Communication papers of the Federated Conference on Computer Science and Information Systems, pp. 387–393 DOI: 10.15439/2017F400 ISSN 2300-5963 ACSIS, Vol. 13

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 assertion 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 as 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 trigged. 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 sematic model of MDSEL, 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 express 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 $time \in [0, \infty)$ we can deduce that the program will terminate or if $time = \infty$ 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 sematic 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.

$P := PC$	primitive commands
$\mid P; Q$	sequential composition
$\mid$ if b then P else Q	conditional construct
$\mid$ while b do P	iteration construction
$\mid P\|\|Q$	parallel composition
$\mid M$	hybrid control
$M := @(x := e) \mid @(g) \mid \#n$	hybrid control
$g := \eta \mid g$ or $g \mid g$ and $g \mid g$ and $\neg g$	logic connection
$\eta := v \mid \uparrow v \mid \downarrow v$	event guard

TABLE I
SYNTAX OF MDESL

A. Programming Language

In this subsection we introduce the syntax of Multithreaded Discrete Event Simulation Language (MDESL), first put forward by Zhu, a Verilog-like language, which not only has real-time feature but also supports the features of shared-variable concurrency and event-driven computation. The syntax is given in TABLE I.

We can explain the syntax (similar to [7], [8]) as follows :

- PC consists of four primitive commands: **Chaos**, **Stop**, **Skip**, $x := e$. **Chaos** represents the worst process, whose behaviour is totally unpredictable. **Stop** is the process that does nothing, in other word, idle process. $x := e$ is the assignment statement, which executes instantaneously. **Skip** behaves the same as $x := x$.
- $P; Q$ is sequential composition. It executes process P first. The process Q starts to executes after P terminates successfully.
- if b then P else Q is the conditional construct.
- while b do P is the iteration construct.
- $P||Q$ is parallel composition. In Verilog, parallel composition can occur only at the outmost level, here we allow it to occur anywhere.
- To accommodate the expansion laws of the parallel construct, we introduce the concept of guarded choice and extend Verilog's event category into the language.

(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) $@(g_1\ or\ g_2)$ is triggered if $@(g_1)$ or $@(g_2)$ is triggered.
(5) $@(g_1\ and\ g_2)$ is triggered if $@(g_1)$ and $@(g_2)$ is triggered simultaneously.
(6) $@(g_1\ and\ \neg g_2)$ 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) $x := e, Skip, @(x := e)$ are instantaneous.
(2) If P and Q are instantaneous, $P; Q$ is also instantaneous.
(3) The transition from $if\ b\ then\ P\ else\ Q$ to P(or Q) is instantaneous.
(4) The transition from $while\ b\ do\ P$ to $P; while\ b\ do\ P$ (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, σ, μ) 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) σ represents the values of program variables when an atomic action is completed.
(3) μ denotes which process provides the status update. When μ=1, it represents the process itself performs the atomic action, μ=0 states the environment engages an atomic action.

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

$$\pi_1(t, \sigma, \mu) =_{df} t\ ,\ \pi_2(t, \sigma, \mu) =_{df} \sigma\ ,\ \pi_3(t, \sigma, \mu) =_{df} \mu$$

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.

$$OP ::= \ ^\frown \mid last \mid \preceq \mid - \mid len$$

(1) $tr_1{}^\frown tr_2$ represents that tr_1 and tr_2 are connected.
(2) $last(tr_1)$ 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 $\forall tr,\ \emptyset \preceq tr$.
(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 $tr_1{}^\frown tr_2$, we can conclude that $tr_2 = tr_1{}^\frown(tr_2 - tr_1)$.
(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 "•" to represent the process's atomic action and "∘" 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 $P =_{df} (x := 1;\ \#1;\ x := 2)$ 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 $tr_1{}^\frown(time_0, \sigma(x := 1), 1)$.

