Model-checking Alternating-time μ-calculus (AMC) Implementation Guide with REDLIB[⋆]

Farn Wang[1,2]

1: Dept. of Electrical Engineering, National Taiwan University
2: Graduate Institute of Electronic Engineering, National Taiwan University

Abstract. Basic materials for Alternating-time μ-calculus (AMC).

Keywords: games, turn-based, AMC, logic, model-checking, expressiveness

1 Introduction

In the verification of distributed real-time systems, the appropriate abstraction of the system behaviors is crucial to the balance between the precision of the models and the efficiency of verification. For instance, if we model how a missile is directed to hit a jet-fighter at the granularity of sub-atomic particle interaction, then of course we have an extremely precise model. However, such cumbersome models can also involve too many details irrelevant to the verification of the system and likely incur infeasible and unnecessary computing resource requirement. One commonly used abstraction technique is to model several events as a simultaneous happening. For example, in figure 1, we have a system of an *SAM (surface-to-air missile)* and multiple hostile jet-fighters. There are two events: the '`hit`' of the missile on a jet-fighter and the observations of the '`explosion`' of the hapless jet-fighter by other jet-fighters. On one hand, the '`hit`' event is an interaction between the missile and the hapless jet-fighter. On the other hand, right after the '`hit`' event, the '`explosion`' event is broadcast to all the remaining enemy jet-fighters and may affect their actions afterward. In most verification tasks, what happen in the split-second between the '`hit`' event and the '`explosion`' event does not matter. To the missile launcher, what matters is when it can start tracking the next target. To the remaining jet-fighters, what matters is their reaction after the observation. Thus, it is only natural to model the '`hit`' event and the '`explosion`' event as a simultaneous happening. Modeling them as two separate events only unnecessarily adds to the verification complexity and does not help engineers in analyzing the behavior of the system.

One language device designed to model simultaneous actions in two processes is the *channel* (i.e., *event* in this work) concept for binary synchronization [**?**]. Conceptually, the device glues two local transitions[1] from two different processes to model a *global transition*.[2] Such a device

---

[⋆] The work is partially supported by NSC (Grant NSC 97-2221-E-002-129-MY3, Taiwan, ROC) and by Research Center for Information Technology Innovation, Academia Sinica, Taiwan, ROC. The authors also would like to thank Prof. Moshe Vardi for his valuable suggestions.

[1] A *local transition* models the observation of a global state-change from a process in a concurrent system.

[2] A *global transition* models a global state-change and could be the simultaneous interaction of several local transitions.

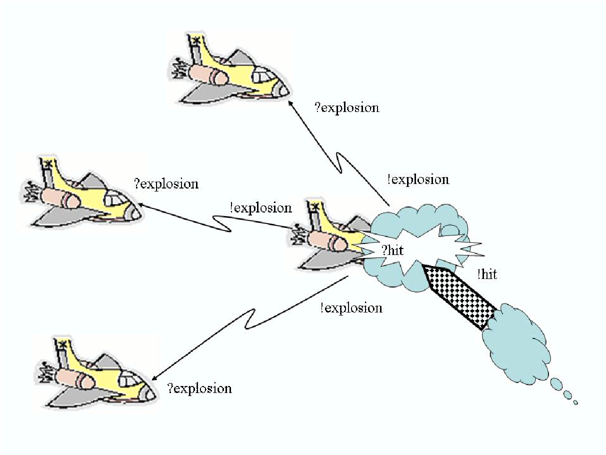


Fig. 1. A system with one missile and 4 jet-fighters

can greatly help to improve the modularity of model descriptions. In the above-mentioned SAM example, the action that a missile hits a jet-fighter can be modeled with an event named `hit` between the missile and the hapless jet-fighter. The language device `!hit` represents the sending (or output) *synchronization operation (sync-op)* by the missile through the event while `?hit` represents the receiving (or input) sync-op by the hapless jet-fighter through the same event. Two local transitions labeled respectively with the input sync-op and output sync-op through the same event must happen at the same instant to make a global transition. Modeling such a global transition as two synchronized local transitions can greatly enhance the modularity in model construction.

We use an extension of binary synchronizations for flexibility in model construction. In our extension, we allows for sync-ops like $!\sigma@q$ where q is a place holder for the identifier of the process that responds to this sync-op. The place-holders can then be used for the expressive description of the corresponding local transition.

Example 1. local transition rules with quantified place-holders When the SAM hits a jet-fighter, the SAM may want to make sure that the fighter is the target and still in the range of the missile. Suppose t is a variable that records the identity of the target. We may have the transition diagrams in figure 2 for this example synchronization. The ovals represent operation modes of the processes. The arcs represent local transition rules. The SAM moves from the `tracking` mode to the `idle` mode while the hapless jet-fighter from the `in-range` mode to the `exploded` mode. On the arcs, we label sync-ops, triggering conditions, and actions. Place-holder q represents the identifier of the process responding to sync-op `!hit`. The hitting local transition rule of the SAM is labeled with sync-op “`!hit`$@q$" and triggering condition “$q = t \wedge \texttt{in-range}[q]$." ■

Although the above-mentioned devices are good for binary synchronizations between two parties, we can use them to construct complex actions, e.g., broadcasting events, out of many simultaneous process transitions. For example, in figure 1, the event that the missile hits one

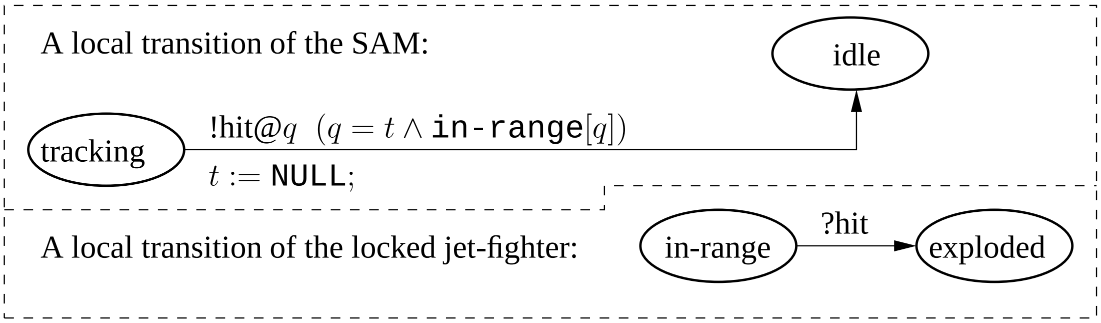
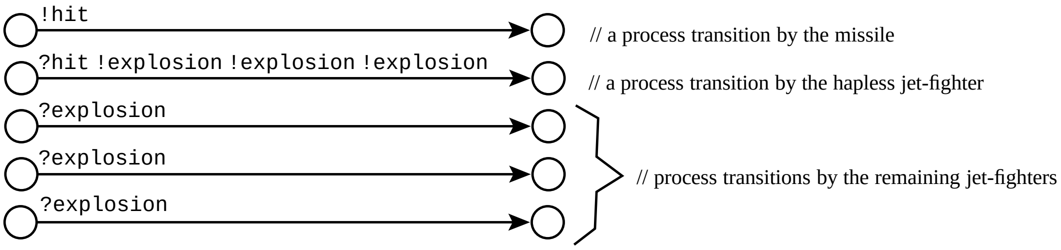

A local transition of the SAM:
idle
tracking
!hit@q $(q = t \wedge \texttt{in-range}[q])$
$t := \texttt{NULL}$;
A local transition of the locked jet-fighter:
in-range
?hit
exploded

Fig. 2. Synchronization of the locking event

jet-fighter can cause an explosion observed by the other three jet-fighters. By constructing the following 5 local transition rules,


!hit
// a process transition by the missile
?hit !explosion !explosion !explosion
// a process transition by the hapless jet-fighter
?explosion
?explosion
?explosion
// process transitions by the remaining jet-fighters

we can glue the five local transition rules to make a global transition that models the simultaneity of the `hit` event and the observations of the `explosion` event. Such a scheme allows for concise and highly abstract models in which the interaction in the split second between the hitting and observation of the explosion is of no concern to the verification engineers. In addition, the scheme also allows for the modular description of each party involved in a complex synchronization. A jet-fighter can be modeled without the knowledge of how many peers are involved in the synchronization.

