Automatic Translation of tock-CSP into Timed Automata

Technical Report

Abdulrazaq Abba, Ana Cavalcanti, Jeremy Jacob
Department of Computer Science
University of York, UK

15th August 2020

Abstract: The process algebra `tock-CSP` provides textual notations for modelling discrete-time behaviours, with the support of various tools for verification. Similarly, automatic verification of Timed Automata (TA) is supported by the real-time verification toolbox Uppaal. TA and `tock-CSP` differ in both modelling and verification approaches. For instance, liveness requirements are difficult to specify with the constructs of `tock-CSP`, but they are easy to verify in Uppaal. In this work, we translate `tock-CSP` into TA to take advantage of Uppaal. We have developed a translation technique and tool; our work uses rules for translating `tock-CSP` into a network of small TA, which address the complexity of capturing the compositionality of `tock-CSP`. For validation, we use an experimental approach based on finite approximations to trace sets. We plan to use mathematical proof to establish the correctness of the rules that will cover an infinite set of traces.

Contents

1. Introduction

Communicating Sequential Processes (CSP) is an established process algebra that provides a formal notation for both modelling and verification of concurrent systems [33, 35, 19]. Use of CSP for verification has been supported by several tools including powerful model-checkers [15, 33, 36].

Interest in using CSP has motivated the introduction of support to model *discrete* time; `tock-CSP` provides an additional event *tock* to record the progress of time. This enables using CSP for modelling time in a format suitable for formal verification. The notation of `tock-CSP` retains that of CSP, thus, automatic verification is supported by existing tools. As a result, `tock-CSP` has been used in the verification of real-time systems such as security protocols [14] and railways systems [21]. Recently, `tock-CSP` has been used for capturing the semantics of RoboChart, a domain-specific language for modelling robotics system [28].

In this work, we add a technique for translating `tock-CSP` into TA that enables using UPPAAL[4] in verifying `tock-CSP` models. UPPAAL is a tool-suite that allows the modelling of *continuous* time systems using a collection of TAs and verifying the systems against temporal logic formulae.

Both temporal logic and refinement are powerful approaches for model checking [26]. The refinement approach models both the system and its specifications with the same notation [33, 35]. Temporal logic enables asking whether a system is a model for a logical formula of the specification ($system \models formula$) [9].

Previously, Lowe has investigated the relationship between the refinement approach (in CSP) and the temporal logic approach [26]. The result shows that, in expressing temporal logic checks using refinement, it is necessary to use the infinite refusal testing model of CSP. The work highlights that capturing the expressive power of temporal logic in specifying the availability of an event (liveness specification) is not possible. Also, due to the difficulty of capturing refusal testing, automatic support becomes problematic. A previous old version of FDR supports refusal testing, but not its recent efficient version [37].

Lowe's work [26] proves that simple refinement checks cannot cope with three operators: *eventually* ($\diamond p$: p will hold in some subsequent state), *until* ($p \mathcal{U} q$: p holds in every state until q hold) and *negation* ($\neg(\diamond p)$: p will never hold in the subsequent states). All these three operators express behaviour that is captured by infinite traces. Our translation work, , presented here will facilitate using the resources of temporal logic (and UPPAAL for automatic support) in checking specifications that are difficult to specify using `tock-CSP` models.

Example 1.1. An Automatic Door System (ADS) opens a door, and after at least one time unit, closes the door in synchronisation with a lighting controller, which turns off the light. In `tock-CSP`, its description is as follows.

```
       ADS = Controller [|{close}|] Lighting
Controller = open -> tock -> close -> Controller
  Lighting = close -> offLight -> Lighting
```

ADS has two components, `Controller` and `Lighting`, which synchronise on the event `close`; this enables `Lighting` to turn off the light after closing the door. For instance, using `tock-CSP`, there is no straightforward way of checking if the system will eventually turn off the light when the environment is ready. However, temporal logic provides a direct construct for specifying these kinds of requirements. For instance, in UPPAAL, the requirement is specified as follow.

- `A<> offLight`
  - - *The system eventually turns off the light, if the environment is ready.*

- `open --> onLight`
  - - *Opening the door leads to turning on the light*

UPPAAL uses a query language, a subset of TCTL, which is based on the notions of path and state [4]. A path formula describes a trace or path, whereas a state formula describes either a location or logical expression. There are five different forms of path formulae that are categorised into, liveness, reachability and safety. Liveness is in the form of either `A<>q` (q is eventually satisfied) or `p --> q` *(a state satisfying p leads to state satisfying q)*. In the literature, $\diamond$ or F are the commonly used symbols for specifying liveness. The reachability formula is in the form of `E<>q` *(a state satisfying q is reachable from the initial state)*. Safety is in the form of either `A[]q` *(q holds in all reachable states)* or `E[]q` *(there is a path where q holds in all states on the path)*.

In verifying the correctness of the translation technique, we follow a series of steps. First, we construct a systematic list of interesting `tock-CSP` processes, which pair all the constructs of `tock-CSP` within the scope of this translation work. Second, we use the developed translation technique and its tool to translate the formulated processes into TA for Uppaal. Third, we use another tool we developed in generating and comparing finite traces of the input `tock-CSP` models and the traces of the translated TA models, with the support of both FDR and Uppaal, respectively. This provides evidence that the translated TA captures the behaviour of the input `tock-CSP` models.

To describe the translation technique, we use Haskell [20], a functional programming language. We use it for expressing, implementing and evaluating the translation technique. The expressive power of Haskell helps us in providing a precise description of the translation technique as a list of translation rules.

The structure of the paper is as follows. Next, Section 2 provides the backgrounds essential for understanding the translation technique. In Section 3, we give an overview of the translation technique. In Section 4, we discuss an evaluation of the translation technique. In Section 5, we highlight relevant related works, and also present a brief comparison with this work. Finally, in Section 6, we highlight future work and also provide a conclusion for the paper.

2. Background

This section discusses the required background for understanding the translation work. We begin by discussing the notations of `tock-CSP` and then discuss the target of the translation work, a network of suitable TA for Uppaal.

2.0.1. tock-CSP

is an extension of CSP, which provides notations for modelling processes and their interactions. In `CSP`, there are constructs for basic processes such as `SKIP` and `STOP`. `SKIP` expresses a successful termination while `STOP` expresses a deadlock behaviour. Also, there are operators such as `prefix (->)`, which describes an event that leads up to a process. For example, the process `move->SKIP` describes a system that moves and then terminates.

