Asynchronous Communication in MSVL*

Dapeng Mo, Xiaobing Wang, and Zhenhua Duan**

Institute of Computing Theory and Technology,
and ISN Laboratory Xidian University,
Xi'an, 710071, P.R. China
zjmdp@foxmail.com, {xbwang,zhhduan}@mail.xidian.edu.cn

Abstract. Projection Temporal Logic (PTL) is a sound formalism for specifying and verifying properties of concurrent systems. The modeling, simulation and verification language MSVL for concurrent systems is an executable subset of PTL. However, asynchronous communication, a key component of modeling distributed system, has not been implemented in MSVL. This paper presents asynchronous communication techniques for MSVL to improve its capability for modeling and verifying distributed systems. First, a process structure is defined; then a channel structure and two pairs of communication commands are formalized; finally, an example of asynchronous communication for the contract signing protocol is demonstrated.

1 Introduction

Temporal logics [1,2,3] have been put forward as a useful tool for specifying and verifying properties of concurrent systems, and widely applied in many fields ranging from software engineering to digital circuit designs. Projection Temporal Logic(PTL)[4] is an extension of Interval Temporal Logic (ITL) and a useful formalism for system verification. The Modeling, Simulation and Verification Language (MSVL)[5] is an executable subset of PTL and it can be used to model, simulate and verify concurrent systems. To do so, a system is modeled by an MSVL program and a property of the system is specified by a Propositional Projection Temporal Logic (PPTL) formula. Thus, whether or not the system satisfies the property can be checked by means of model checking with the same logic framework.

As the complexity of distributed systems increases, a formal language for modeling and verification is desired. Although MSVL has been used to model, simulate and verify a number of concurrent systems, it could not be employed to model an asynchronous distributed system because asynchronous communication techniques have not been implemented in MSVL. For this reason, asynchronous communication construct is to be formalized.

* This research is supported by the National Program on Key Basic Research Project of China (973 Program) Grant No.2010CB328102, National Natural Science Foundation of China under Grant Nos. 60910004, 60873018, 91018010, 61003078 and 61003079, SRFDP Grant 200807010012, and ISN Lab Grant No. 201102001,Fundamental Research Funds for the Central Universities Grant No. JY10000903004.

** Corresponding author.

S. Qin and Z. Qiu (Eds.): ICFEM 2011, LNCS 6991, pp. 82–97, 2011.

Channel structure is commonly found in temporal logic languages due to its importance to describe asynchronous distributed systems. In ASDL[6], any two distinct services are impliedly connected by two unidirectional channels. It is a simple and straightforward approach to implement asynchronous communication technique. For XYZ/E[7], a channel is defined as a variable that can be a parameter of a process. This approach is flexible, but conflicts may occur when more than one process accesses a same channel at the same time. Roger Hale has implemented asynchronous communication technique for Tempura based on a shared buffer and two primitive operations[8]. The buffer is a single slot in which one message can be stored at a time. Communication in CCS[9] and CSP[10] is synchronous and there are no message buffers linking communicating agents, but asynchronous communication can be modeled by introducing buffer agents between two communicating entities. These approaches above provide us a great many ideas to implement asynchronous communication technique in MSVL.

The main contributions of this paper are as follows: 1. A process structure is defined to describe behaviors of systems. In this way, two or more processes can form a larger system with a clear structure; 2. To establish links among processes, a channel structure is presented. Channels are buffers to transport messages; 3. Communication commands, which are executed by processes to send or receive messages, are formalized. After all works above have been done, asynchronous communication is possible and a number of asynchronous concurrent systems can be modeled, simulated and verified with extended MSVL.

To inspect the practicability of our works, an example of electronic contract signing protocol is modeled and verified by the extended MSVL. Processes are used to describe all parties that participate in the protocol and channels are defined to connect all processes;then all processes run in parallel to model the protocol. With some properties specified by PPTL formulas, whether or not the protocol satisfies them are checked.

The paper is organized as follows: In section 2, the syntax and semantics of PTL are presented. In section 3, the language MSVL is briefly introduced. The formal definitions of the process structure and asynchronous communication are formalized in section 4. In section 5, an electronic contract signing protocol is modeled and verified with the extended MSVL. Conclusions are drawn in the final section.

2 Projection Temporal Logic

2.1 Syntax

Let Π be a countable set of propositions, and V be a countable set of typed static and dynamic variables. $B = \{true, false\}$ represents the boolean domain and D denotes all the data we need including integers, strings, lists etc. The terms e and formulas p are given by the following grammar:

$$
\begin{aligned}
e &::= v \mid \bigcirc e \mid \ominus e \mid f(e_1, \ldots, e_m) \\
p &::= \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_m) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \bigcirc p \mid \ominus p \mid \\
&\quad (p_1, \ldots, p_m)\, \mathit{prj}\, p
\end{aligned}
$$

$$\mathcal{I}[v] = s_k[v] = I_v^k[v]$$
$$\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases}$$
$$\mathcal{I}[\ominus e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases}$$
$$\mathcal{I}[f(e_1, \ldots, e_m)] = \begin{cases} f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \text{if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases}$$

Fig. 1. Interpretation of PTL terms

where $\pi \in \Pi$ is a proposition, and v is a dynamic variable or a static variable. In $f(e_1, \ldots, e_m)$ and $P(e_1, \ldots, e_m)$, f is a function and P is a predicate. It is assumed that the types of the terms are compatible with those of the arguments of f and P. A formula (term) is called a state formula (term) if it does not contain any temporal operators ($i.e. \bigcirc, \ominus$ and prj); otherwise it is a temporal formula (term).

