = 2, y = 3 \}, 1 >
\]

Definition 3.1 (Merge of Traces)) As we have seen in Example 3.1, two sequences \( \text{seq}_1 \) and \( \text{seq}_2 \) are said to be comparable if:

1. The time sequences from the two traces are the same
2. They are built from the same sequence of states
3. None of their snapshot is made by both components

We use the following predicate to present their merge:

\[
M(\text{seq}, \text{seq}_1, \text{seq}_2) =_{df} \\
(\pi_1(\text{seq}_1) = \pi_1(\text{seq}_2)) \land \\
(\pi_2(\text{seq}_1) = \pi_2(\text{seq}_2)) \land \\
(\pi_3(\text{seq}) = \pi_3(\text{seq}_1) + \pi_1(\text{seq}_2) \land \\
2 \notin \pi_3(\text{seq}_1) + \pi_3(\text{seq}_2))
\]

In the sequential programs, the \( (\&\text{g}) \) can only be trigged by the execution of its prior atomic action, but in the parallel programs, it also can be trigged by its environment, and the Rule 1(Guard-1) will be replaced by the following rule.

Rule 4. Guard -2

\[
(p \land \text{time} < \infty) \land \text{trig}(\text{g}) \Rightarrow \text{tr} \rightarrow p
\]

The second property of the rule notes that, in the guard statement, if the \( (\&\text{g}) \) is trigged by itself, the atomic action \( (\&\text{g}) \) will be scheduled immediately. This means that only if the guard cannot be trigged by itself, then it will be waiting to be trigged by it’s environment. If the guard is fired, then the guard will complete it’s computation, and the last snapshot of the trace records the terminating time and the final values of the program, so the postcondition is \( q[\pi_1(\text{last}(\text{tr})]/\text{time}, \pi_2(\text{last}(\text{tr})]/\text{sigma}] \).

The proof rule for parallel composition has the following form where we use a merge operator \( M \) to combine the two traces, and use the trace which has been merged we can combine two assertions.

Rule 5. Parallel

\[
\text{tr} : \{ p_1 \} S_1 \{ q_1 \}, \text{ tr} : \{ p_2 \} S_2 \{ q_2 \}, \\
\forall \text{tr}_1, \text{tr}_2 \bullet (M(\text{tr}_3, \text{tr}_1, \text{tr}_2) \land (\text{tr}_1 \rightarrow q_1) \land (\text{tr}_2 \rightarrow q_2)) \\
\Rightarrow q[\text{maxTime}(q_1, q_2)/\text{time}, \pi_2(\text{last}(\text{tr}_3)]/\text{sigma}] \\
\text{tr} : \{ p_1 \land p_2 \} S_1 \parallel S_2 \{ q \}
\]

where \( \text{tr}_1 \) is the trace of \( S_1 \) and \( \text{tr}_2 \) is the trace of \( S_2 \) when \( S_1 \parallel S_2 \).

The \( \text{tr} \rightarrow q \) represents that the assumption \( q \) satisfies the state \( \pi_2(\text{last}(\text{tr}]) \), and the trace will not record the delay statement, so if a statement is followed by a time delay, it will not change the trace. For instance, consider a program \( P : x := x + 1; \#1; x := x + 2 \) with the initial state \( x = 0 \) and the initial time is 0, and the sequence of the snapshots of \( P := \langle 0, x = 1, 1 \rangle, (1, x = 3, 1) > \). When we add a time delay to the end of \( P \), the trace of \( P \) will not change anything. So we define:

\[
\text{tr} \rightarrow q =_{df} \pi_2(\text{last}(\text{tr}]) \land \text{time} < \infty \rightarrow q
\]

Due to the termination times of \( S_1 \) and \( S_2 \) will be different. To obtain a general rule, we use the notation \( \text{maxTime}(q_1, q_2) \) to denote the termination time of \( S_1 \parallel S_2 \), the definition is given as below:

\[
\text{maxTime}(q_1, q_2) =_{df} \text{max}(t_1, t_2) (t_i \text{ is the value of time}
\]
in $q_i$.

C. Auxiliary Axioms and Rules

In this subsection, we will introduce some auxiliary axioms and rules which will be used in our proof system. Some of them have been presented in [9].

Axiom 6. Invariance

\[ tr : \{p\} S \{p\} \]

where \( time \) does not occur in \( p \), and \( var(S) \cap var(p) = \phi \)

Rule 6. Disjunction

\[ tr : \{p\} S \{q\}, tr : \{r\} S \{q\} \]

\[ tr : \{p \lor r\} S \{q\} \]

Rule 7. Conjunction

\[ tr : \{p_1\} S \{q_1\}, tr : \{p_2\} S \{q_2\} \]

\[ tr : \{p_1 \land p_2\} S \{q_1 \land q_2\} \]

The substitution rule means that if a variable does not occur in the program statement, we can use any arbitrary expression to replace it.

