```
config EssentialActivities
{
    switch TestEnabled = false;

    action abstract static void Implementation.Eat();
    action abstract static void Implementation.Drink();
}

config PartyActivities: EssentialActivities
{
    action abstract static void Implementation.Dance();
    action abstract static void Implementation.Sing();
    action abstract static void Implementation.Leave();
    action abstract static void Implementation.KeepPartying();
}

config RegularActivities: EssentialActivities
{
    action abstract static void Implementation.Fast();
    action abstract static void Implementation.Sleep();
}

config AllActivities: PartyActivities, RegularActivities
{ }


/// Union operator (|): builds the union of its operands' traces.
/// Tight sequence operator (;): traces of the second operand are
///     appended to traces of the first one.
/// Optional operator (?): traces are valid with or without the optional
///     sub-behavior.
/// The possible traces of machine Party are
///     (a) Dance then Sing then end or KeepPartying
///     (b) Eat then Drink then end or KeepPartying
/// Notice there are two accepting states before and after KeepPartying.
machine Party() : PartyActivities
{
    (
        (Dance; Sing)
        |
        (Eat; Drink)
    );
    KeepPartying?
}

/// On a non-party day, one can either Eat then Drink,
/// or Fast. In both cases, the day ends by optionally going to Sleep.
machine NoParty() : RegularActivities
{
    (
        (Eat; Drink)
        |
        Fast
    );
    Sleep?
}

/// Synchronized Parallel Composition operator (||).
/// All steps of the composed behaviors must be synchronized;
/// steps that cannot be synchronized are excluded from the result.
/// The offered signature of this behavior is the intersection
/// of the signatures of the subbehaviors.
/// Machines Party and NoParty can synchronize on the Eat then Drink path.
/// But they fail to synchronize from then on, so the behavior ends there.
machine SyncParallel() : AllActivities
{
```

```
    Party || NoParty
}

/// Interleaved Parallel Composition operator (|||).
/// All steps of the sub-behaviors are produced;
/// The offered signature of this behavior is the union
/// of the signatures of the sub-behaviors.
/// Machine InterleavedParallel can produce all possible
/// interleavings of machines Party and NoParty.
machine InterleavedParallel() : AllActivities
{
    Party ||| NoParty
}

/// Synchronized-Interleaved Parallel Composition operator (|?|).
/// Actions in the intersection of the sub-behavior signatures must
/// synchronize, while the remaining actions are interleaved.
/// The offered signature is the union of the signatures of the subbehaviors.
/// In SyncInterleavedParallel machine, Eat and Drink are in the intersection of
/// configs PartyActivities and RegularActivites. All other paths are interleaved.
machine SyncInterleavedParallel() : AllActivities
{
    Party |?| NoParty
}

/// Tight Sequencing operator (;).
/// The (tight) sequencing operation denotes the set of traces obtained by
/// concatenating the traces of the first operand to those of the second one.
/// Notice how the second behavior is appended at all accepting states.
machine TightSequence() : AllActivities
{
    Party; NoParty
}

/// Loose Sequencing operator (->).
/// The loose sequencing operation allows an arbitrary number of
/// actions from the context signature to occur between its first
/// and its second operand.
/// Again, appending occurs in all accepting states.
machine LooseSequence() : AllActivities
{
    Party -> NoParty
}

/// Permutation operator (&).
/// Constructs all possible permutations of two behaviors treated
/// as atomic (no interleaving of individual actions is produced).
machine Permutation() : AllActivities
{
    (Dance;Sleep) & (Eat; Drink)
}

/// Zero or More Repetitions operator (*).
/// Builds a behavior consisting zero or more occurrences of its operand.
/// Notice the accepting initial state (allowing the empty behavior) and
/// and the looping paths, allowing the behavior to restart.
machine ZeroOrMore() : AllActivities
{
    Party*
}

/// One or More Repetitions operator (+).
/// Builds a behavior consisting arbitrary occurrences of its operand.
/// Notice the non-accepting initial state, meaning the empty trace
/// is not included.
machine OneOrMore() : AllActivities
```

```
{
    Party+
}

/// Optional operator (?).
/// Makes a whole behavior optional.
machine Optional() : AllActivities
{
    Party? ; NoParty
}

/// Bounded Repetition operator ({n})
/// Builds a behavior consisting exactly n times of its operand.
machine BoundedRepetitionExact() : AllActivities
{
    NoParty{2}
}

/// Bounded Repetition operator ({n, })
/// Builds a behavior consisting at least n times of its operand.
machine BoundedRepetitionLeast() : AllActivities
{
    NoParty{2,}
}

/// Bounded Repetition operator ({n, m})
/// Builds a behavior consisting at least n times and at most m times of its operand.
machine BoundedRepetitionRange() : AllActivities
{
    NoParty{2, 3}
}

/// Any Action Universal Behavior (_).
/// An underscore used as a behavior stands for the invocation of any single
/// atomic action in the context signature.
/// Notice that two underscores are required to represent a call and its return.
machine AnyAction() : PartyActivities
{
    Eat; _ ; _
}

/// Any Sequence Universal Behavior (...).
/// An ellipsis represents zero or more repetitions of any actions
/// in the context signature. It is equivalent to _*.
machine RepetitionOfAnyAction() : PartyActivities
{
    Eat;...
}

/// Negation operator (!)
/// Builds a behavior consisting actions except its operand.
/// Notice that negation can only be applied on atomic actions (call, return or event)
/// Machine Nagation() indicates the behavior of all actions except (call Dance)
machine Negation() : PartyActivities
{
    ((! call Dance)*)||RepetitionOfAnyAction
}

/// Another machine to illustrate the usage of Negation
/// Composing a behavior that performs action Dance with machine Negation results in all paths that had ↙
    Dance as a step being truncated.
/// Construct accpeting paths is then used in this example to remove the truncated paths.
machine Truncation() : PartyActivities
{
    construct accepting paths for
    (Negation || (Eat;(Drink; Dance; Sing|Drink; Sing; Dance|Drink; Eat; Sing)))
```

```
}
```