2.2 Semantics

A state s is a pair of assignments (I_v, I_p) where for each variable $v \in V$ defines $s[v] = I_v[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi] = I_p[\pi]$. $I_v[v]$ is a value in D or nil (undefined), whereas $I_p[\pi] \in B$. An interval $\sigma =< s_0, s_1, \cdots >$ is a non-empty (possibly infinite) sequence of states. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of states in σ minus one. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set N_0 of non-negative integers and ω, $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0, i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. With such a notation, $\sigma_{(i..j)} (0 \leq i \preceq j \leq |\sigma|)$ denotes the sub-interval $< s_i, \ldots, s_j >$ and $\sigma(k) (0 \leq k \preceq |\sigma|)$ denotes $< s_k, ..., s_{|\sigma|} >$. The concatenation of σ with another interval (or empty string) σ' is denoted by $\sigma \cdot \sigma'$. To define the semantics of the projection operator we need an auxiliary operator for intervals. Let $\sigma =< s_0, s_1, \cdots >$ be an interval and $r_1, \ldots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \cdots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the interval (called projected interval), $\sigma \downarrow (r_1, \ldots, r_h) =< s_{t_1}, s_{t_2}, \ldots, s_{t_l} >$, where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. For example,

$$< s_0, s_1, s_2, s_3, s_4 > \downarrow (0, 0, 2, 2, 2, 3) =< s_0, s_2, s_3 >$$

An interpretation for a PTL term or formula is a tuple $I = (\sigma, i, k, j)$, where $\sigma =< s_0, s_1, \cdots >$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \preceq j \leq |\sigma|$. We use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being s_k. For every term e, the evaluation of e relative to interpretation $I = (\sigma, i, k, j)$ is defined as $I[e]$, by induction on the structure of a term, as shown in Fig.1, where v is a variable and $e_1, \ldots, e_m$ are terms.

$$
\begin{array}{llll}
empty & \stackrel{\text{def}}{=} & \neg \bigcirc true & more \stackrel{\text{def}}{=} \neg empty \\
halt(p) & \stackrel{\text{def}}{=} & \Box(empty \leftrightarrow p) & keep(p) \stackrel{\text{def}}{=} \Box(\neg empty \rightarrow p) \\
fin(p) & \stackrel{\text{def}}{=} & \Box(empty \rightarrow p) & skip \stackrel{\text{def}}{=} \neg empty \\
x\circ = e & \stackrel{\text{def}}{=} & \bigcirc x = e & x := e \stackrel{\text{def}}{=} skip \wedge x\circ = e \\
len(0) & \stackrel{\text{def}}{=} & empty & len(n) \stackrel{\text{def}}{=} \bigcirc len(n-1)(n > 0)
\end{array}
$$

Fig. 2. Some derived formulas

The satisfaction relation for formulas $\models$ is inductively defined as follows.

1. $\mathcal{I} \models \pi$ if $s_k[\pi] = I_p^k[\pi] =$ true.
2. $\mathcal{I} \models e_1 = e_2$ if $\mathcal{I}[e_1] = \mathcal{I}[e_2]$.
3. $\mathcal{I} \models P(e_1, \ldots, e_m)$ if P is a primitive predicate other than = and, for all h, $1 \leq h \leq m, \mathcal{I}[e_h] \neq nil$ and $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) =$ true.
4. $\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.
5. $\mathcal{I} \models p_1 \wedge p_2$ if $\mathcal{I} \models p_1$ and $\mathcal{I} \models p_2$.
6. $\mathcal{I} \models \exists v : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be in the values assigned to variable v at k.
7. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.
8. $\mathcal{I} \models \ominus p$ if $i < k$ and $(\sigma, i, k-1, j) \models p$.
9. $\mathcal{I} \models (p_1, \ldots, p_m)\, prj\ q$ if there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \leq j$ such that $(\sigma, i, r_0, r_1) \models p_1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $r_m < j$ and $\sigma' = \sigma\downarrow(r_0, ..., r_m) \cdot \sigma_{(r_{m+1..j)}}$
   (b) $r_m = j$ and $\sigma' = \sigma\downarrow(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.

A formula p is said to be:

1. $satisfied$ by an interval σ, denoted $\sigma \models p$, if $(\sigma, 0, 0, |\sigma|) \models p$.
2. $satisfiable$ if $\sigma \models p$ for some σ.
3. $valid$, denoted $\models p$, if $\sigma \models p$ for all σ.
4. $equivalent$ to another formula q, denoted $p \equiv q$, if $\models (p \leftrightarrow q)$.