Rule 8. Substitution

\[ tr : \{p\} S \{q\} \]

\[ tr : \{p[z := t]\} S \{q[z := t]\} \]

where \((z \cup var(t)) \cap changes(S) = \phi \) and \( time \) does not occur in \( t \).

Rule 9. Consequence

\[ tr : \{p\} S \{q\}, p_1 \rightarrow p, q \rightarrow q_1 \]

\[ tr : \{p_1\} S \{q_1\} \]

About the rule for sequential consequence, the construct is same as the classic rule of Hoare Logic. In our proof system, we use \( trace \) to help us record the state before a statement begins it’s first statement, and as we described in the parallel rule, \( trace \) does not record the delay statement, so we assume that \( S_1 \) ends with a delay statement \#e \((0 \leq e \leq \infty)\).

Rule 10. Sequential Consequence

\[ tr : \{p\} S_1 \{x\}, tr_1 : \{r\} S_2 \{q\} \]

\[ tr : \{p\} S_1; S_2 \{q\} \]

where \( tr \preceq tr_1, tr_1 \rightarrow r \) and \( \pi_3(last(tr)) + e = t_r. \) (\( t_r \) is the value of time in \( r \)).

IV. CASE STUDY

In this section we give a parallel program written by MDESL, and show how to apply our method to prove the correctness of the program. Consider the program \( P \) and \( Q \):

\[ P ::= \begin{array}{l}
\text{while } x > 0 \text{ do} \\
\quad @y; \\
\quad x := x - 1; \\
\quad \text{ od; } \\
\quad z = 1;
\end{array} \]

\[ Q ::= \#1; \]

\[ \begin{array}{l}
\text{while } z \neq 1 \text{ do} \\
\quad y := y + 1; \\
\quad \#1;
\end{array} \]

Program \( P \) represents that the variable \( x \) will decrease if \( x > 0 \) and variable \( y \) increases. If \( x \leq 0 \), it terminates. Program \( Q \) denotes that \( y \) increases by 1 per one time unit when \( z \neq 1 \) is true, so when \( P \) terminates, it satisfies \( z = 1 \).

We have the following assumptions and notations:

- the initial trace \( tr \) is \(< (0, (x = 2, y = 0, z \neq 1), 1) > \).
- \( x, y, z \) are all integers.
- the precondition \( p_1 \) is \( x = 2 \land y = 0 \land z \neq 1 \land time = 0 \).
- \( S_1 = (@y; x := x - 1) \) and \( S_2 = (y := y + 1; \#1) \).

For \( P || Q \), we want to prove the following correctness formulas:

\[ tr : \{p_1\} P \{x \leq 0 \land time < \infty\} \]

\[ tr : \{p_1\} Q \{z = 1 \land time < \infty\} \]

\[ tr : \{p_1\} P || Q \{x \leq 0 \land z = 1 \land time < \infty\} \]

We denote \( x \leq 0 \land time < \infty \) as \( q_1 \), \( z = 1 \land time < \infty \) as \( q_2 \) and \( x \leq 0 \land z = 1 \land time < \infty \) as \( q \).

Proof:

First, as same as the Example 3.1, we get the traces of \( P \) and \( Q \) when \( P || Q \):

\[ tr_p =< (0, \{x = 2, y = 0, z \neq 1\}, 1), \]

\[ (1, \{x = 0, y = 1, z \neq 1\}, 0) \]

\[ (1, \{x = 1, y = 1, z \neq 1\}, 1) \]

\[ (2, \{x = 1, y = 2, z \neq 1\}, 0) \]

\[ (2, \{x = 0, y = 2, z \neq 1\}, 1) \]

\[ (2, \{x = 0, y = 2, z = 1\}, 1) > \]

\[ tr_q =< (0, \{x = 2, y = 0, z \neq 1\}, 1), \]

\[ (1, \{x = 0, y = 1, z \neq 1\}, 1) \]

\[ (1, \{x = 1, y = 1, z \neq 1\}, 0) \]

\[ (2, \{x = 1, y = 2, z \neq 1\}, 1) \]

\[ (2, \{x = 0, y = 2, z \neq 1\}, 0) \]

\[ (2, \{x = 0, y = 2, z = 1\}, 0) > \]

According to the definition of \( \text{Merge} \), we obtain the trace of \( P || Q \):

\[ tr_{p||q} =< (0, \{x = 2, y = 0, z \neq 1\}, 1), \]

\[ (1, \{x = 0, y = 1, z \neq 1\}, 1) \]

\[ (1, \{x = 1, y = 1, z \neq 1\}, 1) \]

\[ (2, \{x = 1, y = 2, z \neq 1\}, 1) \]

\[ (2, \{x = 0, y = 2, z \neq 1\}, 1) \]

\[ (2, \{x = 0, y = 2, z = 1\}, 1) > \]