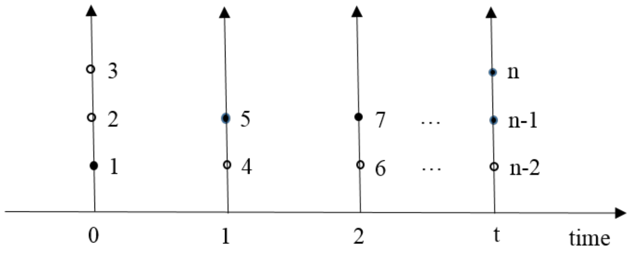

Fig. 1. The trace behaviour of a process

After one time unit, the atomic action $x := 2$ takes place, which generates a snapshot $(time_0+1, \sigma(x := 2), 1)$ attached to the end of the trace tr_2. And we use tr_3 to stand for the new trace $tr_2\frown(time_0 + 1, \sigma(x := 2), 1)$. According to Definition 2.2, we have

$tr_1 \preceq tr_2,\ tr_2 \preceq tr_3;$
$tr_2 - tr_1 = (time_0, \sigma(x := 1), 1);$
$tr_3 - tr_2 = (time_0 + 1, \sigma(x := 2), 1);$
$tr_3 - tr_1 =$
$\quad (time_0, \sigma(x := 1), 1)\frown(time_0 + 1, \sigma(x := 2), 1);$
$last(tr_2) = (time_0, \sigma(x := 1), 1);$
$last(tr_3) = (time_0 + 1, \sigma(x := 2), 1);$
$len(tr_3 - tr_2) = 1;\ len(tr_3 - tr_1) = 2;\ len(tr_2 - tr_1) = 1.$

Now, we can describe the real-time behaviour of a process P by the following aspects:

- the initial state, i.e., the values of the variables and the starting time at the start of the execution.
- the time and the values when the variables of P are changed.
- if P terminates, the final state, i.e., the values of the variables and the termination time at the end of the execution.

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 $TIME \bigcup \{\infty\}$, here $TIME$ is a time domain which is discrete and $TIME = \{x | x \in \mathbb{N}\}$. 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 $TIME \bigcup \{\infty\}$ to $time$ variable.

A set of pairs of the form (σ, tr) represents the semantics of a program P starting in a state σ_0 denoted by $\mathscr{M}(P)(\sigma_0)$. Here σ 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 $\sigma_0(time)$ 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 $\mathscr{M}(P)(\sigma_0)$, then the value of $\sigma(time)$ represents the termination time. If P does not terminate then $\sigma(time) = \infty$ 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, $@(g)$ 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 (∞ if S does not terminate),
- the finial 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 $trig(g)\ at\ tr$.

$trig(g)\ at\ tr\ =_{df}\ \exists tr_1 \bullet$
$\quad tr_1 \preceq tr\ \wedge$
$\quad len(tr - tr_1) = 1\ \wedge$
$\quad fire(g)(\pi_2(last(tr_1)), \pi_2(last(tr)))$

where $fire(g)(\sigma_0, \sigma_1)$ represents the transition from state σ_0 to σ_1 can awake the guard $@g$.

Definition 2.4 If a guard g of a program S is not trigged until the $trace$ of S is tr_1, and the beginning $trace$ of S is tr_0. During this period, denoted by $await(g)\ during\ [tr_0,\ tr_1)$.

$await(g)\ during\ [tr_0,\ tr_1)\ =_{df}\ \forall tr_2, tr_3\ \bullet$
$\quad tr_0 \preceq tr_2 \preceq tr_3 \preceq tr_1\ \wedge$
$\quad len(tr_3 - tr_2) = 1\ \wedge$
$\quad \neg fire(g)(\pi_2(last(tr_2)), \pi_2(last(tr_3)))$

Definition 2.5 If a guard g of a program S is trigged during tr_0 and tr_1, this period is represented by $trig(g)\ during\ [tr_0, tr_1]$.

$trig(g)\ during\ [tr_0, tr_1]\ =_{df}\ \exists tr_2\ \bullet$
$\quad tr_0 \preceq tr_2 \preceq tr_1\ \wedge$
$\quad await(g)\ during\ [tr_0, tr_2)\ \wedge$
$\quad trig(g)\ at\ tr_2$

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 $tr_0 : \{p\}\ S\ \{q\}$ is true, we can write $\models tr_0 : \{p\}\ S\ \{q\}$, iff for the initial state σ_0, and any σ, tr with $(\sigma, tr) \in \mathscr{M}(P)(\sigma_0)$, we have that $(\sigma_0, tr_0) \models p$ implies $(\sigma, tr) \models q$.