The abbreviations $true$, $false$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $true \stackrel{\text{def}}{=} P \vee \neg P$ and $false \stackrel{\text{def}}{=} \neg P \wedge P$ for any formula P. Also some derived formulas is shown in Fig.2.

3 Modeling, Simulation and Verification Language

The Language MSVL with frame[11] technique is an executable subset of PTL and used to model, simulate and verify concurrent systems. The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$
\begin{array}{l}
e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0\, op\, e_1 (op ::= + \mid - \mid * \mid / \mid mod) \\
b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1
\end{array}
$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows:

Assignment:	$x = e$
P-I-Assignment:	$x \Leftarrow e$
Conditional:	if b then p else $q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
While:	while b do $p \stackrel{\text{def}}{=} (b \wedge p)^* \wedge \Box(empty \rightarrow \neg b)$
Conjunction:	$p \wedge q$
Selection:	$p \vee q$
Next:	$\bigcirc p$
Always:	$\Box p$
Termination:	$empty$
Sequential:	$p; q$
Local variable:	$\exists x : p$
State Frame:	$lbf(x)$
Interval Frame:	$frame(x)$
Parallel:	$p \parallel q \stackrel{\text{def}}{=} p \wedge (q; true) \vee q \wedge (p; true)$
Projection:	$(p_1, \ldots, p_m)$ *prj* q
Await:	$await(b) \stackrel{\text{def}}{=} (frame(x_1) \wedge \cdots \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$ where $x_i \in V_b = \{x \mid x$ appears in $b\}$

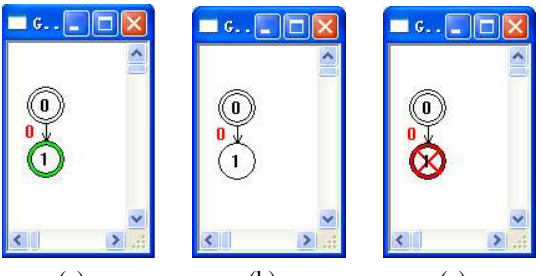
where x denotes a variable, e stands for an arbitrary arithmetic expression, b a boolean expression, and $p_1, \ldots, p_m$, p and q stand for programs of MSVL. The assignment $x = e$, $x \Leftarrow e$, $empty$, $lbf(x)$, and $frame(x)$ can be regarded as basic statements and the others composite ones.

The assignment $x = e$ means that the value of variable x is equal to the value of expression e. Positive immediate assignment $x \Leftarrow e$ indicates that the value of x is equal to the value of e and the assignment flag for variable x, p_x, is true. Statements of $if\ b\ then\ p\ else\ q$ and $while\ b\ do\ p$ are the same as that in the conventional imperative languages. $p \wedge q$ means that p and q are executed concurrently and share all the variables during the mutual execution. $p \vee q$ means p or q are executed. The next statement $\bigcirc p$ means that p holds at the next state while $\Box p$ means that p holds at all the states over the whole interval from now. $empty$ is the termination statement meaning that the current state is the final state of the interval over which the program is executed. The sequence statement $p; q$ means that p is executed from the current state to its termination while q will hold at the final state of p and be executed from that state. The existential quantification $\exists x : p$ intends to hide the variable x within the process p. $lbx(x)$ means the value of x in the current state equals to value of x in the previous state if no assignment to x occurs, while $frame(x)$ indicates that the value of variable x always keeps its old value over an interval if no assignment to x is encountered. Different from the conjunction statement, the parallel statement allows both the processes to specify their own intervals. e.g., $len(2) \| len(3)$ holds but $len(2) \wedge len(3)$ is obviously false. Projection can be thought of as a special parallel computation which is executed on different time scales. The projection $(p_1, \ldots, p_m)$ *prj* q means that q is executed in parallel with $p_1, \ldots, p_m$ over an interval obtained by taking the endpoints

of the intervals over which the p_i's are executed. In particular, the sequence of p_i's and q may terminate at different time points. Finally, $await(b)$ does not change any variable, but waits until the condition b becomes true, at which point it terminates.

An MSVL interpreter has been implemented in Microsoft Visual C++. An MSVL program can be transformed to a logically equivalent conjunction of the two formulaes, $Present$ and $Remains$. $Present$ consists of immediate assignments to program variables, output of program variables, $true$, $false$ or $empty$. It is executed at the current state. The formula $Remains$ is what is executed in the subsequent state (if any). The interpreter accepts well-formed MSVL programs as its input and interprets them in a serial states. If a program is reduced to $true$, it is satisfiable and a model is found, otherwise it has no model.