Furthermore, there are additional binary operators such as sequential composition (;), which combines two processes serially. For instance, `P3 = P1;P2` behaves as process `P1`, and then after successful termination of `P1`, `P3` behaves as `P2`. There are other binary operators for composing processes in different ways, such as concurrency, choice and interrupt. Also, CSP provides a special event *tau* (τ) that represents invisible actions that are internal to a system, specifically for internal communications of a system that are hidden from the operating environment. The collection of these operators provides a rich set of constructs for modelling untimed systems. Comprehensive full details of CSP constructs, together with examples, are available in these books [33, 35].

Consequently, `tock-CSP` uses a special event `tock` [33]. Each event `tock` specifies a single unit of time. For example, a process

```
Pt= move->tock->tock->turn->SKIP
```

specifies the behaviour of a system that `moves`, and then after at least two time units, `turns` and then terminates.

We provide a BNF for the notation of `tock-CSP` we consider in developing our translation technique and its supporting tool. Part of the BNF is presented in Definition A.1 in Haskell. The BNF is a derivative of the `tock-CSP` BNF, details description is provided in Appendix A.

Definition 2.1. Data definition of `CSPproc`

```
data CSPproc = STOP
             | Stopu
             | SKIP
             | Skipu
             | WAIT        Int
             | Waitu       Int
             | Prefix      Event      CSPproc
             | IntChoice   CSPproc    CSPproc
             | ExtChoice   CSPproc    CSPproc
             | Seq         CSPproc    CSPproc
             | Interleave  CSPproc    CSPproc
             | GenPar      CSPproc    CSPproc [Event]
             | Interrupt   CSPproc    CSPproc
             | Timeout     CSPproc    CSPproc Int
             | Hiding      CSPproc    [Event]
             | Rename      CSPproc    [(Event, Event)]
             | Proc        NamedProc
             | Exception   CSPproc    CSPproc [Event]
             | EDeadline   Event      Int
             | ProcID      String
```

2.0.2. Timed Automata for Uppaal

UPPAAL provides graphical notations for modelling hybrid systems using TA. In UPPAAL, systems are modelled as a network of TA with additional variables. Mathematically, a TA is defined as a tuple (L, l_0, C, A, E, I), where L is a set of locations such that l_0 is the initial location, C is a set of clocks, A is a set of actions, E is a set of edges that connects the locations L, and I is an invariant that is associated with a location $l \in L$. So, an edge E is defined as $E \subseteq (L \times A \times B(C) \times 2^C \times L)$, which describes an edge from location $l \in L$ that is triggered by an action $a \in A$, guarded with a guard $B(C)$, and associated clocks C that are reset on following an edge $e \in E$ to a location $l \in L$. Lastly, I is a function that assigns an invariant to a location $I : L \longrightarrow B(C)$ [4, 6].

A system is modelled as a network of TA that communicate over channel synchronisations or shared variables. A sending channel is decorated with an exclamation mark ($c!$) while the corresponding receiving channel is decorated with a question mark $c?$. To describe behaviour, a TA performs an action $c!$ to communicate with another TA that performs the corresponding co-action $c?$.

For expressing urgency, there are urgent channels that do not allow delay. Broadcast channels facilitate communication among multiple TA, in the form of one-to-many communication (one sender with multiple receivers). In addition, there are urgent and committed types of location. An urgent location does not allow time to pass, whereas a committed location is an urgent location that must participate in the next transition before any other transition. A committed location is useful in expressing atomicity, specifically a compound action that spans multiple transitions that must be executed as a unit. There are also invariants for specifying and enforcing progress [4].

Overall, UPPAAL provides graphical notations for modelling a system as a network of TA that model components that communicate with one another and the explicit operating environment, also described as a TA. For example, the collection of TA in Figure 1 describes a system and its operating environment, which is explained later in details. For specifications, UPPAAL uses a subset of Timed Computational Tree Logic (TCTL) [5] for specifying system properties that can be verified automatically [25, 24, 4].

3. An Overview of the Translation Technique

Our translation technique produces a list of small TA. The occurrence of each `tock-CSP` event is captured in a small TA with an UPPALL's action, which records an occurrence of the translated event. The action has the same name as the translated event. The technique composes these small TA into a network of TA, which captures the behaviour of the input `tock-CSP` model.

For example, a translation of the process ADS, from Example 1.1, produces the network of small TA in Figure 1. Details of the translated TA are as follows. Starting from the top-left corner, the first TA captures concurrency by starting the two concurrent automata corresponding to the processes `Controller` and `Lighting` in two possible