In this work, we discuss how to do AMC model-checking for concurrent game graphs constructed from process automatas that communicate with binary synchronization described in the above.

2 Concurrent game graphs

Definition 1. Concurrent game graphs A *concurrent game* is played by many agents. Assume that the number of agents is m and we index the agents with integers 1 through m. It is formally presented as a tuple $G = \langle m, Q, q_0, P, \lambda, R, \Sigma, \delta \rangle$ with the following restrictions. m is the number of agents in the game. Q is a finite set of states. $q_0 \in Q$ is the *initial state* of G. P is a finite set of atomic propositions. $\lambda : Q \mapsto 2^P$ is a proposition labeling function. For convenience, we assume that the $\lambda()$ labeling of a state uniquely identifies the state. $R \subseteq Q \times Q$ is the set of transitions. Σ is a finite alphabet. $\delta : R \mapsto \Sigma^m$ is a labeling function that labels each transition with a vector in Σ^m representing the choices of events at the transition by the agents. ■

Example 2. : In figure 3, we have the graphical representation of a concurrent game graph of three players. The ovals represent states while the arcs represent state transitions. The δ vectors of events of transitions are labeled inside the states. We also put down the λ values inside the corresponding states. ■

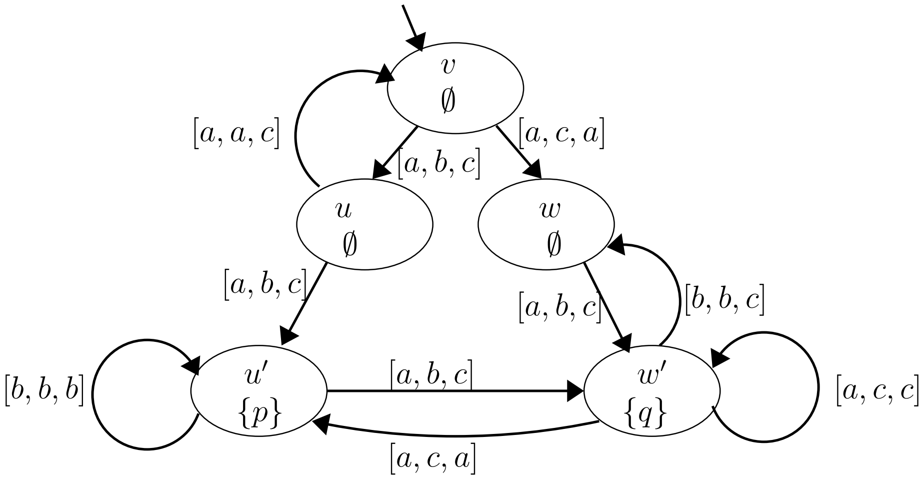


Fig. 3. A concurrent game graph for three agents

A state predicate of P is a Boolean combination of elements in P. We let *SP*(P) be the set of state predicates of P. The satisfaction of a state predicate η at a state q, in symbols $q \models \eta$, is defined in a standard way.

For convenience, given a game graph $G = \langle m, Q, q_0, P, \lambda, R, \Sigma, \delta \rangle$, we denote $m, Q, q_0, P, \lambda, R, \Sigma$, and δ respectively as $m_G, Q_G, q_{0A}, P_G, \lambda_G, R_G, \Sigma_G$, and δ_G.

Definition 2. **Plays and play prefixes** A *play* is an infinite path in a game graph. A play is *initial* if it begins with the initial state. Given a play $\pi = \bar{q}_0\bar{q}_1 \ldots$, for every $k \geq 0$, we let $\pi(k) = \bar{q}_k$. Also, given $h \leq k$, we let $\pi[h, k]$ denote $\pi(h) \ldots \pi(k)$ and $\pi[h, \infty)$ denote the infinite tail of π from $\pi(h)$. A *play prefix* is a finite segment of a play from the beginning of the play. For convenience, we let *pfx*(G) be the set of initial play prefixes of G and *pfx*$^\omega(G)$ be the set of initial plays of G. ■

Given a play prefix $\rho = \bar{q}_0\bar{q}_1 \ldots \bar{q}_n$, we let $|\rho| = n + 1$. For convenience, we use *last*(ρ) to denote the last state in π, i.e., $\rho(|\pi| - 1)$.

Given a sequence A of agent indices, we let $\{A\}$ be the set of agent indices in A. A set of agent indices is also called an *agency*.

Definition 3. Strategies For an agent a in an agency $\{A\} \subseteq [1,m]$, a (*memoryless*) *strategy binding* σ for agents in A is a function from $Q_G \times \{A\}$ to Σ_G such that for every $q \in Q_G$ and $a \in \{A\}$, $\sigma(q,a) \in \Sigma_G$.

We let $str^{\{A\}}(G)$ be the set of strategy binding of G for agency $\{A\}$. ■

A play π is compatible with a strategy σ of an agent $a \in [1,m]$ iff for every $k \in [0,\infty)$, $\omega(\pi(k)) = a$ implies $\sigma(\pi[0..k]) = \pi(k+1)$. The play is compatible with a S-binding Σ of agency $\{A\}$ iff for every $a \in \{A\}$, the play is compatible with $\Sigma(a)$ of a.

3 How to construct concurrent game graphs from process automatas

Given a system of m processes, we use integers $1,\ldots,m$ to respectively identify the m processes. We also use `NULL` to denote zero. Each process in our system models may have a set of local variables. For $1 \leq p \leq m$, to access a local variable named 'y' of process p, we may simply write '$y@(p)$.' While in the scope of execution of process p, we may use the *default shorthand* 'y' for '$y@(p)$.' If the process identifier of a peer process is stored in local variable q, then we can also use '$y@(q)$' to access the local variable y of the peer process.

Formally speaking, a sync-op of event set Σ and process identifier place-holder set Q can be viewed as a triple $\langle d,\sigma,q\rangle$ with the following restrictions.

- d is the direction of the synchronization. $d =$'!' means it is an output sync-op. $d =$'?' means it is an input sync-op.
- σ is an event name in Σ.
- $q \in Q$ is a place-holder for the identifier of the process that responds to this sync-op. A place-holder is quantified over the execution scope of one local transition rule. Thus the same place-holder name can be used in many local transition rules without ambiguity in the interpretation of the values of the place-holders.

For convenience, we shall write $d\sigma@q$ in place of $\langle d,\sigma,q\rangle$ from now on. When the place-holder of the identifier of the responding process is not used, we may write $d\sigma$ (and $\langle d,\sigma,\rangle$) for simplicity.

Example 3. Synchronizations with quantified place-holders Suppose that we have processes 1 and 2 synchronizing with sync-ops $\langle ?,\texttt{hit},q\rangle$ and $\langle !,\texttt{hit},q\rangle$ respectively. In the scope of the synchronization, the former q and the latter q are interpreted respectively as 2 and 1. ■

We let *SOSEQ*(Σ,Q) be the set of all finite sequences of sync-ops of Σ and Q. A variable q is declared in a sequence $[s_1 s_2 \ldots s_n] \in$ *SOSEQ*(Σ,Q) if for some $1 \leq i \leq n$, $s_i = \langle d,\sigma,q\rangle$ for some d and $\sigma \in \Sigma$.

Given a set L of local discrete variable namesand a set Q of process identifier place-holder names, we use *VAR*(L,Q) for the following variable reference name set $L \cup \{y@(q) \mid y \in L, q \in L \cup Q\}$.

We use *LP*(L,Q) as the set of all Boolean combinations of atoms of the form $y \sim c$ where $y \in Q \cup$ *VAR*(L,Q), '$\sim$' is one of $\leq, <, =, >, \geq$, and $c \in Q \cup \mathbb{N}$. An element in *LP*(L,Q) is called a *local predicate* of L and Q. For example, $x \leq 5 \wedge y@(q) = 3 \wedge x@(t) > 2$ is a local predicate in *LP*$(\{t,x,y\},\{q\})$.