The interpreter can work in three modes: modeling, simulation and verification. In the modeling mode, given the MSVL program p of a system, all execution paths of the system are given as an Normal Form Graph (NFG)[5] of p. A correct path ends with a bicyclic node as shown in Fig.3(a). Under the simulation mode, an execution path of the system is output according to minimal model semantics[12] of MSVL. With the verification mode, given a system model described by an MSVL program, and a property specified by a PPTL formula, it can automatically be verified whether or not the system satisfies the property, and the counterexample will be pointed out if the system does not satisfy it. A satisfiable path ends with a circular node as shown in Fig.3(b) while an unsatisfiable path ends with a terminative node as shown in Fig.3(c).



(a) (b) (c)

Fig. 3. Three types of nodes

4 Asynchronous Communication

4.1 Process

In[10], a process stands for the behavior pattern of an object. A service can be viewed as a computational entity in[6]. Similarly, we use a process to describe the manner of an object. Furthermore, process is a reasonable structure when several MSVL statements run in parallel and each independently determines the interval length. The introduction of the process structure simplifies and modularizes programming so that a complicated system can be separated into several processes. Besides, processes are viewed as communication entities and make the implementation of asynchronous communication more feasible.

The process structure consists of two parts: declaration part and calling part. The declaration part has three components: process name, formal parameters and process body. The calling part consists of process name and actual parameters.

Let $ProcName$ be the name of a process, and P, P_1, P_2 be MSVL statements. x, y are variables, and c, d denotes channel variables. The formal definitions of process are shown below:

$$\begin{aligned} proc\ ProcName(x) &\stackrel{\text{def}}{=} \{Pro\_Body\} \\ Pro\_Body &\stackrel{\text{def}}{=} x = e \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \bigcirc P \mid \Box P \mid \exists x : P \mid P_1 \,||\, P_2 \mid \\ &\quad if\ b\ then\ P_1\ else\ P_2 \mid P_1; P_2 \mid while\ b\ do\ P \mid frame(x) \mid \\ &\quad send(c, e) \mid receive(d, y) \mid empty \\ ProcName(y) &\stackrel{\text{def}}{=} (y/x)Pro\_Body_{ProcName} \end{aligned}$$

Where $proc$ is a key word, and $Proc\_Body$ is the main body of a process. The statement $ProcName(y)$ refers to call the process $ProcName$ with the actual parameter y. The semantic of $ProcName(y)$ is to replace the value of all x's in the main body of $ProcName$ by the value of y. The statement $send(c, e)$ represents to send a message(value of expression e) to channel c while the statement $receive(d, y)$ means to receive a message from channel d and assign it to the variable y.

4.2 Channel

Channel communication can be synchronous or asynchronous. For synchronous communication, a receiver blocks until a compatible party is ready to send. As to asynchronous communication, a communicating party can start a sending or receiving activity at any time without consideration of the state of the other party, because there is a buffer between them.

Before presenting formal definitions, we firstly give informal descriptions of channel communication. In MSVL, a channel is a bounded First-In-First-Out (FIFO) list where a message can be inserted at one end and received sequentially at the other. Sending a message equals appending it to the tail of the channel; receiving a message is to remove the head of the channel. Only when there is at least one empty place available in the channel will a sending activity be successful, otherwise waiting for an empty place or terminating the sending activity may be selected. A similar procedure applies to a receiving activity. As formal parameters in the declaration of a process can be channel variables, we can transfer a defined channel variable as the actual parameter to the formal parameter when calling a process. Then the process can access the channel to transport messages.

A channel is regarded as a bounded FIFO list and its declaration is given below:

$$chn\ c(n) \stackrel{\text{def}}{=} c =<> \wedge max_c = n$$

where chn is a key word and $chn\ c(n)$ declares channel c with a capacity of n. Here c is an empty list, and max_c is a static variable that represents the capacity of list c. Some list operators make it behave like a bounded FIFO.

Any process can access a channel if the channel is visible in its scope. Hence, the number of processes that a channel can connect is not restricted. Obviously, conflicts

may happen when more than one process accesses a same channel at the same time and therefore some exclusion algorithms are necessary. Unfortunately, the algorithms based on hardware instructions are not workable since atomic operations are incapable of being expressed in MSVL, and the algorithms related to software are so complicated that they will make MSVL programs in confusion and barely intelligible. According to our experience, attaching exactly one process to each end of a channel will be a wise choice.

4.3 Communication Commands

For simplicity, we firstly introduce two predicates as follows:

$$\begin{array}{ll} isfull(c) & \stackrel{\text{def}}{=} |c| = max_c \\ isempty(c) & \stackrel{\text{def}}{=} |c| = 0 \end{array}$$

- $isfull(c)$ evaluates to $true$ if channel c is full, otherwise $false$.
- $isempty(c)$ evaluates to $true$ if channel c is empty, otherwise $false$.