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

$$tr : \{p\}\ Skip\ \{p\}$$

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

$$tr : \{p\}\ Chaos\ \{q \wedge time = \infty\}$$

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

Axioms 3. Nontermination

$$tr : \{p \wedge time = \infty\}\ S\ \{p \wedge time = \infty\}$$

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 $q[time = time + e]$ 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_∞ to represent a nonterminating computation of infinite waiting.

Rule 1. Guard -1

$$\frac{\begin{array}{l}(p \wedge time < \infty) \wedge trig(g)\ at\ tr\ \rightarrow p\\ (p \wedge time < \infty) \wedge \neg trig(g)\ at\ tr \rightarrow q_\infty\end{array}}{tr : \{p\}\ @(g)\ \{p \vee q_\infty\}}$$

Rule 2. Conditional

$$\frac{tr : \{p \wedge b\}\ S_1\ \{q\},\ tr : \{p \wedge \neg b\}\ S_2\ \{q\}}{tr : \{p\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}}$$

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

$$\frac{\begin{array}{l}tr : \{I \wedge b \wedge time < \infty\}\ S\ \{I\}\\ (\forall tr_1, \exists tr_2, tr \preceq tr_1 \preceq tr_2) \rightarrow q_\infty\\ S := Skip \rightarrow q_\infty\\ I \rightarrow I_1,\\ (\forall t_1, \exists t_2 > t_1 : I_1[t_2/time]) \rightarrow q_\infty\end{array}}{tr : \{I\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{(I \wedge \neg b) \vee (q_\infty \wedge time = \infty)\}}$$

We will give an informal description of the soundness of the While rule, For a while program **while** b **do** S **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 $tr : \{I \wedge b \wedge time < \infty\}\ S\ \{I\}$ holds in the case, this means that the last computation $\neg b$ must be true, which leads to $(I \wedge \neg b)$.

In the second case, we assume that it starts in a state satisfying I and nonterminating, i.e., for the initial state σ_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 $I \rightarrow I_1 \wedge time = \infty$ hold in this model, so $(\forall t_1, \exists t_2 > t_1 : I_1[t_2/time])$ is true. And it leads to q_∞.

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

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 $Skip$ statement, then it will lead to q_∞.
- If it doesn't contain any time delay, the trace of the while program will be infinite. Hence, we obtain $(\forall tr_1, \exists tr_2, tr \preceq tr_1 \preceq tr_2) \rightarrow q_\infty$, and then it leads to q_∞.
- 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, $(\forall t_1, \exists t_2 > t_1 : I_1[t_2/time]) \rightarrow q_\infty$, and we obtain q_∞.

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 $P \mid\mid Q$ is activate with $x = y = 0$ and $time = 0$. If P is scheduled to execute first, then the sequence of snapshots of P is :

$$\begin{aligned} seq_p =< & (0, \{x = 2, y = 0\}, 1), \\ & (0, \{x = 2, y = 2\}, 0), \\ & (1, \{x = 2, y = 3\}, 1) > \end{aligned}$$

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 is Q and the computation of Q yields the following sequence :

$$\begin{aligned} seq_q =< & (0, \{x = 2, y = 0\}, 0), \\ & (0, \{x = 2, y = 2\}, 1), \\ & (1, \{x = 2, y = 3\}, 0) > \end{aligned}$$

Due to seq_p and 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 $P \mid\mid Q$:

$$\begin{aligned} seq_{p||q} =< & (0, \{x = 2, y = 0\}, 1), \\ & (0, \{x = 2, y = 2\}, 1), \\ & (1, \{x = 2, y = 3\}, 1) > \end{aligned}$$

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

$$\begin{aligned} seq_p =< & (0, \{x = 0, y = 2\}, 0), \\ & (0, \{x = 2, y = 2\}, 1), \\ & (1, \{x = 2, y = 3\}, 1) > \\ seq_q =< & (0, \{x = 0, y = 2\}, 1), \\ & (0, \{x = 2, y = 2\}, 0), \\ & (1, \{x = 2, y = 3\}, 0) > \end{aligned}$$

Their trace of $P||Q$ is their merge. The trace is:

$$\begin{aligned} seq_{p||q} =< & (0, \{x = 0, y = 2\}, 1), \\ & (0, \{x = 2, y = 2\}, 1), \\ & (1, \{x = 2, y = 3\}, 1) > \end{aligned}$$