A local predicate may contain references to local variables whose identities are interpreted with respect to the executing process. For example, $x \leq 5$ refers to a local variable x. When the inequality is used in the execution of a rule by process 5, then x is to be interpreted as $x@(5)$. For convenience, we define *inst*(η, p) as the instantiation of local predicate η with respect to process p. Inductively, it is defined as follows.

- *inst*$(\eta_1 \vee \eta_2, p) \;\equiv\;$ *inst*$(\eta_1, p) \vee$ *inst*(η_2, p)
- *inst*$(\neg\eta_1, p) \;\equiv\; \neg$*inst*$(\eta_1, p)$
- *inst*$(y \sim c, p) \;\equiv\; y[p] \sim c$
- for every $q \in L \cup Q$, *inst*$(y@(q) \sim c, p) \;\equiv\; y@(q@(p)) \sim c$
- for every $1 \leq p' \leq m$, *inst*$(y@(p') \sim c, p) \;\equiv\; y@(p') \sim c$.

We define $\eta_1 \wedge \eta_2$ and $\eta_1 \rightarrow \eta_2$ respectively as the shorthands for $\neg((\neg\eta_1) \vee (\neg\eta_2))$ and $(\neg\eta_1) \vee \eta_2$. *inst*(η, p) falls in the class of *instantiated local predicates*. For example,

$$\textit{inst}(x \leq 5 \wedge y@(q) = 3 \wedge x@(t) > 2, 5) \;\equiv\; x@(5) \leq 5 \wedge y@(q@(5)) = 3 \wedge x@(t@(5)) > 2$$

An *assignment* of L and Q is "$y := c$;" with $y \in$ *VAR*(L, Q) and $c \in \mathbb{N} \cup Q$. We use *ASEQ*(L, Q) for the set of all assignment sequences of L and Q.

Definition 4. **process rule system templates (PRST)** A *PRST* A is given as a tuple $\langle \Sigma, L, Q, E, \epsilon, \tau, \pi \rangle$ with the following restrictions. Σ is a finite set of event names. L is a finite set of local discrete state variable names while Q is the finite set of local place-holder names for process identifiers. For simplicity of presentation, we require that $Q \cap L = \emptyset$. E is a finite set of *local transition rules* (*rules* for short). $\epsilon : E \mapsto$ *SOSEQ*(Σ, Q) defines the sequence of sync-ops of each rule. $\tau : E \mapsto$ *LP*(L, Q) defines the triggering condition of each rule. $\pi : E \mapsto$ *ASEQ*(L, Q) defines the assignment sequence to local variables of each rule. For convenience, we require that there is a null rule $\perp \in E$ such that $\epsilon(\perp) = [\,]$, $\tau(\perp) =$ *true*, and $\pi(\perp) = [\,]$. ■

To specify the global states of a system, we define *SP*(L, m) as the set of all Boolean combinations of atoms of the form $y@(p) \sim c$ where $y \in L \cup X$, $1 \leq p \leq m$, '$\sim$' is one of $\leq, <, =, >, \geq$, and $c \in \mathbb{N}$. An element in *SP*(L, m) is called a *state predicate* of L and X with respect to concurrency m. For example, $x@(5) \leq 5 \wedge y@(1) = 3 \wedge x@(3) > 2$ is a state-predicate.

3.1 Global transitions of a network of PRST

A PRST cannot execute its rules by its own. According to the semantics of binary synchronization [**?**], a rule can be executed if and only if all its input sync-ops have been sent out by some processes at the same time and all its output sync-ops have also been received by some processes at the same time. There are several issues in defining semantically correct global transitions. In the following, we first define *transition plans* which may not result in sensible global transitions. Then we use examples to explain what could go wrong in the definition and propose restrictions to refine the definition.

Definition 5. **transition plan (TP)** A *transition plan (TP)* of a network of m PRST is conceptually a function from $\{1, \ldots, m\}$ to E and suggests the composition of a global transition. ■

A TP may not describe a consistent global transition in that some sync-ops are not responded in the global transition. We have the following example to explain the issue.

Intuitively, we say a TP is *consistent* if the following constraint is satisfied. For each $\sigma \in \Sigma$, the number of output sync-ops of event type σ must match the number of input sync-ops of event type σ in a TP. Let *value*$(?) = -1$ and *value*$(!) = 1$. For each $e \in E$ with $\epsilon(e) = [\langle d_1, \sigma_1, q_1 \rangle \ldots \langle d_n, \sigma_n, q_n \rangle]$, we let *ECount*$_\sigma(e) =$ *SUM*$_{1 \leq i \leq n; \sigma_i = \sigma}$*value*$(d_i)$. For convenience, if σ is not used in $\epsilon(e)$, then we let *ECount*$_\sigma(e) = 0$. The intuition is as follows.

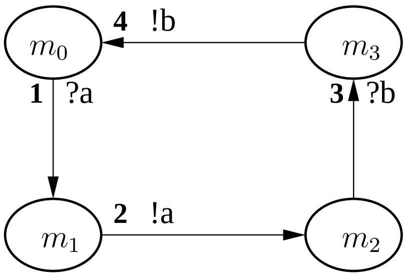


Fig. 4. PRTST for explaining minimality

- If $\mathit{ECount}_\sigma(e) > 0$, there are $\mathit{ECount}_\sigma(e)$ output sync-ops of event type σ in $\epsilon(e)$.
- If $\mathit{ECount}_\sigma(e) < 0$, there are $\mathit{ECount}_\sigma(e)$ input sync-ops of event type σ in $\epsilon(e)$.
- If $\mathit{ECount}_\sigma(e) = 0$, there is no sync-ops of event type σ in $\epsilon(e)$.

The consistency constraint of TP T means that $\forall\sigma \in \Sigma(\mathit{SUM}_{1\leq p\leq m}\mathit{ECount}_\sigma(T(p)) = 0)$.

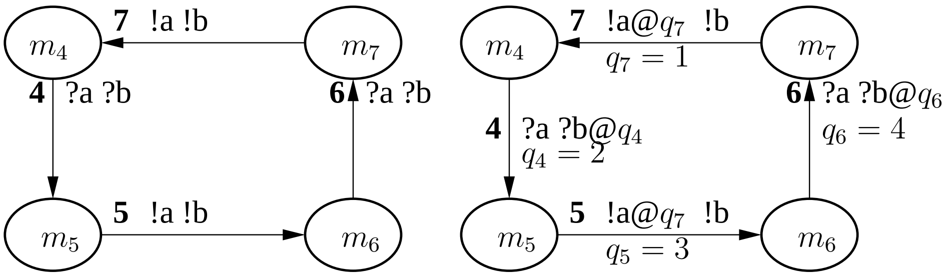
Example 4. An inconsistent TP We may have TP $g_1 = \{1 \leftarrow \mathbf{2}, 2 \leftarrow \mathbf{9}, 3 \leftarrow\perp\}$ for the concurrent system in example 9. g_1 is inconsistent in that the `!lock`$@q$ issued by process 1 with rule **2** is not responded by process 2 with rule **9**. ■

Here we have another example that motivates for the semantic constraints on TPs.

Example 5. Compatibility with interleaving semantics Suppose we have the PRST in figure 4 for four processes. The TP of $T_2 = \{1 \leftarrow \mathbf{1}, 2 \leftarrow \mathbf{2}, 3 \leftarrow \mathbf{3}, 4 \leftarrow \mathbf{4}\}$ can be partitioned into two TPs $T_3 = \{1 \leftarrow \mathbf{1}, 2 \leftarrow \mathbf{2}, 3 \leftarrow\perp, 4 \leftarrow\perp\}$ and $T_4 = \{1 \leftarrow\perp, 2 \leftarrow\perp, 3 \leftarrow \mathbf{3}, 4 \leftarrow \mathbf{4}\}$. If we allows for the atomic execution of T_2, it seems somewhat incompatible with the popular interleaving semantics of concurrent systems. In this work, we need constraints on the semantics of global transitions to eliminate such transition plans. ■

The constraints for compatibility with interleaving semantics discussed in example 5 is not at all straightforward. One naive semantic definition is to disallow any global transition whose TP can be broken down to two nontrivial consistent TPs. We use the following example to show why such a naive semantics may be contradictory to an intuitive model that the users want to construct.