Then we prove the correctness of (4.1) and (4.2) by using their trace \( tr_p \) and \( tr_q \).
To prove (4.1), we set a global invariant variable \( I_1 \) as:
\[
I_1 = z \neq 1 \land \text{time} < \infty.
\]
And we need to prove the correctness of following formula:
\[
tr : \{ I_1 \land x > 0 \} \oplus (\uparrow y) \{ I \land x > 0 \}
\]
and
\[
tr_{p2} : \{ I_1 \land x > 0 \} \oplus (\uparrow y) \{ I \land x > 0 \}
\]
Note that the implications
\[
I_1 \land (\exists tr_{p1} : tr \leq tr_{p1} \land \\
\text{await}(g) \text{ during } [tr, tr_{p1}] \land (trig(g) \text{ at } tr_{p1}))
\]
\[
\rightarrow I_1 \land x > 0
\]
and
\[
I_1 \land (\exists tr_{p3} : tr_{p2} \leq tr_{p3} \land \\
\text{await}(g) \text{ during } [tr_2, tr_{p3}] \land (trig(g) \text{ at } tr_{p3}))
\]
\[
\rightarrow I_1 \land x > 0
\]
hold. Thus by the rule 4 (Guard -2), we prove (4.4) and (4.5).
Further by the assignment axiom 4, we prove
\[
\{ I \land x > 0 \} : \{ x := x - 1 \} \land \{ I \land x > 0 \}
\]
(4.6)
By (4.4), (4.5), (4.6) and the while rule 3, we obtain
\[
tr : \{ I \} \text{ while } x > 0 \text{ do } S_1 \text{ od } \{ I \land x \leq 0 \}
\]
(4.7)
By the consequence rule 9, we can prove that (4.1) is correct.
We can use the same method to prove (4.2) is correct too.

At last we prove (4.3). Since
\[
\pi_2(\text{last}(tr_p)) \land \text{time} < \infty \rightarrow q_1
\]
and
\[
\pi_2(\text{last}(tr_q)) \land \text{time} < \infty \rightarrow q_2.
\]
\[
tr_p \rightarrow q_1 \text{ and } tr_q \rightarrow q_2 \text{ are hold in this model, and using the definition of maxtime and } \pi_2(tr), \text{ we get}
\]
\[
\text{maxtime}(q_1, q_2) = q \text{ time} \leq \infty
\]
and
\[
\pi_2(\text{last}(tr_p||q)) = \{ x = 0, y = 2, z = 1 \}.
\]
We obtain
\[
\forall tr_p, tr_q : (M(tr_p||q, tr_p, tr_q) \land (tr_p \rightarrow q_1) \land (tr_q \rightarrow q_2))
\]
\[
\rightarrow q \{ \text{maxtime}(q_1, q_2) \}, \pi_2(\text{last}(tr_q))/\text{time}, \pi_2(\text{last}(tr_p))/\sigma
\]
(4.8)
Combine formula (4.1), (4.2) and (4.8) and by the Parallel rule, we can derive the correctness formula:
\[
tr : \{ p_1 \} P||Q \{ x \leq 0 \land z = 1 \land \text{time} < \infty \}
\]

V. CONCLUSION

In this paper, we have presented a proof system for MDESL (Multithreaded Discrete Simulation Language) which aims to prove the correctness of MDESL. Compared to classical Hoare Logic, our proof system uses the global clock variable time to express the real-time feature. It represents the starting time in the precondition and the terminating time in the postcondition. The value of time also can help us handle the nonterminating computations. And in order to verify the feature of event-driven we have extended the old triple \( \{ p \} s \{ q \} \) to \( tr : \{ p \} s \{ q \} \) by adding a data structure \( tr \). To make the verification more straightforward, we have provided some axioms and rules for sequential. Then we give the composition rules which make our proof system become a complete system. Our key contribution is the guard rule. It can be applied in the verification of the property of event-driven in MDESL.

For the future, we want to explore our proof system for hardware description language (HDL). On one hand, the probability feature [12] has been proposed in a new Verilog-like language PTSC [13]. We can deduce some specific rules to describe the feature and verify the properties related to probability. On the other hand, we want to link our proof system with the semantics (operational, denotational, algebraic) [14], [15] respectively for MDESL. Furthermore, we want to implement the proof system in a tool so that it can verify the correctness of programs automatically.

ACKNOWLEDGE

This work was partly supported by Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No. ZF1213).

REFERENCES