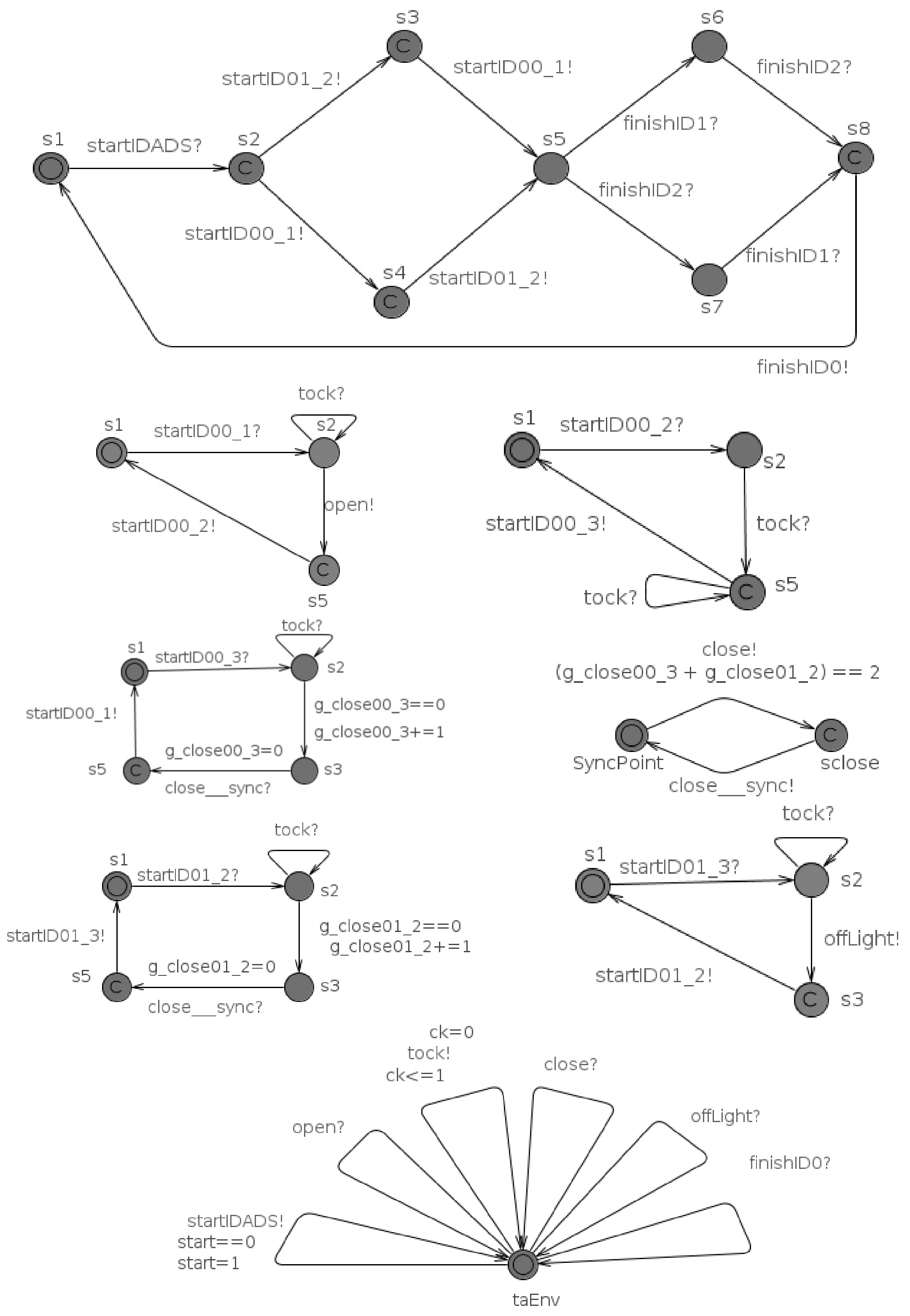


Figure 1: A list of networked TA for the translation of the process ADS

orders, either `Controller` then `Lighting` or vice versa, depending on the choice of the operating environment. Afterwards, it also waits on state `s5` for their termination actions in the two possible orders, either `Controller` then `Lighting` or vice versa, depending on the process that terminates first. The whole system terminates with the action `finishID0`.

The second, third and fourth TA capture the translation of the process `Controller` . The second TA captures the occurrence of the event `open`. The third TA captures the occurrence of the event `tock?` (with a question mark for synchronising with the environment in recording the progress of time. The fourth TA captures the occurrence of the event `close`, which synchronises with the synchronising controller, the fifth TA.

The sixth and seventh TA capture the translation of the process `Lighting`. The sixth TA captures the translation of `close`, which also needs to synchronise with the synchronisation controller (fifth TA). The seventh TA captures the event `offLight`. Finally, the last TA is an environment TA that has co-actions for all the translated events. Also, the environment TA serves the purpose of 'closing' the overall system as required for the model checker. In this last TA, we use the variable *start* to construct a guard $start == 0$ that blocks the environment from restarting the system. This concludes the description of the list of TA for ADS.

An alternative to using a list of small TA is using a single large TA that captures the behaviour of the translated `tock-CSP` process. The main reason for using a list of small TA is coping with the compositional structure of `tock-CSP`, which is not available in TA [12]. For instance, it can be argued that a linear process constructed with a series of prefix operators can be translated into a linear TA. However, the compositional structure of `tock-CSP` is not suitable for this straightforward translation. A network of small TA provides enough flexibility for composing TA in various ways to capture the behaviour of the original `tock-CSP` process, as explained below.

In constructing the network of TA, we use additional ***coordinating actions*** that link the small TA into a network. The coordinating actions link the small TA into a network of TA to establish the flow of the translated `tock-CSP` process. For instance, the channel `startIDADS` forms a link that connects the environment TA (last TA in Figure 1.1) with the first TA, on performing the action `startIDADS!` and its co-action `startIDADS?` A precise definition of the coordinating actions is provided below in Definition 3.1.

Definition 3.1. A Coordinating Action is an UPPAAL action that does not correspond to a `tock-CSP` event. There are six types of coordinating actions as follows:

1. **Flow actions** coordinate a link between two TAs for capturing the flow of their behaviour;

2. **Terminating actions** record termination information, in addition to coordinating a link between two TAs;

3. **Synchronisation actions** coordinate a link between a TA that participate in a synchronisation action and a TA for controlling the multi-synchronisation;

4. **External choice actions** coordinate an external choice, such that choosing one of the TA that is part of the external choice blocks the other alternative choices TAs;

5. **Interrupting actions** initiate an interrupting transition that enables a TA to interrupt another; and **Exception actions** coordinate a link between a TA that raises an exception and a control TA that handles the exception.

The names of each coordinated action are unique for ensuring the correct flow of the translated TA. In our implementation, the name of the flow actions are generated in the form `startIDx`, where `x` is either a natural number or the name of the input `tock-CSP` process. This can be seen in Example 1.1; `startID00_1` is the flow action that connects the first TA with the second TA.

Likewise, the names of the remaining coordinating actions follow the same pattern: `keywordIDx`, where `keyword` is a designated word for each of the coordinating actions; `finish` for a terminating action, `ext` for an external choice action, `intrp` for an interrupting action, and `excp` for an exception action. Similarly, the name of a synchronising action is in the form `eventName___sync`: an event name appended with the keyword `___sync` to differentiate a synchronising action from other actions, especially for analysis.

We use termination actions for cases where a TA needs to communicate a successful termination for another TA to proceed. For example, in the case of the translation of sequential composition `P1;P2` where the process `P2` begins only after successful termination of the process `P1`.

For each translated `tock-CSP` specification, we provide an environment TA, like the last TA in Figure 1.1. The environment TA has corresponding co-actions for all the translated events of the input `tock-CSP` model. Also, the environment TA has two coordinating actions that link the environment TA with the network of the translated TA. First, a flow action links the environment with the first TA in the list of the translated TA. This first flow action is the starting action that activates the behaviour of the translated TA. Second, a terminating action links back the final TA to the environment TA, and records a successful termination of a process. A precise definition of the structure of environment TA is below.

Definition 3.2. An Environment TA models an explicit environment for UPPAAL model. The environment TA has one state and transitions for each co-action of all the events in the input `tock-CSP` process, in addition to two transitions for the first starting flow action and the final termination co-action, as well as the action `tock!` for indicating the progress of time.