Example 6. More on the compatibility with interleaving semantics Suppose we have the PRST in figure 5(a) for four processes. The TP of $T_5 = \{1 \leftarrow \mathbf{4}, 2 \leftarrow \mathbf{5}, 3 \leftarrow \mathbf{6}, 4 \leftarrow \mathbf{7}\}$ can actually be partitioned into two nontrivial consistent TPs $T_6 = \{1 \leftarrow \mathbf{4}, 2 \leftarrow \mathbf{5}, 3 \leftarrow\perp, 4 \leftarrow\perp\}$ and $T_7 = \{1 \leftarrow\perp, 2 \leftarrow\perp, 3 \leftarrow \mathbf{6}, 4 \leftarrow \mathbf{7}\}$. It can also be partitioned as $T_8 = \{1 \leftarrow \mathbf{4}, 2 \leftarrow\perp, 3 \leftarrow\perp, 4 \leftarrow \mathbf{7}\}$ and $T_9 = \{1 \leftarrow\perp, 2 \leftarrow \mathbf{5}, 3 \leftarrow \mathbf{6}, 4 \leftarrow\perp\}$. If we just examine the binary synchronization relation among the sync-ops, we could run into the conclusion that T_5 violates the interleaving semantics and should not be generated. But it could also happen that neither of the two decompositions in the above actually match the intention of the users. For example, the users may have actually put down constraints on the quantified process identifier place-holders and described the model as in figure 5(b). As can be seen, the users may actually want to model a circular synchronization with the four processes. We certainly do not want to rule out such synchronizations. ■



(a) (b)

Fig. 5. PRTST for explaining minimality

According to examples 5 and 6, we propose a *connectivity* requirement on transition functions. For convenience of the definition of the requirement, we need the following definitions. For each place-holder q in a rule executed by process p, we introduce an auxiliary local discrete variable $q[p]$. This is necessary since two rules respectively executed by two processes may contain place-holders with the same name. We assume that given a sequence λ, $|\lambda|$ is its length. Given A, we let *slen*(A) be $\max_{e \in E} |\epsilon(e)|$. For convenience of discussion, given a network m PRTSs, we assume the following sync-ops.

$\langle d_{1,1}, \sigma_{1,1}, q_1 \rangle$	$\langle d_{1,2}, \sigma_{1,2}, q_2 \rangle$	$\ldots$	$\langle d_{1,\mathit{slen}(A)}, \sigma_{1,\mathit{slen}(A)}, q_{\mathit{slen}(A)} \rangle$
$\langle d_{2,1}, \sigma_{2,1}, q_1 \rangle$	$\langle d_{2,2}, \sigma_{2,2}, q_2 \rangle$	$\ldots$	$\langle d_{2,\mathit{slen}(A)}, \sigma_{2,\mathit{slen}(A)}, q_{\mathit{slen}(A)} \rangle$
$\vdots$	$\vdots$		$\vdots$
$\langle d_{m,1}, \sigma_{m,1}, q_1 \rangle$	$\langle d_{m,2}, \sigma_{m,2}, q_2 \rangle$	$\ldots$	$\langle d_{m,\mathit{slen}(A)}, \sigma_{m,\mathit{slen}(A)}, q_{\mathit{slen}(A)} \rangle$

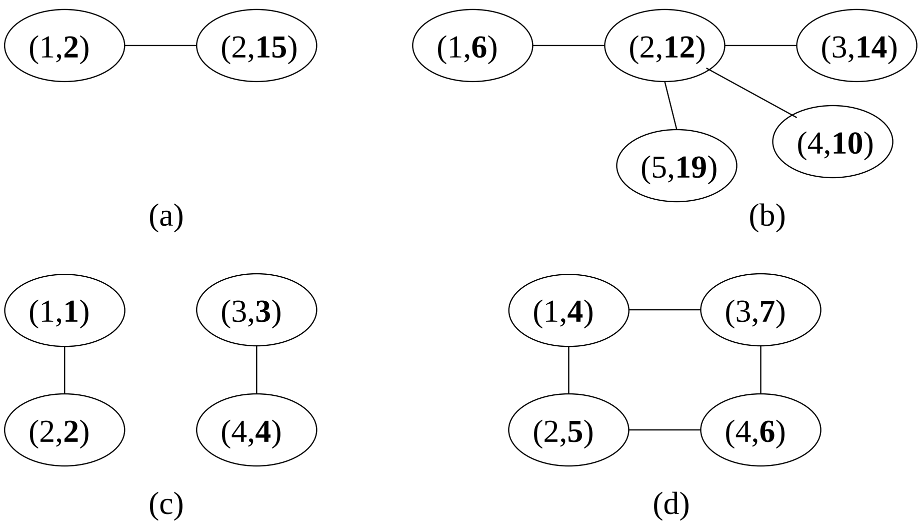
In this array for synchronization, we use the following $m \times \mathit{slen}(A)$ place-holders of the process identifiers.

$$\begin{array}{lll} q_1@(1) & q_2@(1) & \ldots q_{\mathit{slen}(A)}@(1) \\ \vdots & \vdots & \ldots \vdots \\ q_1@(m) & q_2@(m) & \ldots q_{\mathit{slen}(A)}@(m) \end{array}$$

In a global transition, according to the traditional semantics of binary synchronization in the literature [**?**], a binary synchronization must happen exactly between a pair of input-output sync-ops in the array. Moreover, the two sync-ops cannot participate in any other binary synchronization. To explain this precisely, we use the following matrix.

Definition 6. synchronization plan matrix (SPM) An SPM compatible with a transition plan T of a network of m process PRTS with a common template A is an $m \times \mathit{slen}(A)$ 2-dimensional matrix Ψ of integer pairs.

$\langle P_{1,1}, S_{1,1} \rangle$	$\langle P_{1,2}, S_{1,2} \rangle$	$\ldots$	$\langle P_{1,\mathit{slen}(A)}, S_{1,\mathit{slen}(A)} \rangle$
$\langle P_{2,1}, S_{2,1} \rangle$	$\langle P_{2,2}, S_{2,2} \rangle$	$\ldots$	$\langle P_{2,\mathit{slen}(A)}, S_{2,\mathit{slen}(A)} \rangle$
$\vdots$	$\vdots$		$\vdots$
$\langle P_{m,1}, S_{m,1} \rangle$	$\langle P_{m,2}, S_{m,2} \rangle$	$\ldots$	$\langle P_{m,\mathit{slen}(A)}, S_{m,\mathit{slen}(A)} \rangle$



(a) (b)

(c) (d)

Fig. 6. Synchronization trees

For each $1 \leq h \leq m$ and $1 \leq i \leq \mathit{slen}(A)$, $\Psi[h,i] = \langle P_{h,i}, S_{h,i} \rangle$ with the following restrictions. $P_{h,i}$ records the process identifier that responds to the i'th sync-op in rule $T(h)$ issued by process h. $S_{h,i}$ records the index of sync-op in $\epsilon(T(P_{h,i}))$ that responds to the i'th sync-op in rule $T(h)$ issued by process h. A basic requirement of an SPM is that if $\langle P_{h,i}, S_{h,i} \rangle = \langle k, j \rangle$, then $\langle P_{k,j}, S_{k,j} \rangle = \langle h, i \rangle$. An SPM Ψ is compatible with T if the following constraints hold.