Definition 3.1 (**Merge of Traces**)) As we have seen in Example 3.1, two sequences $seq1$ and $seq2$ are said to be comparable if

(1) The time sequences from the two traces are the same $\pi_1(seq1) = \pi_1(seq2)$

(2) They are built from the same sequence of states $\pi_2(seq1) = \pi_2(seq2)$

(3) None of their snapshots is made by both components $2 \notin \pi_1(seq1) + \pi_1(seq2)$

We use the following predicate to present their merge:

$$M(seq, seq_1, seq_2) =_{df}$$

$$\begin{pmatrix} (\pi_1(seq_1) = \pi_1(seq_2) = \pi_1(seq))\wedge \\ (\pi_2(seq_1) = \pi_2(seq_2) = \pi_2(seq))\wedge \\ (\pi_3(seq) = \pi_3(seq_1) + \pi_3(seq_2))\wedge \\ (2 \notin \pi_3(seq_1) + \pi_3(seq_2)) \end{pmatrix}$$

In the sequential programs, the $@(g)$ 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

$$\frac{\begin{array}{l} (p \wedge time < \infty) \wedge trig(g)\ at\ tr \ \rightarrow\ p \\ (p \wedge time < \infty)\ \wedge\ (\exists tr_1 \bullet tr \leq tr_1 \wedge \\ \qquad await(g)\ during\ [tr, tr_1)\ \wedge (trig(g)\ at\ tr_1)) \\ \qquad \rightarrow q[\pi_1(last(tr_1))/time, \pi_2(last(tr_1))/\sigma] \\ (p \wedge time < \infty) \wedge await(g)\ during\ [tr,\ tr_1)\wedge \\ \qquad \pi_2(last(tr_1)) = \infty \rightarrow q_\infty \end{array}}{tr : \{p\}\ @(g)\ \{p \vee q \vee q_\infty\}}$$

The second property of the rule notes that, in the guard statement, if the $@(g)$ is trigged by itself, the atomic action $@(g)$ 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 finial values of the program, so the postcondition is $q[\pi_1(last(tr_1))/time, \pi_2(last(tr_1))/\sigma]$.

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

$$\frac{\begin{array}{l} tr : \{p_1\}\ S_1\ \{q_1\},\ \ tr : \{p_2\}\ S_2\ \{q_2\}, \\ \forall tr_1, tr_2\ \bullet\ (M(tr_3, tr_1, tr_2) \wedge (tr_1 \rightarrow q_1) \wedge (tr_2 \rightarrow q_2)) \\ \qquad \rightarrow q[maxTime(q_1, q_2)/time, \pi_2(last(tr_3))/\sigma] \end{array}}{tr : \{p_1 \wedge p_2\}\ S_1\ ||\ S_2\ \{q\}}$$

where tr_1 is the trace of S_1 and tr_2 is the trace of S_2 when $S_1||S_2$.

The $tr \rightarrow q$ represents that the assumption q satisfies the state $\pi_2(last(tr_2))$, 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 $P :< (0, x = 1, 1), (1, x = 3, 1) >$. When we add a time delay to the end of P, the trace of P will not change anything. So we define

$$tr \rightarrow q\ =_{df}\ \pi_2(last(tr)) \wedge time < \infty \rightarrow q$$

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

$maxTime(q_1, q_2) =_{df} max(t_1, t_2)$ (t_i 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

$$\frac{tr : \{p\}\ S\ \{q\},\ tr : \{r\}\ S\ \{q\}}{tr : \{p \vee r\}\ S\ \{q\}}$$

Rule 7. Conjunction

$$\frac{tr : \{p_1\}\ S\ \{q_1\},\ tr : \{p_2\}\ S\ \{q_2\}}{tr : \{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

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

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

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

Rule 9. Consequence

$$\frac{tr : \{p\}\ S\ \{q\}, p_1 \rightarrow p, q \rightarrow q_1}{tr : \{p_1\}\ S\ \{q_1\}}$$