In translating multi-synchronisation, we adopt a centralised approach developed in [31] and implemented using Java in [11]. The approach described uses a separate centralised controller for controlling multi-synchronisation events. Here, we use a separate TA with UPPAAL broadcast channel to communicate synchronising information among synchronising TA and a control TA.

An example can be seen in Figure 1.1, where we illustrate the strategy in translating `close`, which synchronises the two processes `Controller` and `Lighting`. In Figure 1, the fifth TA captures the translation of `close`, which synchronises the fourth and sixth TA for those processes. The fifth TA is a synchronising TA; and its uses the broadcast channel `close___sync` to synchronise the fourth and sixth TAs.

Each synchronising TA has a guard for ensuring that the TA synchronises with the required number of TAs. The guard is a logical expression that combines variables from all the TAs that synchronise on the synchronisation action. Each TA updates its variable from 0 to 1 to indicate its readiness for the synchronisation. For instance in Figure 1, the fifth TA is a synchronising TA that has a guard expression `(g_close00_3 + g_close01_2)==2`, which becomes true when the fourth and sixth TA update their respective variables: `g_close00_3` and `g_close01_2`, from 0 to 1. Then the fifth TA notifies the occurrence of the action `close` and broadcasts the synchronising action `close___sync!`. A definition of synchronisation TA is provided in Definition A.2.11.

Definition 3.3. A synchronisation TA coordinates a synchronisation actions. Each synchronisation TA has an initial state and a committed state for each synchronisation action, such that each committed state is connected to the initial state with two transitions. The first transition from the initial state has a guard and an action. The guard is enabled when all the processes are ready for the synchronisation, which also enables the TA to perform the associated action that notifies the environment of its occurrence. In the second transition, the TA broadcasts the synchronisation action to all the processes that synchronise on the synchronisation action.

In translating external choice, we provide additional transitions that enables the behaviour of the chosen process to block the behaviour of the other processes. Initially, in the translated TA, all the initials [1] of the translated processes are available such that choosing one of the processes blocks the other alternative.

[1]Initials are the first visible events of a process.

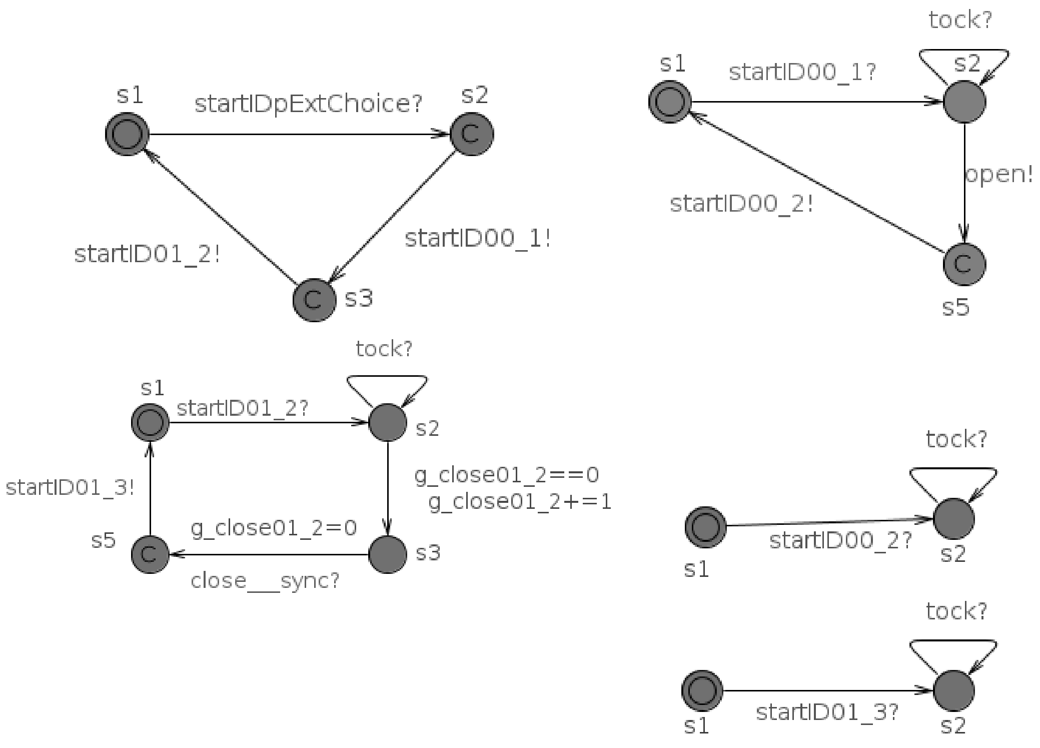


Figure 2: A list of TA for the translated behaviour of the process
`Pe = (right->STOP)[](left->SKIP)`

An example of translating external choice is provided in Figure 2 for the process `Pe = (left->STOP)[](right->STOP))`, which composes two processes `(left->STOP)` and `(right->STOP)` with the operator of external choice.

In Figure 2, starting from the top-left, the first TA is a translation of the operator external choice. The second and third TAs are translations of the LHS process `left->SKIP`. The fourth and fifth TAs are translations of the RHS process `(right->SKIP)`. The first TA has three transitions labelled with actions: `startIDp0_6?`, `startID00_1!` and `startId01_2!`. The TA begins with the first flow action `startIDp0_6?`, then starts the TAs for both the left and right process using `startID00_1!` and `startId01_2!`.

The second TA is a translation the event `left`. Initially, the TA synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has 3 possible transitions labelled with the actions: `left_exch?`, `right_exch!` and `tock?`. With the co-action `tock?`, the TA records the progress of time and remains on the same location `s2`. With the co-action `right_exch?`, the TA performs an external choice co-action for blocking the TA of the LHS process when the environment chooses the right process, and the TA returns to its initial location `s1`. Lastly, the TA performs the action `left_exch!` when the environment chooses the LHS process, and the TA progress to location `s3` to perform the chosen action `left` that leads to location `s5` for performing the flow action `startID00_2`, which activates the third TA for the subsequent process

`STOP`. This describes the behaviour of the LHS process `left->STOP`.

The fourth TA is a translation of `right`, similar to the previous translation of `left` in the second TA. The fifth TA is a translation of the process `STOP`. The omitted environment TA is similar to the last TA in Figure 1.

Similarly, in `tock-CSP`, a process can be interrupted by another process when they are composed with an interrupting operator (/\). For capturing interrupting behaviour in TA, we provide additional transitions for expressing an interrupting action, which enables a TA to interrupt another one. The translation of an interrupting process resembles the translation of an external choice.