- For any $1 \leq h \leq m$, if $T(h) = \perp$, then $\forall 1 \leq i \leq \mathit{slen}(A)(\langle P_{h,i}, S_{h,i} \rangle = \langle \perp, 0 \rangle)$.
- For any $1 \leq h \leq m$, if $T(h) \neq \perp$, then $\forall i > |\epsilon(T(h))|(\langle P_{h,i}, S_{h,i} \rangle = \langle \perp, 0 \rangle)$.
- For any $1 \leq h \leq m$ and $1 \leq i \leq \mathit{slen}(A)$, if $P_{h,i} \neq \perp$ and $S_{h,i} \neq 0$, then $\mathit{value}(d_{h,i}) + \mathit{value}(d_{P_{h,i}, S_{h,i}}) = 0$ and $\sigma_{h,i} = \sigma_{P_{h,i}, S_{h,i}}$. ■

Definition 7. **synchronization graph** Given an SPM Ψ compatible with a TP T, the *synchronization graph* $G(T, \Psi)$ of T and Ψ is defined as follows. The node set of $G(T, \Psi)$ is $\{(p, e) \mid 1 \leq p \leq m, T(p) = e \neq \perp\}$ while the edge set is $\{((p, e), (p', e')) \mid \exists h \exists h', (\Psi[p, h] = \langle p', h' \rangle)\}$. ■

Example 7. Synchronization graphs of global transitions Assume that we have a SAM system and 4 jet-fighters with models in figure 8. A synchronization graph for TP $\{1 \leftarrow \mathbf{2}, 2 \leftarrow \mathbf{15}, 3 \leftarrow \perp, 4 \leftarrow \perp, 5 \leftarrow \perp\}$ is in figure 6(a). A synchronization graph for TP $\{1 \leftarrow \mathbf{6}, 2 \leftarrow \mathbf{12}, 3 \leftarrow \mathbf{14}, 4 \leftarrow \mathbf{10}, 5 \leftarrow \mathbf{19}\}$ is in figure 6(b). The only synchronization graph for T_2 in example 5 is in figure 6(c) while the only triggerible synchronization graph for T_5 in figure 5(b) is in figure 6(d). ■

Definition 8. **global transitions** A global transition is a pair $\langle T, \Psi \rangle$ where T is a consistent TP and Ψ is an SPM such that $G(T, \Psi)$ is connected. ■

Intuitively, if $G(T, \Psi)$ is not connected, then we can break it into two unsynchronized global transitions, just like the case in figure 6(c). As can be seen, it is not possible to construct a

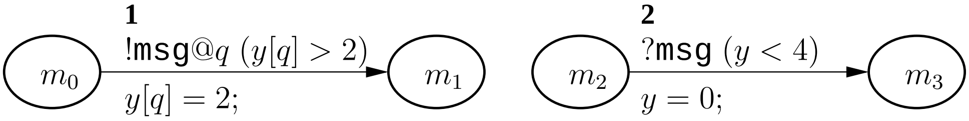


Fig. 7. PRTST for explaining ITR-conditions

connected synchronization graph for T_2. A global transition can be executed if we know both its TP and the compatible SPM that makes the synchronization graph connected.

In general, in a system, some processes may choose not to respond to a synchronization event sent to them in a complex synchronization. With definition 8, we can model such non-responding actions with dummy correspnding transitions that do nothing. It is also easy to extend our language to incorporate such flexibilities, i.e., not all parties have to respond to a set of simultaneous binary synchronizations. However, this is actually equivalent to adding those dummy corresponding transitions in an automatic preprocessing step.

However there could be *intra-transition race-conditions* (*ITR-condition*) in global transitions. An ITR-condition between two rules happens when the order of execution of the assignments in the two rules may affect the result of the synchronization between the two rules. Depending on the intention of the users, an ITR-condition could represent an anomaly in a model that the users are not aware of.

Example 8. Intra-transition race-conditions (ITR-condition) Suppose we have a concurrent system with PRTS templates depicted in figure 7. Suppose process 1 is to execute rule **1** while process 2 is to execute rule **2**. In this case, the $y@(q)$ in rule **1** and the y in rule **2** are aliases for $y@(2)$. If the assignment of rule **1** happens before that of rule **2**, $y@(2) = 0$ after the synchronization. Otherwise, $y@(2) = 2$ after the synchronization. This is a *write-write* ITR-condition between processes 1 and 2. ■

In a general distributed system model, there could also be read-write ITR-conditions. In our models, because the assignments are executed only after the triggering conditions of all the synchronizing rules have been satisfied and because we only assign constants to variables, only write-write ITR-conditions are possible. In the semantic definition of the concurrent networks of PRTSs, we assume that all global transitions are ITR-condition free. In subsection **??**, we present algorithms to check and eliminate ITR-conditions from global transition characterizations.

Definition 9. **states** Suppose we are given a network of m PRST with a common template $\langle \Sigma, L, Q, E, \epsilon, \tau, \pi \rangle$. A state for the network is a valuation $\nu : \{x@(p) \mid x \in L, 1 \leq p \leq m\} \mapsto \mathbb{N}$.

An *extended state* ν with a set Q of process identifier place-holders is a state extended with function from $\{q@(p) \mid q \in Q, 1 \leq p \leq m\}$ to $\{1, \ldots, m\}$. Given a state ν and a global transition $\langle T, \Psi \rangle$, the extended state $\nu\langle T, \Psi \rangle$ is identical to ν except that for each $1 \leq h \leq m$ and $1 \leq i \leq \mathit{slen}(A)$, if $\Psi[h, i] = \langle P, S \rangle$, $\nu\langle T, \Psi \rangle(q_i@(h)) = P$. ■

Definition 10. **Concurrent game graph from a network of PRSTs** Given m PRSTs with a common template $\langle \Sigma, L, Q, E, \epsilon, \tau, \pi \rangle$, we define its concurrent game graph G as follows.

- $m_G = m$.

- Q_G is the set of states of the network of the m PRSTs.
- $q_{0,G}$ is the initial state of G and
- P_G can be defined as the set of state inequalities that can be defined on L for the processes.
- λ_G labels elements in P_G on states in Q_G such that consistency among the inequalities is maintained.
- R_G is the set of global transitions of the network of the PRSTs.
- Σ_G is the set of finite sequences of elements of the form: $!(k)e$ or $?(k)e$ where k is an integer that denotes the number of copies of event e (respectively issued or received).
- $\delta_G : R_G \mapsto \Sigma_G^m$ maps each transition to a vector of m event sequences in Σ_G. The p'th component, $p \in [1, m]$, in the vector denotes the events issued by process p in executing the global transition.

The *concurrency* (or number of processes) of the system is m. ■

The general scheme of networks of process automatas allows for the modeling of broadcasting and multicasting of many generic transmission events.

Example 9. Fighters and SAM system In figure 8, we show our PRST for the system of fighters and SAM described in figure 1. We use integer 1 for the identifier of the SAM system while integers $2, \ldots, m$ for those of the $m-1$ jet-fighters. The connected graph in the upper-half represents the transition-diagram of the SAM while the one in the lower-half represents that of the fighters. The SAM has three operation modes: `SI`(SAM idle), `ST`(SAM tracking), and `SD`(SAM destroyed). The SAM starts it execution form mode `SI`. The fighters have three operation modes: `JF`(jet-fighter faraway), `JIR`(jet-fighter in range), and `JEM`(jet-fighter in evasive maneuver). The fighters start their execution from mode `JF`. The SAM and the fighters interact through events `lock`, `lockquery`, `bomb`, and `hit`. The fighters also communicate with one another through event `explosion` to model the observation of the explosion at the hitting of a missile.

x is a local clock name. t is another local integer variable name that records the identifier of the fighter that has been locked by the SAM. There is also an implicit integer variable name `mode` that represents the operation modes of each process.

Note that the triggering condition of transition **2** is a quantified predicate. Without loss of generality, we use this as a shorthand for the disjunction of the instantiated quantified subformulas. In fact, our implementation does accept such quantified predicates.

The initial condition of the concurrent game graph is $\texttt{mode}[1] = \texttt{SI} \wedge x[1] = 0 \wedge \bigwedge_{1<p\leq m}(\texttt{mode}[p] = \texttt{JF} \wedge x[p] = 0)$. The invariance condition is $(\texttt{mode}[1] = \texttt{SI} \vee \texttt{mode}[1] = \texttt{ST} \vee \texttt{mode}[1] = \texttt{SD}) \wedge x[1] \leq 1 \wedge \bigwedge_{1<p\leq m}(\texttt{mode}[p] = \texttt{JF} \vee \texttt{mode}[p] = \texttt{JIR} \vee \texttt{mode}[p] = \texttt{JEM})$. The destruction of the SAM is modeled with event `bomb` which can happen if a jet-fighter stay in mode `JIR` for more than 13 time units. The SAM can take down a jet-fighter by tracking it in one time units with event `hit`. The SAM launcher wants to avoid being bombed.