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

$$\frac{tr : \{p\}\ S_1\ \{r\}, tr_1 : \{r\}\ S_2\ \{q\}}{tr : \{p\}\ S_1; S_2\ \{q\}}$$

where $tr \preceq tr_1$, $tr_1 \rightarrow$ r and $\pi_1(last(tr)) + e = t_r$. ($t_r$ 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 :

$$\begin{aligned}
P ::= \ & \textbf{while}\ x > 0\ \textbf{do} \\
& \quad @(\uparrow y); \\
& \quad x := x - 1; \\
& \textbf{od;} \\
& z = 1; \\
Q ::= \ & \#1; \\
& \textbf{while}\ z \neq 1\ \textbf{do} \\
& \quad y := y + 1; \\
& \quad \#1; \\
& \textbf{od;}
\end{aligned}$$

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 $x = 2 \wedge y = 0 \wedge z \neq 1 \wedge time = 0$.
- $S_1 = (@(\uparrow y);\ x := x - 1)$ and $S_2 = (y := y + 1;\ \#1)$.

For $P||Q$, we want to prove the following correctness formulas:

$$tr : \{p_1\}\ P\ \{x \leq 0\ \wedge\ time < \infty\} \quad (4.1)$$

$$tr : \{p_1\}\ Q\ \{z = 1\ \wedge\ time < \infty\} \quad (4.2)$$

$$tr : \{p_1\}\ P||Q\ \{x \leq 0\ \wedge\ z = 1\ \wedge\ time < \infty\} \quad (4.3)$$

We denote $x \leq 0\ \wedge\ time < \infty$ as q_1, $z = 1\ \wedge\ time < \infty$ as q_2 and $x \leq 0\ \wedge\ z = 1\ \wedge\ time < \infty$ as q .

Proof :

First, as same as the Example 3.1, we get the traces of P and Q when $P||Q$.

$$\begin{aligned}
tr_p =< & (0,\ \{x = 2, y = 0, z \neq 1\},\ 1), \\
& (1,\ \{x = 0, y = 1, z \neq 1\},\ 0), && (tr_{p1}) \\
& (1,\ \{x = 1, y = 1, z \neq 1\},\ 1), && (tr_{p2}) \\
& (2,\ \{x = 1, y = 2, z \neq 1\},\ 0), && (tr_{p3}) \\
& (2,\ \{x = 0, y = 2, z \neq 1\},\ 1), && (tr_{p4}) \\
& (2,\ \{x = 0, y = 2, z = 1\},\ 1) >
\end{aligned}$$

$$\begin{aligned}
tr_q =< & (0,\ \{x = 2, y = 0, z \neq 1\},\ 1), \\
& (1,\ \{x = 0, y = 1, z \neq 1\},\ 1), && (tr_{q1}) \\
& (1,\ \{x = 1, y = 1, z \neq 1\},\ 0), && (tr_{q2}) \\
& (2,\ \{x = 1, y = 2, z \neq 1\},\ 1), && (tr_{q3}) \\
& (2,\ \{x = 0, y = 2, z \neq 1\},\ 0), && (tr_{q4}) \\
& (2,\ \{x = 0, y = 2, z = 1\},\ 0) >
\end{aligned}$$

According to the definition of $Merge$, we obtain the trace of $P||Q$.

$$\begin{aligned}
tr_{p||q} =< & (0,\ \{x = 2, y = 0, z \neq 1\},\ 1), \\
& (1,\ \{x = 0, y = 1, z \neq 1\},\ 1), \\
& (1,\ \{x = 1, y = 1, z \neq 1\},\ 1), \\
& (2,\ \{x = 1, y = 2, z \neq 1\},\ 1), \\
& (2,\ \{x = 0, y = 2, z \neq 1\},\ 1), \\
& (2,\ \{x = 0, y = 2, z = 1\},\ 1) >
\end{aligned}$$

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 :

$I_1 = z \neq 1 \wedge time < \infty$.

And we need to prove the correctness of following formula :

$$tr : \{I_1 \wedge x > 0\} \ @(\uparrow y) \ \{I \wedge x > 0\} \qquad (4.4)$$

and

$$tr_{p2} : \{I_1 \wedge x > 0\} \ @(\uparrow y) \ \{I \wedge x > 0\} \qquad (4.5)$$