An example of translating interrupt is provided in Figure 3, in translating the process `Pi = (open->SKIP)/\(fire->close->SKIP)`. For the process `Pi`, the RHS process `(fire -> close -> SKIP)` can interrupt the behaviour of `(open->SKIP)` at any stable state. In the translated behaviour of the LHS process, we provide additional interrupting actions (`fire_intrpt`) that enable the translated behaviour of the RHS process to interrupt the LHS process. The interrupting actions are provided only for the initials of the RHS process (`fire`).

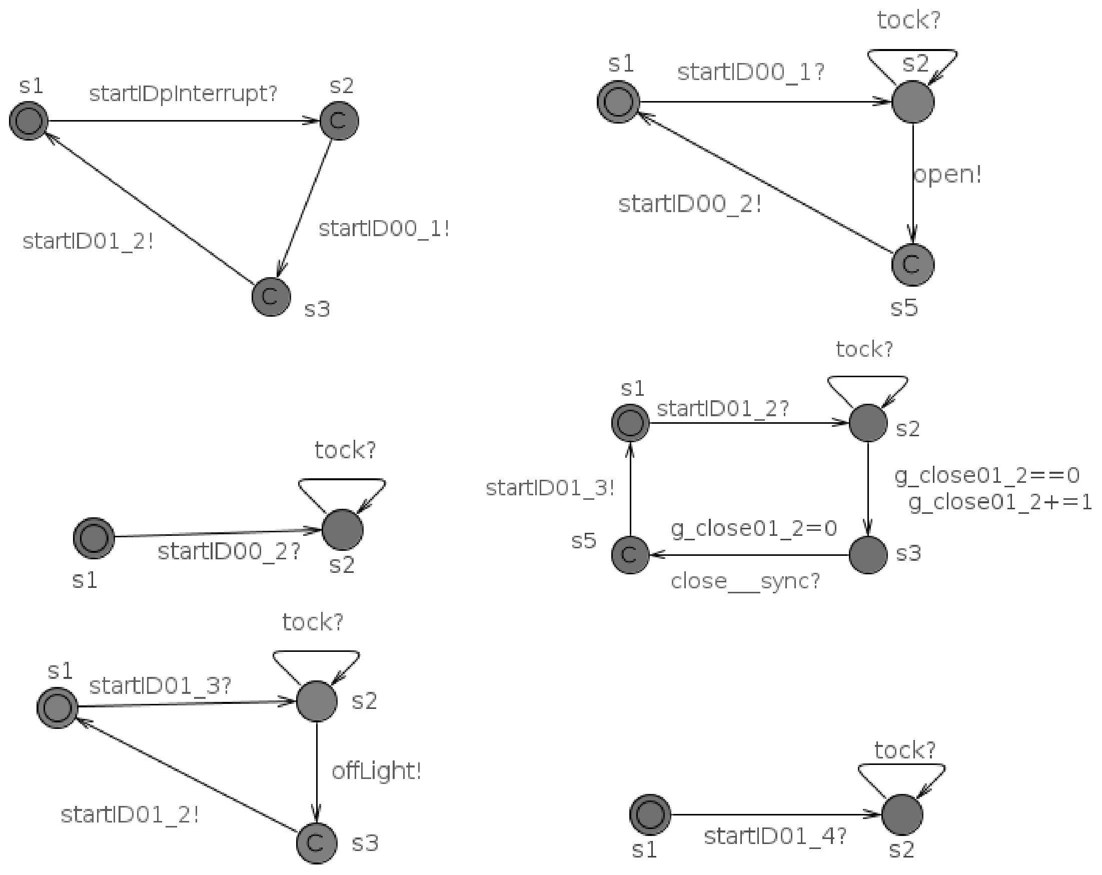


Figure 3: A list of TA for the translated behaviour of the process
`Pi = (open->SKIP)/\(fire->close->SKIP)`

In Figure 3, starting from the top-left, the first TA is a translation of the operator interrupt. The second and third TAs are the translation of the LHS process `open->SKIP`. The fourth, fifth and sixth TAs are translation of the RHS process `fire->close->SKIP`. The omitted environment TA is similar to the last TA in Figure 1.

The first TA starts the TA for both processes using `startID00_1!` and `startID01_2!`, respectively. The second TA synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has 3 possible transitions for the actions: `tock?`, `open!` and `fire_intrpt?`. With the co-action `tock?`, the TA records the progress of time and remains on the same location `s2`. With the co-action `fire_intrpt?`, the TA is interrupted by the RHS, and it returns to its initial location `s1`. With the action `open!`, the TA progresses to location `s5` to perform the flow action `startID00_2`, which activates the third TA for the subsequent process `STOP`.

The third TA synchronises on the flow action `startID00_2` and moves to location `s2`, where it either performs the action `tock?` to record the progress of time or is interrupted with the co-action `fire_intrpt?`, and returns to its initial location `s1`. This completes the behaviour of the process `open->SKIP`.

The fourth TA is a translation of the event `fire`. The TA begins with synchronising on the flow action `startID01_2`, which progress by interrupting the LHS process using the interrupting flow action `fire_intrpt`, then `fire`, and proceeds to `startID01_3` for starting the fifth TA. The fifth TA synchronises on the flow action and moves to location `s2`, where it either performs the action `tock?` for the progress of time and remains in the same location or performs the action `close`, and proceeds to location `s5` then performs the flow action `startID01_4` for starting the sixth TA for the translation of `STOP` (deadlock).

Also, in `tock-CSP`, an event can be renamed or hidden from the environment. In translating renaming, our technique uses a list of renamed events; before translating each event, we check if the event is renamed, and then translate the event appropriately with the corresponding new name. In the like manner, if an event is part of the hidden events, using a list of hidden events, the technique uses a special name *itau* in place of the hidden event. The event *itau* has similar meaning to *tau*, which represents a hidden event in CSP. For the purpose of analysis, we differentiate the translated event *itau* from *tau*.

We translate the event `tock` into a corresponding action `tock` for recording the progress of time in the translated TA using a broadcast channel that enables the translated TA to synchronise in recording the progress of time. The action `tock` is broadcast by the environment TA, which causes all the processes to synchronise and record the progress of time using their corresponding co-actions `tock?` (with a question mark). In particular, in Figure 1 the last TA that models the environment, has a transition with an action `tock` that is guarded with the expression $ck \leq 1$, so that the action `tock` happens every 1 time unit, and resets the clock $ck = 0$ to zero on following the transition.