Let x be an output expression, and y be an input variable, and c be a channel variable. Communication commands are defined as follows:

$$\begin{array}{ll} send(c,x) & \stackrel{\text{def}}{=} await(!isfull(c)); c := c \cdot < x > \\ receive(c,y) & \stackrel{\text{def}}{=} await(!isempty(c)); y := head(c) \wedge c := tail(c) \end{array}$$

- The command $send(c, x)$ will block until c has at least one empty place. If c is not full at current state, x can be inserted into the tail of c at the next state, otherwise $await(!isfull(c))$ statement will be executed repeatedly at the next state in accordance with the semantic of $await$ structure.
- If c is not empty at current state, the message at the head of c will be removed and assigned to the variable y at the next state, otherwise $await(!isempty(c))$ statement will be executed at the next state.
- The length of intervals of the two commands is 1 at least if the predicates $isfull$ and $isempty$ are $false$ at the initial state. The length, however, may be infinite if the predicates are always $true$.

An example is demonstrated to illustrate the use of $send$ and $receive$.

Example 1. A and B are two processes, and variable c is a channel between them. The pointer symbols * and & are defined in[13]. The MSVL program is given in Fig.4.

- state s_0: A gets ready to append x to the tail of c at the next state. B will execute $await(!isempty(c))$ statement again at the next state, since there is no message in c at the current state.
- state s_1: A puts x at the tail of c and then terminates. B prepares to get x at the next state since x is at the head of c at the current state.
- state s_2: B removes x from c and assigns it to the variable y. Then B terminates.

$$
\begin{aligned}
proc\ P(ch) = \{&exists\, x : \{ \\
&frame(x)\, and\, x = 1\, and\, send(*ch, x)\} \\
&\}; \\
proc\ Q(ch) = \{&exists\, y : \{ \\
&frame(y)\, and\, y = 0\, and\, receive(*ch, y)\} \\
&\};
\end{aligned}
$$

$$frame(c)\, and\, chn\, c(1)\, and\, (P(\&c)||Q(\&c))$$

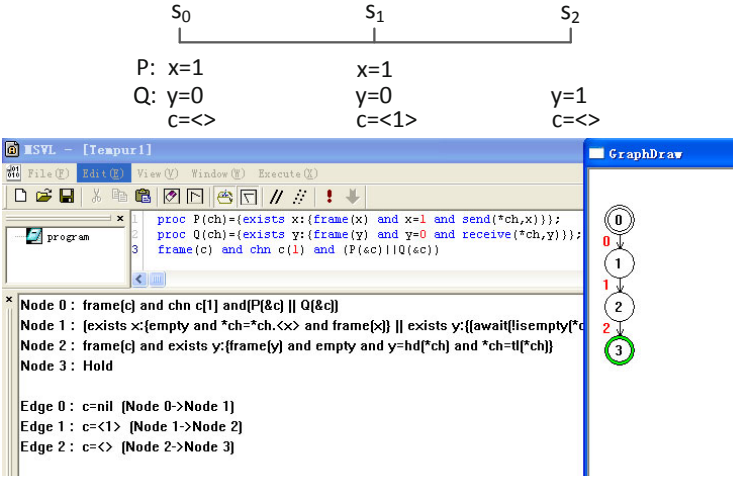


Fig. 4. Example of send and receive

While modeling a distributed system with timing constraints, some party may have to time out, which happens frequently in communication, to give up waiting if its request is not responded for a long time. The commands $send$ and $receive$ do not terminate until the predicates $isfull$ and $isempty$ become $false$, which implies they are not capable of handling timeout mechanism in these systems. Hence, another pair of communication commands is provided:

$$
\begin{aligned}
put(c, x) &\stackrel{\text{def}}{=} if(\,!isfull(c)\,)\ then\ \{\, c := c\cdot < x > \}\\
&\quad else\{skip\} \\
get(c, y) &\stackrel{\text{def}}{=} if(\,!isempty(c)\,)\ then\ \{\, y := head(c) \wedge c := tail(c)\,\} \\
&\quad else\{\, skip\,\}
\end{aligned}
$$

We replace the $await$ structure by $if-else$ structure. If the predicate $isfull$ or $isempty$ is $true$, $skip$ is executed. This pair of commands enable us to deal with timeouts in modeling the systems with timing constraints while the commands $send$ and $receive$ are convenient to describe the other systems. An appropriate selection should be made according to the features of the system.

5 An Application

5.1 An Example of Electronic Contract Signing Protocol

The crux of a commercial transaction is usually an exchange of one item for another. More specifically, electronic contract signing can be considered as a fair exchange of digital signatures to the same contract.

An electronic contract signing protocol allows n parties to sign a contract over networks. As the protocol relates to all parties' benefits, some critical properties need to be ensured, e.g., fairness[14]. Fairness denotes that either all parties obtain a signed contract, or nobody does. A trust third party(TTP) is necessary to guarantee the fairness, which is proved by Pagnina and Gartner in 1999[15].

The most straightforward contract signing protocol uses a TTP that first collects all signatures and then distributes the decision signed or failed. But as the third party has to be involved in all protocol executions it might easily become a reliability and performance bottleneck. To avoid such a bottleneck, optimistic protocols which do not involve a TTP in the normal, exception-less case but only involve it in the presence of faults or in the case of dishonest parties who do not follow the protocol are researched.