We have also labeled each rule a bold-face number for the convenience of latter discussion. Global transitions can be flexibly constructed out of the binary synchronization events. For example, to model the bombing of the a jet-fighter, the SAM synchronizes with a jet-fighter through event `bomb`. This may happen between the SAM process using one rule of **3**, **5**, **8** with any jet-fighter process using rule **11**. Thus in total, there could be $3 \times (m-1)$ such global transitions for the modeling of jet-fighter bombing. But in the example, we successfully decompose the many global transitions into the modular descriptions of three SAM rules and one jet-fighter rule. As the number of jet-fighter processes increases, the benefit for conciseness in modular model construction with our scheme will become even more salient. ■

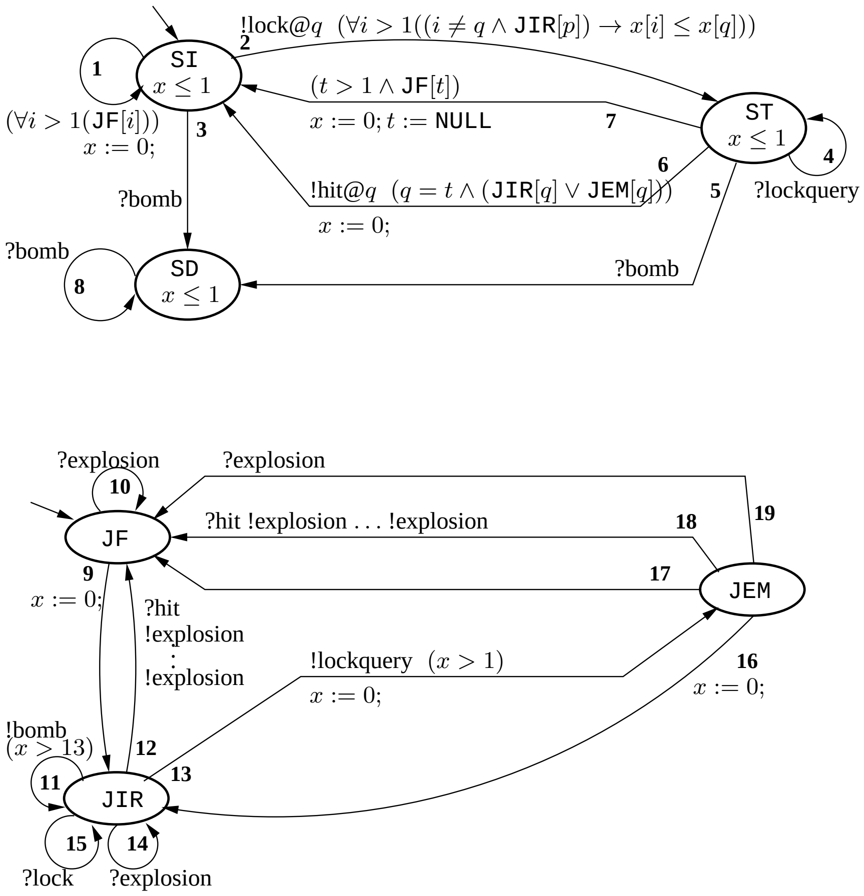


Fig. 8. Model templates of fighters and SAM

4 Alternating-time μ-calculus (AMC)

4.1 Syntax

For a turn-based game graph G of m agents, an AMC formula ϕ is constructed with the following syntax rule.

$$\phi ::= p \mid X \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \mathbf{lfp} X.\phi_1 \mid \langle A \rangle \bigcirc \phi_1$$

Here p is an atomic proposition in P_G. X is a variable for sets of states, i.e., subsets of Q_G. We require that X is under a syntax path of even number of negation from its least fixpoint quantifier. Formula $\mathbf{lfp}X.\phi_1$ is a least fixpoint formula that describes the minimal set that satisfies $\phi_1(X) = X$ when ϕ_1 is considered a function with free variable X.

‘A’ is a sequence of integers in $[1, m]$. Operators of the form $\langle A \rangle$ are called a *strategy quantifier* (*SQ*). Formulas starting with an SQ are defined as for ATL [1] and mean that there exist strategies of the agents in $\{A\}$ to enforce the development of only plays satisfying ψ.

A set variable X is *free* in an AMC formula if it is not within the scope of a fixpoint operator on X in the formula. If it is not free in a formula, then it is *bound* in the formula. A formula is *well-formed* and called a *sentence* if all its set variables are bound and all SIQs occurs in the scope of an SQ.

For convenience, some useful shorthands follows.

$$\begin{aligned}
\textit{true} &\equiv p \vee (\neg p) \\
\textit{false} &\equiv \neg \textit{true} \\
\phi_1 \wedge \phi_2 &\equiv \neg((\neg\phi_1) \vee (\neg\phi_2)) \\
\phi_1 \Rightarrow \phi_2 &\equiv (\neg\phi_1) \vee \phi_2 \\
\mathbf{gfp} X.\phi_1(X) &\equiv \neg \mathbf{lfp} X.\neg\phi_1(\neg X) \\
\langle A \rangle \phi_1 \mathrm{U} \phi_2 &\equiv \mathbf{lfp} X.(\phi_2 \vee (\phi_1 \wedge \langle A \rangle \bigcirc X)) \\
\langle A \rangle \Diamond \phi_1 &\equiv \langle A \rangle \textit{true}\, \mathrm{U} \phi_1 \\
\langle A \rangle \Box \phi_1 &\equiv \mathbf{gfp} X.(\phi_1 \wedge \langle A \rangle \bigcirc X) \\
[A] \bigcirc \phi_1 &\equiv \neg \langle A \rangle \bigcirc \neg\phi_1 \\
[A] \Diamond \phi_1 &\equiv \neg \langle A \rangle \Box \neg\phi_1 \\
[A] \Box \phi_1 &\equiv \neg \langle A \rangle \Diamond \neg\phi_1
\end{aligned}$$

Given a finite set K, we let $\overline{K}$ be the sequence of elements in K in some dictionary order. For turn-based games, we also have $[A]\psi \overset{\text{def}}{\equiv} \langle \overline{[1, m] - \{A\}} \rangle \psi$.

Example 10. : Consider the following formulas for a concurrent game of three agents.

$$\begin{aligned}
\langle 1, 2 \rangle \Box \neg \text{risk} &\equiv \mathbf{gfp} X.((\neg \text{risk}) \wedge \langle 1, 2 \rangle \bigcirc X) \\
&\equiv \neg \mathbf{lfp} X.\neg((\neg \text{risk}) \wedge \langle 1, 2 \rangle \bigcirc \neg X) \\
&\equiv \neg \mathbf{lfp} X.(\text{risk} \vee \neg \langle 1, 2 \rangle \bigcirc \neg X) \\
&\equiv \neg \mathbf{lfp} X.(\text{risk} \vee \neg \langle 1, 2 \rangle \bigcirc \neg X)
\end{aligned}$$

Yet another!

$$\langle 1, 2 \rangle \text{booting} \mathrm{U} \text{boot\_success} \equiv \mathbf{lfp} X.(\text{boot\_success} \vee (\text{booting} \wedge \langle 1, 2 \rangle \bigcirc X))$$

Here is a strange formula.

$$\mathbf{gfp} X.(p \wedge \mathbf{lfp} Y.((X \wedge q) \vee (Y \wedge \neg q)))$$

Here is another strange formula.

$$\mathbf{gfp} X.(p \wedge \mathbf{lfp} Y.(((q \wedge \langle 1, 2 \rangle \bigcirc X) \vee (\neg q \wedge \langle 2, 3 \rangle \bigcirc (Y \wedge \neg p))))$$

■

4.2 Semantics

Given an agency $\{A\}$ and two transition $(q, q'), (q, q'') \in R_G$, we say (q, q') matches (q, q'') in $\{A\}$, in symbols $(q, q') \overset{A}{\equiv} (q, q'')$, if and only if for each $a \in \{A\}$, $\delta_G((q, q'), a) = \delta_G((q, q''), a)$.

An AMC subformula is interpreted with respect to an interpretation, called an *environment*, of the fixpoint variables. Given an environment $\mathcal{E}$, an formula ψ is satisfied at a state q, in symbols $G, q \models_{\mathcal{E}} \psi$, if and only if the following inductive constraints are satisfied.