Silent transitions enable a TA to take any of the silent transitions. We translate non-deterministic choice into silent transitions, such that the translated TA follow one of the silent transitions non-deterministically. Nevertheless, divergence is not part of the construct that we consider in this translation work because divergence is a sign of bad behaviour. This completes an overview of the strategy we follow in developing the translation technique.

An example of the formalisation of the translation rules in Haskell is provided in Rule 1, which describes the translation of the construct `STOP`. All the remaining rules can be found in [1]. Rule A.1 produces an output TA that is depicted in Figure 9, which has two locations and two transitions as defined in Rule A.1, Lines 7–9 and Lines 11–13, respectively. In Figure 9, we map the structure of the TA with the names used in the translation rule. The names are `loc1, loc2, tran1, tran2, intrpt`.

Rule 3.1. Translation of STOP

```
1  transTA STOP processName bid sid _ usedNames =
2    (([(TA idTA [] [] locs [] (Init loc1) trans)]), [] )
3    where
4     idTA = "taSTOP__" ++ bid ++ show sid
5
6       --   Definition of locations in the TA
7       -- = Location ID     Name  Label        LocType
8     loc1 = Location "id1"  "s0"  EmptyLabel  None
9     loc2 = Location "id2"  "s1"  EmptyLabel  None
10    locs = [loc1, loc2]
11
12      --    Definition of transitions in the TA
13      --  = Transition Source   Target  [Label] [Edge]
14    tran1 = Transition loc1     loc2    [lab1]  []
15    tran2 = Transition loc2     loc2    [lab2]  []
16    intrp = transIntrpt intrptsInits loc1 loc2
17    trans = [tran1, tran2] ++ intrp
18
19    --  Labels of the transitions in the TA
20    lab1 = Sync (VariableID
21             (startEvent processName (bid ++ sid)) [])
22              Ques
23    lab2 = Sync (VariableID "tock" []) Ques
24
25    -- Get the initial events of an interrupting process
26    (_, _, _, _, _, intrptsInits, _, _) = usedNames
27
```

A detailed description of Rule A.1 is as follows. Each rule considers one construct. Line 1 identifies that this a definition of our translation function `transTA` for `STOP`, and

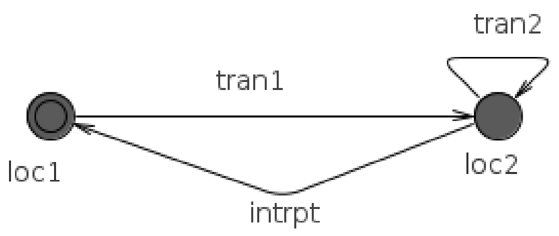


Figure 4: A structure of an output TA for the translation of STOP from Rule 1.

it takes three extra parameters: `processName`, `bid` and `sid`. The symbol underscore [2] represents unused arguments.

Line 2 defines the output pairs, a list of output TA and a list of synchronising actions. The second element is empty ([]) in Rule 1 because there is no synchronising action in translating the construct `STOP`. The first element describes the output TA, which has 6 parameters. First, `idTA` is an identifier for the TA, which is defined subsequently in Line 4, as the concatenation of the keyword `"taSTOP__"` with the 2nd and 3rd arguments of the function `transTA`, that is `bid` and `sid` respectively. Additionally, still in Line 2, in the definition of the output TA, the 2nd, 3rd and 5th parameters are empty for the output TA. While the 4th parameter `locs` is a list of locations for the output TA, as defined in Lines 7–9. The 6th parameter `(Init loc1)` specifies `loc1` as the initial location of the output TA. Last parameter `trans` describes a list of transitions that connect the locations of the TA, as defined in line 12–13. Line 14 defines interrupting transitions that are generated with the function `transIntrpt`, that is in the case where a process is composed with an interrupting process. Lines 17–20 define the labels of the transitions. Lastly, Line 23 extracts the initials of an interrupting process from the parameter `usedNames`, which collects information such as the names of hidden events and renamed events.

The behaviour of the output TA begins with the first flow action (defined in Line 17). After that, the TA performs the action `tock` (line 18), repeatedly, which allows the progress of time, unless there is an interrupt, which causes the TA to follow the interrupting transition (defined in Line 14).

Example 3.1. An example of translating a process `STOP` that produces a list of TA, which contains two TA as shown in Figure 5. The LHS TA captures the translation of the construct *STOP*, in this case there is no interrupting transition because there is no interrupting process. A translation of interrupt has been provided separately. The RHS TA is an environment TA that we provide for translating each process, as described in Definition 3.2.

```
transTA(STOP) = [
```

[2]In Haskell, the symbol of underscore indicates a position of unused arguments. For conciseness, we use the underscore to omit unused arguments and provide only the required arguments for each translation rule.

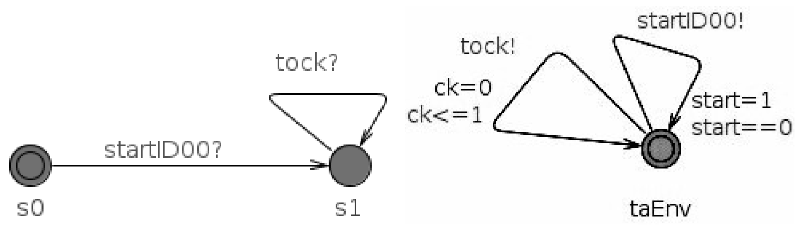


Figure 5: A list of TA for the translation of the construct STOP

]

4. Evaluation

A sound translation ensures that properties of the source model are preserved in the targeted translated models. This is determined by comparing the behaviours of the source model and the translated model [27, 29, 22, 3].

4.1. Experimental Evaluation

In this work, we use traces to compare the behaviour of the input `tock-CSP` models and the traces of the translated TA models. We have developed an evaluation tool, which uses the translation tool to translate `tock-CSP` model into TA, and then uses both FDR and Uppaal as black boxes in generating sets of finite traces. The structure of the evaluation tool is shown in Figure 6.

In generating traces, like most model checkers, FDR produces only one trace (counter-example) at a time. So, based on the testing technique in [30], we have developed a trace-generation technique that repeatedly invokes FDR for generating traces until we get all the required traces set of the input `tock-CSP`. Similarly, based on another testing technique [25], we have developed another trace-generation technique, which uses Uppaal in generating finite discrete traces for the translated TA models.