The optimistic multi-party contract signing protocols can run on synchronous or asynchronous networks. Basically, "synchronous"[14] means that all parties have synchronized real-time clocks, and that there is a known upper bound on the network delays. The most widely synchronous protocol is described in [16], which is to be modeled and verified with extended MSVL below. "Asynchronous"[14] means that there are no assumptions on clock synchronization and network delays. This means more precisely that a communication allows parties to respond at arbitrary times or infinite network delay. The first asynchronous optimistic multi-party contract signing protocol is described in[14]. Nevertheless, the protocols for asynchronous networks are more expensive. An improved version presented in[17] requires 2 rounds with the premise that the number of dishonest parties is less than half parties. Unfortunately, it cannot be predicted.

Before we present the protocol, some assumptions are listed as follows:

1. There is an active-time limit t, after which all parties are guaranteed that the state of the transaction is not changed. Requests for exceptions must be made before an earlier deadline. Hence, all parties have to synchronize the clocks in order to agree on the active-time limit as well as to compute local timeouts within rounds. In our model, we assume clocks of all parties are synchronized and each party may decide independently when to time out, and each step runs within a reasonable time limit.
2. The channels between the TTP and all other parties are reliable according to the conclusion that Pagnina and Gartner drew in 1999, whereas other channels may be unreliable. Namely, messages are delivered eventually between TTP and any other party, but the reliability of message passing cannot be guaranteed in other cases.
3. As already mentioned, TTP is involved in case of exceptions. Exceptions in the protocol mainly develop in two forms: receiving invalid signatures and losing

messages, which respectively are caused by dishonest parties and unreliable networks. For the simplicity of modeling, both of the two forms are regarded as some party's not sending message to others. Therefore once a message is received, it always represents a valid signature signed by the sender.

The protocol consists of main protocol and recovery protocol. If all parties are honest and no message is lost, the recovery protocol will not be involved. The details are described as follows[16]:

The Main Protocol

- *The First Round*
  - P_i sends $m_{[1,i]} = sign_i(1, c)$ to other parties
  - From all message of type $m_{[1,j]}$, P_i tries to compile vector $M_1 = (m_{[1,1]}, \ldots, m_{[1,n]})$. If this succeeds and each $m_{[1,j]}$ is a valid signature, then P_i enters the second round, otherwise P_i waits for a message from TTP.
- *The Second Round*
  - P_i sends $m_{[2,i]} = sign_i(2, c)$ to other parties
  - From all message of type $m_{[2,j]}$, P_i tries to compile vector $M_2 = (m_{[2,1]}, \ldots, m_{[2,n]})$. If this succeeds and each $m_{[2,j]}$ is a valid signature, then P_i decides signed and stops, otherwise P_i sends $m_{[3,i]}$=$sign_i(3, M_1)$ to TTP and waits for reply.

The Recovery Protocol

- TTP: If TTP receives at least one message $m_{[3,i]}$ which contains a full and consistent M_1, then TTP sends $M_{ttp} = sign_{ttp}(M_1)$ to all parties, and each P_i receiving this decides signed, otherwise TTP does not send anything, and each P_i waiting for a message from TTP decides failed if none arrives, or signed in case M_{ttp} is received, and stops.

Some explanations of the protocol are listed as follows:

1. The vector M_2 and M_{ttp} are equivalent, and they both refer to a valid contract. Assume an honest party get a valid contract. If this happens because of M_{ttp}, then TTP has distributed it to all parties, and all honest parties decide signed. Now assume an honest party V accepts because of M_2. As M_2 contains all P_i's signature $m_{[2,i]}$, P_i successfully complied M_1 in round 1. If P_i received M_2 in Round 2 it decides signed. Otherwise it initiates an recovery, which is necessarily answered by M_{ttp}, and P_i decides signed.
2. In a synchronous network, each P_i waiting for a message from TTP can correctly decide failed when times out, however, it would not be effective in a asynchronous network, as a party could not decide whether a message was not sent, or just not delivered yet.

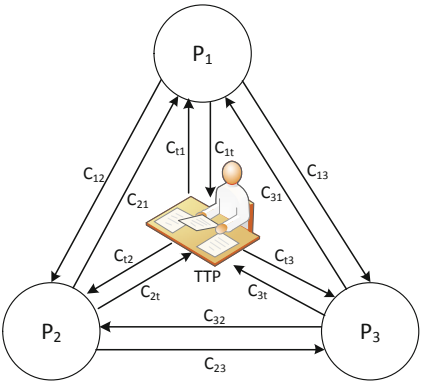


Fig. 5. Protocol structure for three parties

5.2 Modeling, Simulation and Verification with MSVL

Especially, we assume that three parties plan to sign a contract over a synchronous network by executing the protocol. TTP is included to deal with exceptions. We focus largely on the procedure of the protocol, and all messages are simplified as strings. The MSVL code of the protocol and the executable file of the interpreter can be downloaded by visiting `http://ictt.xidian.edu.cn/example.zip`.

An analysis of possible execution paths is made according to the number of parties who fail to send messages in the first round. Not sending messages is caused by two reasons mentioned above.