- $G, q \models_{\mathcal{E}} p$ if and only if $q \in Q_G$ and $p \in \lambda(q)$.
- $G, q \models_{\mathcal{E}} X$ if and only if $q \in \mathcal{E}(X)$.
- $G, q \models_{\mathcal{E}} \neg\phi_1$ if and only if it is not the case that $G, q \models_{\mathcal{E}} \phi_1$.
- $G, q \models_{\mathcal{E}} \phi_1 \vee \phi_2$ if and only if either $G, q \models_{\mathcal{E}} \phi_1$ or $G, q \models_{\mathcal{E}} \phi_2$.
- $G, q \models_{\mathcal{E}}$ **lfp**$X.\phi_1$ if and only if $q \in \bigcup_{k\in\mathbb{N}} X_k$ where for all $k \in \mathbb{N}$, X_k is inductively defined as $X_0 = \emptyset$ and for every $k > 0$,
$$X_k = X_{k-1} \cup \{q' \mid G, q' \models_{\mathcal{E}[X\mapsto X_{k-1}]} \textit{Scope}(X)\}.$$
- $G, q \models_{\mathcal{E}} \langle A\rangle \bigcirc \phi_1$ if and only if there exists a (q, q') such that for every $(q, q'') \overset{A}{\equiv} (q, q')$, $G, q'' \models_{\mathcal{E}} \phi_1$.

For convenience, we let $\emptyset$ be a null mapping, i.e., a function that is undefined on everything. If q is a state and ϕ_1 such that for every environment $\mathcal{E}$, $G, q \models_{\mathcal{E}} \phi_1$, then we may simply write $G, q \models \phi_1$. If $G, q_{0G} \models \phi_1$, then we may simply write $G \models \phi_1$.

5 Model-checking algorithm for AMC

We assume that all AMC sentences given to us are in *positive normal form* (*PNF*). That is, we assume that all negation only happens right before atomic proposition symbols.

Table 1. A model-checking algorithm for AMC and μ-calclulus

```
mck($G,\psi$) // Assume a given $\mu$-calculus sentence $\psi$ with distinct fixpoint variables
          // $X_1,\ldots,X_n$ in the decreasing order of scope sizes.
  Let $\mathcal{E}$ be env($\emptyset, 1$).
  if $q_{0,G} \in$ eval($\mathcal{E},\psi$) then return true. else return false. end if
```

6 Procedures for precondition calculation

Basically, you need loops that evaluate the fixpoint formulas. In each loop iteration, you need to iterate through all the synchronous transitions. The synchronous transitions are indexed from zero to `red_query_sync_xtion_count()`-1.

```
extern int  red_query_sync_xtion_count(
int  flag_sync_xtion_table_choice
// This argument is used to choose from the declared
// synchronous transition table and the
// game synchronous transition table.
// There are the following two values.
// RED_USE_GAME_SYNC_XTION
// RED_USE_DECLARED_SYNC_XTION
);
```

Table 2. An algorithm for constructing fixpoint environments

```
$env(\mathcal{E}, i)$ // Assume a given $\mu$-calculus sentence $\psi$ with distinct fixpoint variables
        // $X_1, \ldots, X_n$ in the decreasing order of scope sizes.
if $i > n$ then
    return $\mathcal{E}$.
else if $X_i$ is a least fixpoint variable then
    Let $V$ be $\emptyset$ and $V'$ be $Q_G$.
    while $V \neq V'$ do
        Let $V'$ be $V$.
        Let $\mathcal{E}'$ be $env(\mathcal{E}[X_i \mapsto V], i+1)$.
        Let $V$ be $V \cup eval(\mathcal{E}', Scope(X_i))$.
    end while
    return $\mathcal{E}[X_i \mapsto V]$.
end if
```

Table 3. An algorithm for subformula evaluation with an environment

```
$eval(\mathcal{E}, \phi)$ // Assume a given $\mu$-calculus sentence $\psi$ with distinct fixpoint variables
        // $X_1, \ldots, X_n$ in the decreasing order of scope sizes.
if $\phi$ is $false$ then
    return $\emptyset$.
else if $\phi$ is $p$ then
    return $\{q \mid q \in Q_G, p \in \lambda_G(q)\}$.
else if $\phi$ is $X$ or lfp$X.\phi_1$ then
    return $\mathcal{E}(X)$.
else if $\phi$ is $\neg\phi_1$ then
    return $Q_G - eval(\mathcal{E}, \phi_1)$.
else if $\phi$ is $\phi_1 \vee \phi_2$ then
    return $eval(\mathcal{E}, \phi_1) \cup eval(\mathcal{E}, \phi_2)$.
else if $\phi$ is $\langle A \rangle \bigcirc \phi_1$ then
    return $\left\{ q \,\middle|\, (q,q') \in R_G, \forall (q,q'') \in R_G((q,q') \overset{A}{\equiv} (q,q'') \Rightarrow q'' \in eval(\mathcal{E}, \phi_1)) \right\}$.
end if
```

Then for each synchronous transition, you need the following procedure to calculate the preconditions.

```
extern redgram red_sync_xtion_bck(
redgram  ddst,
redgram  dpath,
int  flag_sync_xtion_table_choice,
// This argument is used to choose from the declared
// synchronous transition table and the
// game synchronous transition table.
// There are the following two values.
// RED_USE_GAME_SYNC_XTION
// RED_USE_DECLARED_SYNC_XTION
int sxi,
int flag_game_roles,
// The argument consists of three flag values.
// Each flag may appear or not.
// The three flag values are
// RED_GAME_MODL,
// RED_GAME_SPEC,
// RED_GAME_ENVR.
int  flag_time_progress,
// Two argument values are
// RED_NO_TIME_PROGRESS and
// RED_TIME_PROGRESS.
int flag_normality,
// Some argument values are
// RED_NORM_ZONE_NONE,
// RED_NORM_ZONE_MAGNITUDE_REDUCED, and
// RED_NORM_ZONE_CLOSURE.
int flag_action_approx,
// Some argument values are
// RED_NO_ACTION_APPROX,
// RED_ACTION_APPROX_NOXTIVE, and
// RED_ACTION_APPROX_UNTIMED.
int flag_reduction,
// Two argument values are
// RED_NO_REDUCTION_INACTIVE and
// RED_REDUCTION_INACTIVE.
int flag_state_approx,
// The argument values is the bit-wise
// of four flag values of the following form:
// fm | fs | fe | fg.
// fm is for abstraction of the model variables
// and can be of the following values:
// RED_NOAPPROX_MODL_GAME
// RED_OAPPROX_MODL_GAME_DIAG_MAG
// RED_OAPPROX_MODL_GAME_DIAGONAL
// RED_OAPPROX_MODL_GAME_MAGNITUDE
// RED_OAPPROX_MODL_GAME_UNTIMED
// RED_OAPPROX_MODL_GAME_MODE_ONLY
// RED_OAPPROX_MODL_GAME_NONE.
//
// fs is for abstraction of the specification variables
// and can be of the following values:
// RED_NOAPPROX_SPEC_GAME
// RED_OAPPROX_SPEC_GAME_DIAG_MAG
// RED_OAPPROX_SPEC_GAME_DIAGONAL
// RED_OAPPROX_SPEC_GAME_MAGNITUDE
// RED_OAPPROX_SPEC_GAME_UNTIMED
// RED_OAPPROX_SPEC_GAME_MODE_ONLY
// RED_OAPPROX_SPEC_GAME_NONE
//
// fe is for abstraction of the environment variables
// and can be of the following values:
// RED_NOAPPROX_ENVR_GAME
// RED_OAPPROX_ENVR_GAME_DIAG_MAG
// RED_OAPPROX_ENVR_GAME_DIAGONAL
// RED_OAPPROX_ENVR_GAME_MAGNITUDE
// RED_OAPPROX_ENVR_GAME_UNTIMED
// RED_OAPPROX_ENVR_GAME_MODE_ONLY
// RED_OAPPROX_ENVR_GAME_NONE
//
// fe is for abstraction of the global variables
// and can be of the following values:
// RED_NOAPPROX_GLOBAL_GAME
// RED_OAPPROX_GLOBAL_GAME_DIAG_MAG
// RED_OAPPROX_GLOBAL_GAME_DIAGONAL
// RED_OAPPROX_GLOBAL_GAME_MAGNITUDE
// RED_OAPPROX_GLOBAL_GAME_UNTIMED
// RED_OAPPROX_GLOBAL_GAME_MODE_ONLY
// RED_OAPPROX_GLOBAL_GAME_NONE
int flag_symmetry,
// Some possible argument values are
// RED_NO_SYMMETRY,
// RED_SYMMETRY_ZONE,
// RED_SYMMETRY_DISCRETE,
// RED_SYMMETRY_POINTER,
// RED_SYMMETRY_STATE.
int  flag_experiment
);
```