These two trace-generation techniques are components of our trace analysis tool, which has two stages of analysing traces. In the first stage, we generate traces of the input `tock-CSP` and its corresponding translated TA, using both FDR and Uppaal, respectively. Then, we compare the generated traces, and if they do not match, we move to the second stage for further analysis. In FDR, the trace-generation technique is capable of generating and detecting traces with different permutations of events. In contrast, Uppaal is not capable of detecting traces with different permutations because Uppaal uses a logical formula in generating traces [25, 4]. Thus, we use the power of FDR to complement Uppaal in generating and comparing traces of `tock-CSP` and TA.

Basically, Uppaal was developed for checking if a system satisfies a logical formula for a requirement specifications, irrespective of the behaviour of the system. For example, Uppaal does not distinguish between the two traces $\langle e1, e2, e3\rangle$ and $\langle e1, e3, e2\rangle$, if both of them satisfy the specification formula. The two traces contain the same three

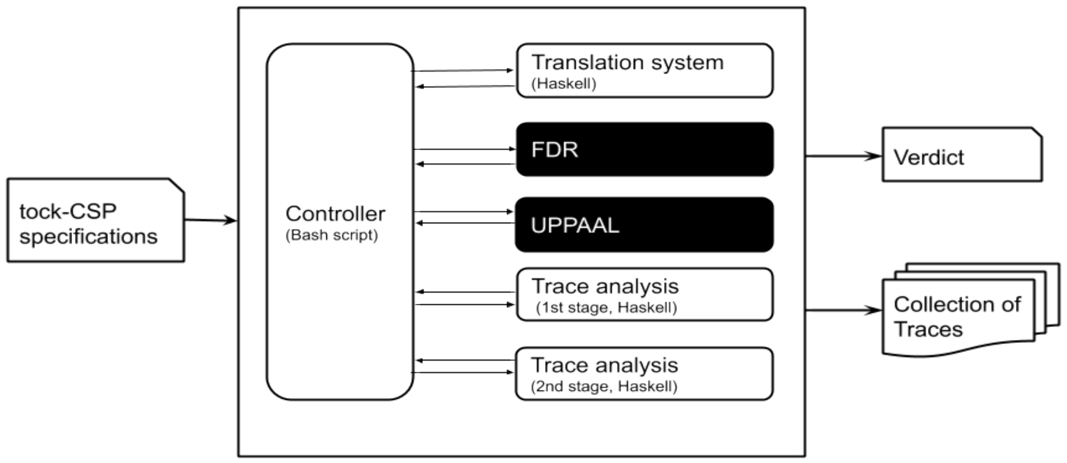


Figure 6: Structure of the trace analysis system

No.	System	States	Transitions	Events
1	Thermostat Machine	7	16	5
2	Bookshop Payment System	7	32	9
3	Simple ATM	15	33	15
4	AutoBarrier system	35	84	10
5	Rail Crossing System	80	361	12

Table 1: An overview of the case studies

actions, but the action $e2$ and $e3$ occur in a different order. However, FDR is capable of detecting and generating both traces. Thus, in the second stage, we use UPPAAL to check if the traces of FDR are acceptable traces of the translated TA.

We use this tool in evaluating the translation technique by translating a list of systematically formulated `tock-CSP` processes that pair the constructs of `tock-CSP`, The list contains 111 processes that are not more than five states in size. Archives of the processes and their traces are available in a repository [1].

In addition, the translation technique has been tested in various larger examples such as an Automated Barrier to a car park [35], a Thermostat machine for monitoring ambient temperature [35], an Automated Teller Machine (ATM) [34], a Bookshop Payment System [35], and a Railway Crossing System [33]. An overview of the processes is provided in Table 1, while details specifications is provided in the appendix.

Trace analysis is an approximation to establishing correctness with a finite set of traces. Proving correctness for the complete set of traces involves using mathematical proof. This is achieved using structural induction. An account of our initial effort to produce a proof is provided next.

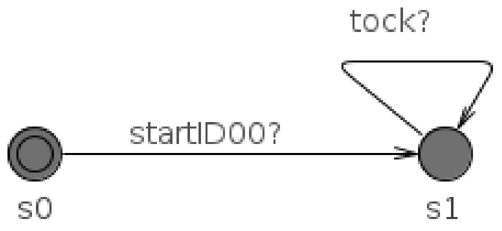
4.2. An Initial Mathematical Proof

This sections discusses our plan for using mathematical proofs in justifying the translation technique. So far, we use trace analysis for evaluating the translation technique. However, trace analysis is an approximation to establishing correctness with a finite set of traces. Proving correctness for the complete set of traces involves using mathematical proof. This is achieved using structural induction. An account of our initial effort to produce a proof is provided in this section. We illustrate an early part of the proof using a translation of the constructs *STOP* and *SKIP* as base cases of the structural induction, and using a Internal choice as for illustrating an induction step of the structural induction.

We begin with describing three helping functions that we use in the proof. The functions are $trace'_{TA}(TA)$, $trace_{TA}(TA)$ and $traces_{tock-CSP}(Proc)$. Here we describe the types of the functions $trace'_{TA}(TA)$ and $traces_{tock-CSP}(Proc)$ and definition of the function $trace_{TA}(TA)$ in term of the first function $trace'_{TA}(TA)$. The first function $trace'_{TA}(TA)$ takes a list of TA and returns its traces including the flow actions. The second function $trace_{TA}(TA)$ takes a list of TA and returns its traces without the flow actions. Lastly, the function $traces_{tock-CSP}(Proc)$ takes a `tock-CSP` process and returns its traces, as defined by Roscoe [33].

For example, let consider `TA1` as the TA for the translation of *STOP* using Rule A.1 (Page 36) as follows.

$TA1 =$



]

$$\therefore TA1 = (\{s0, s1\}, s0, \{ck\}, \{startID00, tock\}, \{(s0, startID, \emptyset, \emptyset, s1),$$
$$(s1, tock, ck \leq 1, ck, s1)\}\{\}) \quad (1)$$

In the context of TA, a path [2, 6] is a sequence of consecutive transitions that begins from the initial state, which may possibly be an empty sequence. And a trace [2, 6] (or word in the language of TA) is a sequence of actions in a given path. There is only one infinite path in TA1; the first transition from location $s0$ to location $s1$ and the second transition from location $s1$ and return to location $s1$, and repeated infinitely. The traces on the path are as follows.

$$traces'_{TA}(TA1) = \{\langle\rangle\} \cup \{\langle startID00\rangle \frown \langle tock\rangle^n \mid n \in \mathbb{N}\} \quad (2)$$

The function $trace'_{TA}(TA)$ describes the traces of TA1 as follows. The first empty sequence happens before the first transition. The action `startID00` happens on the first transition. The action `tock` happens on the second transition, which is repeated infinitely to produce the infinite traces $\langle tock \rangle^n$. The second function $trace_{TA}(TA)$ is similar to $traces'_{TA}(TA)$ but removes all the coordinating actions (Definition 3.1) from the traces.

$$traces_{TA}(TA1) = \{t \setminus CoordinatingActions \mid t \in traces'_{TA}(TA1)\} \tag{3}$$

For the proof of our translation function, we need to establish that, for each valid `tock-CSP` process `CSPproc`, within the provided BNF (Section A.1, Page 28), the following equation holds.

$$\forall P : CSPproc.traces_{TA}(transTA(CSPproc)) = traces_{tock-CSP}(CSPproc) \tag{4}$$

Therefore, for each translation rule (Section A.2), we are going to show that the translated TA captures the behaviour of its translated construct of `tock-CSP` in the BNF (Section A.2) in the following sections.