- Situation 1: All parties send messages to others in the first round. There are $2^3 = 8$ cases in all, according to whether the three parties send messages or not in the second round. In any case, all parties can gain a signed contract eventually.
- Situation 2: Two of them send messages but the third one fails to send in the first round. Then the third one sends a recovery request to TTP in the second round, therefore all parties will get a signed contract broadcasted by TTP. There are C_3^2x2=6 cases in all.
- Situation 3: Only one party sends messages in the first round, nobody can successfully compile M_1 to enter the second round, then all parties will time out in waiting for TTP's broadcast. Hence, there are C_3^2=3 cases in all.
- Situation 4: All parties fail to send messages in the first round. Nobody can enter the second round and there is 1 case in all.

There are 18 cases in all according to the analysis above. We run the program with extended MSVL interpreter under modeling mode and all 18 execution paths are shown in Fig.6. Due to the fact that some paths are too long to completely show in the figure, we use suspension points to represent part of them. In the modeling mode, the bicyclic nodes merely represent a successful modeling procedure since some nodes stand for a successful signing and the others represent a failed signing.

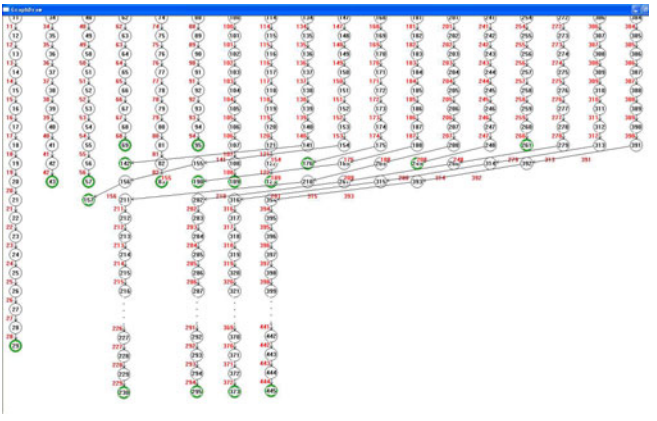


Fig. 6. Modeling result of the protocol

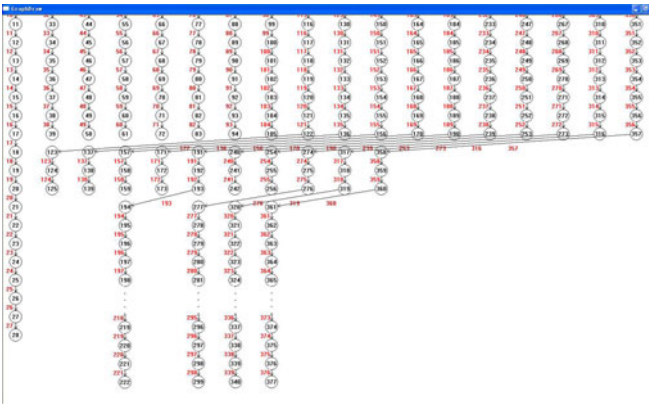


Fig. 7. Verification result of property(1)

Before verifying the properties fairness and optimism, we need to specify them by PPTL formulas. Then all works of the verification can be done automatically by means of model checking with the MSVL interpreter.

– A fairness property

$$
\begin{aligned}
&define\ l : cont_1 \ = \text{"}nil\text{"};\\
&define\ m : cont_2 = \text{"}nil\text{"};\\
&define\ n : cont_3 \ = \text{"}nil\text{"};\\
&define\ p : cont_1 \ = \text{"}signed\text{"};\\
&define\ q : cont_2 \ = \text{"}signed\text{"};\\
&define\ r : cont_3 \ = \text{"}signed\text{"};
\end{aligned}
$$

$$
fin((p\ and\ q\ and\ r)\ or\ (l\ and\ m\ and\ n)) \tag{1}
$$

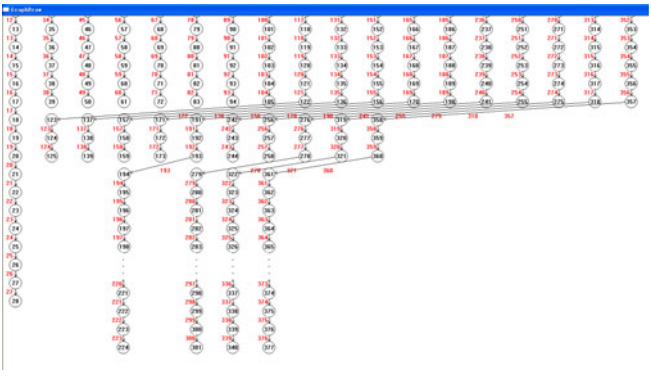


Fig. 8. Verification result of property(2)