However, you need the following procedures to query the events of transitions executed by each agent.

```
extern int  red_query_sync_xtion_party_count(
int  flag_sync_xtion_table_choice,
// This argument is used to choose from the declared
// synchronous transition table and the
// game synchronous transition table.
// There are the following two values.
// RED_USE_GAME_SYNC_XTION
// RED_USE_DECLARED_SYNC_XTION
int  sxi
);

extern int  red_query_sync_xtion_party_process(
  int  flag_sync_xtion_table_choice,
  int sxi,
  int party_index
);



extern int  red_query_sync_xtion_party_xtion(
  int  flag_sync_xtion_table_choice,
  int sxi,
  int party_index
);

extern int  red_query_xtion_count();
extern int  red_query_xtion_attribute(int xi, int attr);


extern int red_query_xtion_sync_attribute(int xi, int si, int attr);
/*  The values of attr can be RED_XTION_SYNC_DIRECTION,
 *  RED_XTION_SYNC_QUANTIFIED_ADDRESS, RED_XTION_SYNC_VAR_INDEX, and
 *  RED_XTION_SYNC_QFD_CORRESPONDENCE_VAR_INDEX.
 *  The return values for attribute RED_XTION_SYNC_DIRECTION are
 *  RED_XTION_SYNC_XMIT and RED_XTION_SYNC_RECV.
 *  The return values for attribute case RED_XTION_SYNC_QUANTIFIED_ADDRESS
 *  can be RED_XTION_SYNC_NO_CORRESPONDENCE_REQUIREMENT,
 *  RED_XTION_SYNC_QUANTIFIED_CORRESPONDENCE_VAR, and
 *  RED_XTION_SYNC_CORRESPONDENCE_EXPRESSION.
 *  The return values for attribute RED_XTION_SYNC_VAR_INDEX
 *  is the variable index for the synchronizer.
 *
 *  If attribute RED_XTION_SYNC_QUANTIFIED_ADDRESS is of value
 *  RED_XTION_SYNC_QUANTIFIED_CORRESPONDENCE_VAR,
 *  then the return values for attribute
 *  RED_XTION_SYNC_QFD_CORRESPONDENCE_VAR_INDEX is the
 *  variable index for the holder of the address of the process
 *  corresponding to this synchronization.
 *  Otherwise, the return value is RED_FLAG_UNKNOWN.
 */
```

```
extern char
  // The following procedure can be used to query the string for a
  // synchronization in a transiton.
  // To get the synchronization names of a transition with index xi,
  // we can execute the following loop.
  /* for (i = 0;
  //      i < red_query_xtion_attribute(xi, RED_XTION_SYNC_COUNT);
  //      i++
  //      ) {
  //   printf("\nxi:%1d, si:%1d, %s", xi, i,
  //     red_query_string_xtion_sync(xi,i)
  //   );
  // }
  // printf("\n");
  */
  *red_query_string_xtion_sync(
    int, // This is a transition index
    int  // This is a sync index in the list of synchronizations of the
         // the transition.
  ),
  *red_query_string_xtion_sync_correspondence_exp(
    int, // This is a transition index
    int  // This is a sync index in the list of synchronizations of the
         // the transition.
  );
extern char  *red_query_string_xtion_action(int, int);
extern char  *red_query_string_xtion(int, int);
```

7 How to implement the algorithm ?

In the following, we explain how to implement procedure *eval*$()$ with REDLIB. We assume that $\mathcal{E}$ is implemented as an array of $1 \ldots n$ fixpoint variables.

- Case ϕ is *false*: return `red_false()`.
- Case p: return `red_diagram(`p`)`.
- Case X_i or **lfp**$X_i.\phi_1$: return $\mathcal{E}[i]$.
- Case $\neg\phi_1$: return `red_not`$($*eval*$(\mathcal{E}, \phi_1))$.
- Case $\phi_1 \vee \phi_2$: return `red_or`$($*eval*$(\mathcal{E}, \phi_1)$, *eval*$(\mathcal{E}, \phi_2))$.
- Case $\langle A \rangle \bigcirc \phi_1$: Proceed in the following step.
  - Let R be `red_false()`.
  - Let X be *eval*$(\mathcal{E}, \phi_1)$.
  - Unmark all sxi from 1 to `red_query_sync_xtion_count(RED_USE_DECLARED_SYNC_XTION)`-1.
  - For sxi := 1 to `red_query_sync_xtion_count(RED_USE_DECLARED_SYNC_XTION)`-1, do the following.
    - ∗ If sxi has been marked, continue.
    - ∗ Let S be `red_true()`.
    - ∗ For sxj := sxi to `red_query_sync_xtion_count(RED_USE_DECLARED_SYNC_XTION)`-1, do the following.
      - · If sxi and sxj are not the same in A's events, continue;
      - · Mark sxj.
      - · Let S be

        ```
        red_and(S, red_sync_xtion_bck(X,
            red_query_diagram_global_invariance(),
            RED_USE_DECLARED_SYNC_XTION
            sxj,
            RED_GAME_MODL | RED_GAME_SPEC | RED_GAME_ENVR,
            RED_NO_TIME_PROGRESS,
            RED_NORM_ZONE_NONE,
            RED_NO_ACTION_APPROX,
            RED_NO_REDUCTION_INACTIVE,
            0,
            RED_NO_SYMMETRY,
            0
        ) ).
        ```
    - ∗ Let R be `red_or`(R, S).
  - return R.

  For example, assume that we have five processes. Then $\langle 2, 3 \rangle \bigcirc p$ means that for each event combination of 2,3, all combinations of 1,4,5 must end at a p state. Thus we need to disjunct the preconditions of all the event combinations of 2,3 subject to the conjunction of all event combinations of 1,4,5.

There are several implementation details. Specifically, we need to know how to check whether sxi and sxj share the same event combinations with respect to A. You need to use

```
    red_query_sync_xtion_party_count(
      RED_USE_DECLARED_SYNC_XTION, sxi
    )
```

to get the number of processes participating in sxi. Then you need to use

```
    red_query_sync_xtion_party_process(
      RED_USE_DECLARED_SYNC_XTION, sxi, ipi
    )
```

and

```
    red_query_sync_xtion_party_xtion(
      RED_USE_DECLARED_SYNC_XTION, sxi, ipi
    )
```

to get the process index and transition index of the ipi'th party. Then if sxi and sxj do not have the same participation in A, then obviously they do not share the same event combination. If they have the same participation, then we check for each pi in A that participating in both sxi and sxj, if their corresponding transition index share the same events. (Note that in general, the two same process index may not occur in the same position in the party vector of sxi and sxj.)

Suppose that we now know party ipi of sxi and party ipj of sxj are the same and both in A. Further suppose taht their corresponding transition indices are respectively xi and xj out of

```
    red_query_sync_xtion_party_xtion(
      RED_USE_DECLARED_SYNC_XTION, sxi, ipi
    )
```

and

```
    red_query_sync_xtion_party_xtion(
      RED_USE_DECLARED_SYNC_XTION, sxj, ipj
    )
```

Then we can use

```
  red_query_xtion_attribute(xi, RED_XTION_SYNC_COUNT)
```

to get the event count of transition xi. Then we can use

```
  red_query_string_xtion_sync(xi,i)
```

to get the string representation of the i'th event of transition xi. We can also use

```
  red_query_string_xtion_sync_correspondence_exp(
    xi, i
  )
```

to get the address string representation of the i'th event of transiton xi.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, September 2002.