Note that the implications

$$I_1 \ \wedge \ (\exists tr_{p1} \bullet tr \leq tr_{p1} \wedge \\ await(g) \ during \ [tr, tr_{p1}) \ \wedge (trig(g) \ at \ tr_{p1})) \\ \rightarrow I_1 \wedge x > 0$$

and

$$I_1 \ \wedge \ (\exists tr_{p3} \bullet tr_{p2} \leq tr_{p3} \wedge \\ await(g) \ during \ [tr_2, tr_{p3}) \ \wedge (trig(g) \ at \ tr_{p3})) \\ \rightarrow I_1 \wedge x > 0$$

hold. Thus by the rule 4 (Guard -2), we prove (4.4) and (4.5). Further by the assignment axiom 4, we prove

$$\{I \wedge x > 0\} \ x := x - 1 \ \{I \wedge x \geq 0\} \qquad (4.6)$$

By (4.4), (4.5), (4.6) and the while rule 3, we obtain

$$tr : \{I\} \ \textbf{while} \ x > 0 \ \textbf{do} \ S_1 \ \textbf{od} \ \{I \wedge x \leq 0\} \qquad (4.7)$$

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

$$\pi_2(last(tr_p)) \wedge time < \infty \rightarrow q_1$$

and

$$\pi_2(last(tr_q)) \wedge time < \infty \rightarrow q_2.$$

$tr_p \rightarrow q_1$ and $tr_q \rightarrow q_2$ are hold in this model, and using the definition of $maxtime$ and $\pi_2(tr)$, we get

$$maxTime(q_1, q_2) =_{df} time \leq \infty$$

and

$$\pi_2(last(tr_{p||q})) = \{x = 0, y = 2, z = 1\}.$$

We obtain

$$\forall tr_p, tr_q \ \bullet \ (M(tr_{p||q}, tr_p, tr_q) \wedge (tr_p \rightarrow q_1) \wedge (tr_q \rightarrow q_2)) \\ \rightarrow q[maxTime(q_1, q_2)/time, \pi_2(last(tr_3))/\sigma] \qquad (4.8)$$

Combine formula (4.1), (4.2) and (4.8) and by the Parallel rule, we can derive the correctness formula :

$$tr : \{p_1\} \ P||Q \ \{x \leq 0 \ \wedge \ z = 1 \ \wedge \ time < \infty\}$$

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.

Acknowledgment

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

References

[1] IEEE, *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Laguage,* Volume IEEE Standard 1364-1995. IEEE, 1995.

[2] IEEE, *IEEE Standard Hardware Description Language based on the Verilog Hardware Description Laguage,* Volume IEEE Standard 1364-2001. IEEE, 2001.

[3] N. Nissanke, *Realtime Systems,* Prentice Hall International Series in computer Science, 1997.

[4] Haase, V., *Real-time Behavior of Programs,* IEEE Transactions on Software Engineering SE-7,5 (Sept. 1981), 497-501.

[5] Zhu, H., *Linking the Semantics of a Multithreaded Discrete Event Simulation Language,* London South Bank University, 2005.

[6] Golze, U., *VLSI Chip Design with the Hardware Description Language Verilog: An Introduction Based on a Large RISC Processor Design,* Springer-Verlag New York, Inc. 1996.

[7] Gordon, M.J.C., *The semantic challenge of Verilog HDL,* In Proc. Tenth Annual IEEE Symposium on Logic in Computer Science, page 136-145. IEEE Computer Society Press, June 1995.

[8] Gordon, M.J.C., *Relating event and trace semantics of hardware description languages,* The Computer Journal, 45(1): 27-36, 2002.

[9] Apt, K.R., de Boer, F.S., Olderog, E., *Verification of sequential and concurrent programs,* Texts in Computer Science, Springer, 2009.

[10] Hoare, C.A.R., *Communicating Sequential Processes,* Prentice Hall International Series in Computer Science, 1985.

[11] Hooman, J., *Extending hoare logic to real-time,* Formal Aspects of Computing, 1994, 6(1):801-825.

[12] Park, S., Pfenning, F., Thrun, S., *A probabilistic language based upon sampling functions,* Acm Transactions on Programming Languages, 2004, 40.1(2004):171-182.

[13] Zhu, H., Qin, S., He, J., Bowen, J.P., *PTSC: probaility, time and shared-variable concurrency,* Innovations in Systems and Software Enginerring: A NASA Journal 5(4), 271-284(2009).

[14] Hoare, C.A.R., *Algebra of concurrent programming,* In: Meeting 52 of WG 2.3(2011).

[15] Hoare, C.A.R., He, J., *Unifying Theories of programming,* Prentice Hall International Series in Computer Science, 1998.