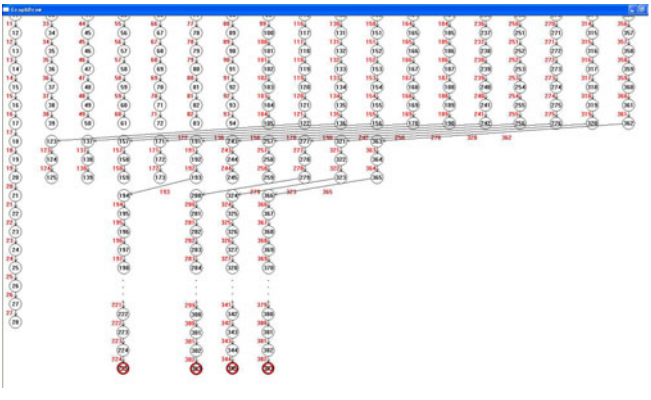


Fig. 9. Verification result of an unsatisfied PPLT formula

The proposition $cont_i =$ "$signed$" means P_i has got a valid contract while $cont_i =$ "nil" implies P_i has failed to get a valid contract. Therefore the property (1) implies either all parties obtain a signed contract, or nobody does at final states. All 18 paths ended with circular nodes in Fig.7 show that the protocol satisfies the property (1).

– An optimism property

$$
\begin{aligned}
&define\ d : opt_1 = 1;\\
&define\ e : opt_2 = 1;\\
&define\ f : opt_3 = 1;\\
&define\ g : opt\ \ = 1;
\end{aligned}
$$

$$fin((d\ and\ e\ and\ f) \rightarrow g) \tag{2}$$

The proposition $opt_i = 1$ means P_i has compiled vector M_2 successfully. The proposition $opt = 1$ indicates TTP is not involved. Therefore the property (2) implies TTP will not participate if all parties can compile vector M_2 at final states. Fig.8 indicates the protocol satisfies the property (2).

– An unsatisfiable PPTL formula

$$fin(p\ and\ q\ and\ r) \tag{3}$$

The formula (3) means all parties will obtain a valid contract at final states. Apparently, the formula is unsatisfiable in accordance with the analysis above. Some cases will lead to a situation that nobody gets a valid contract. Fig.9 shows the protocol does not satisfy the formula (3).

6 Conclusion

In this paper, we have discussed the implementation of asynchronous communication technique in MSVL. The formal definitions of process structure, channel structure and communication commands are presented. This enables us to model and verify concurrent systems with asynchronous communications. In addition, an example of optimistic multi-party contract signing protocol has been employed to show how our method works. Its fairness and optimism have been proved satisfiable with extended MSVL. In contrast, an unsatisfiable property has also been checked and all counterexamples have been pointed out. In the future, we will further investigate the operational and axiomatic semantics of MSVL with asynchronous communication. In addition, we will also try to model and verify some larger example to our approach.

Acknowledgment. We would like to thank Miss Qian Ma and Miss Xia Guo for their useful help. In particular, Guo's help on MSVL interpreter and Ma's suggestion on the verification example are very appreciated.

References

1. Pnueli, A.: The temporal semantics of concurrent programs. In: Proceedings of the 18th IEEE Symposium Foundations of Computer Science, pp. 46–67 (1997)
2. Karp, Alan, R.: Proving failure-free properties of concurrent systems using temporal logic. ACM Trans. Program. Lang. Syst. 6, 239–253 (1984)
3. Cau, A., Moszkowski, B., Zedan, H.: Itl and tempura home page on the web, `http://www.cse.dmu.ac.uk/STRL/ITL/`
4. Tian, C., Duan, Z.: Propositional projection temporal logic, buchi automata and ω-regular expressions. In: Agrawal, M., Du, D.-Z., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
5. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
6. Solanki, M., Cau, A., Zedan, H.: Asdl: A wide spectrum language for designing web services. In: WWW, pp. 687–696 (2006)
7. Tang, Z.: Temporal Logic Program Designing and Engineering, vol. 1. Sicence Press, Beijing (1999)
8. Hale, R.: Programming in Temporal Logic. Cambridge University, Cambridge (1988)
9. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980)

10. Hoare, C.A.R.: Communicating sequential processes (August 1978)
11. Duan, Z., Koutny, M.: A framed temporal logic programming language. Journal Computer Science and Technology 19(3), 341–351 (2004)
12. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Science of Computer Programming 70, 31–61 (2008)
13. Duan, Z., Wang, X.: Implementing pointer in temporal logic programming languages. In: Proceedings of Brazilian Symposium on Formal Methods, Natal, Brazil, pp. 171–184 (2006)
14. Baum-waidner, B., Waidner, M.: Optimistic asynchronous multi-party contract signing (1998)
15. Pagnia, H., Gartner, F.C.: On the impossibility of fair exchange without a trusted third party. Darmstadt University of Technology, Tech. Rep. Technical Report: TUD-BS-1999-02 (1999)
16. Asokan, N., Baum-waidner, B., Schunter, M., Waidner, M.: Optimistic synchronous multi-party contract signing (1998)
17. Baum-Waidner.: Optimistic asynchronous multi-party contract signing with reduced number of rounds (2001)