4.2.1. Proof for the Construct STOP *(Base case)*

Starting with the first rule for translating the construct of a basic process `STOP`; the first construct for the provided BNF. Thus, for *STOP*, we are going to show that:

$$traces_{TA}(transTA(STOP)) = traces_{tock-CSP}(STOP) \tag{5}$$

We will continue using the concept of Haskell by providing only the required parameters for the function and represent the remaining parameters with an underscore.

$traces_{TA}($ $transTA\ STOP\ processName\ bid\ sid\ \_\ \_)$

$=$ [Translate the process *STOP* (Rule A.1 (Page 36), into mathematical definition of TA (Equation 1, Page 19]

$traces_{TA}([(\{s0, s1\}, s0, \{ck\}, \{startIDbid\_sid, tock\}, \{(s0, startIDbid\_sid, \emptyset, \emptyset, s1),$
$(s1, tock, ck \leq 1, ck, s1)\}, \{\})])$

$=$ [Generate traces using the second function $traces_{TA}(TA)$ (Equation 2, Page 19)]

$\{t \setminus CoordinatingActions \mid t \in traces'_{TA}([\{s0, s1\}, s0, \{ck\}, \{startIDbid\_sid, tock\},$
$\{(s0, startID\_bid\_sid, \emptyset, \emptyset, s1), (s1, tock, ck \leq 1, ck, s1)\}, \{\}])\}$

$=$ [Generate traces using the first function $traces'_{TA}(TA)$ (Equation 3, Page 20)]

$\{t \setminus CoordinatingActions \mid t \in (\{\langle\rangle\} \cup \{\langle startIDbid\_sid \rangle \frown \langle tock \rangle^n \mid n \in \mathbb{N}\})\}$

$=$ [Remove the flow actions from the traces]

$\{\langle tock \rangle^n \mid n \in \mathbb{N}\}$

$=$ [Traces of tock-CSP from Roscoe [33] (Section 14.2)]

$traces_{tock-CSP}(STOP)$

$$\therefore\ traces_{TA}(transTA(STOP)) = traces_{tock-CSP}(STOP)$$

This proves that the traces of the translated TA for $STOP$ captures its traces correctly.

4.2.2. Proof for the Construct SKIP (*Base case*)

For this section, we are going to show that

$$traces_{TA}(transTA(SKIP)) = traces_{tock-CSP}(SKIP) \qquad (6)$$

$traces_{TA}(\ transTA\ SKIP\ processName\ bid\ sid\ fid\ \_)$

$=$ [Translate the construct SKIP (Rule A.3 (Page 41), into mathematical definition of TA (Equation 1, Page 19)]

$traces_{TA}([\{s0, s1, s2\}, s0, \{ck\}, \{startIDbid\_sid, tock, tick, finishfid\},$
$\{(s0, startIDbid\_sid, \emptyset, \emptyset, s1), (s1, tock, ck \leq 1, ck, s1), (s1, tick, \emptyset, \emptyset, s2),$
$(s2, finishfid, \emptyset, \emptyset, s0)\}, \{\}]$

$=$ [Generate traces using the second function $traces_{TA}(TA)$ (Equation 2, Page 19)]

$\{t \setminus CoordinatingActions \mid t \in traces'_{TA}([\{s0, s1, s2\}, s0, \{ck\},$
$\{startIDbid\_sid, tock, tick, finishfid\}, \{(s0, startIDbid\_sid, \emptyset, \emptyset, s1),$
$(s1, tock, ck \leq 1, ck, s1), (s1, tick, \emptyset, \emptyset, s2),$
$(s2, finishfid, \emptyset, \emptyset, s0)\}, \{\}])$

$=$ [Generate traces using the first function $traces'_{TA}(TA)$ (Equation 3, Page 20)]

$\{t \setminus CoordinatingActions \mid t \in (\{\langle\rangle, \langle startIDbid\_sid\rangle\}$

$\cup \{\langle startIDbid\_sid\rangle \frown \langle tock\rangle^n \frown \langle tick^m\rangle \mid n \in \mathbb{N}, m \in \{0,1\}\})\}$

$\cup \{\langle startIDbid\_sid\rangle \frown \langle tock\rangle^n \frown \langle tick\rangle \frown \langle finishfid\rangle \mid n \in \mathbb{N}\})\}$

$=$ [Remove the flow actions from the traces]

$\{\langle tock\rangle^n \frown \langle tick^m\rangle \mid n \in \mathbb{N}, m \in \{0,1\}\}$

$=$ [Traces of tock-CSP from Roscoe [33] (Section 14.2)

$traces_{tock-CSP}(SKIP)$

$$\therefore\ traces_{TA}(transTA(SKIP)) = traces_{tock-CSP}(SKIP)$$

This proves that, for the construct $SKIP$, the traces of the translated TA captures the traces of the process $SKIP$ correctly.

4.2.3. Proof for the Construct Internal Choice $(Inductive\ step)$

In this Section, we show that the translation of the internal choice captures the traces correctly by proving the following equation.

$$traces_{TA}(transTA(IntChoice\ P1\ P2)) = traces_{tock-CSP}(IntChoice\ P1\ P2) \qquad (7)$$

Here, we are going to proof the translation of internal choice, assuming that the proof holds for the individual processes $P1$ and $P2$ as follows.

$$traces_{TA}(transTA(P1)) = traces_{tock-CSP}(P1) \qquad (8)$$

$$traces_{TA}(transTA(P2)) = traces_{tock-CSP}(P2) \qquad (9)$$

Starting from LHS of Equation 7, the proof goes as follows.

$traces_{TA}\ (transTA\ (IntChoice\ P1\ P2)\ processName\ bid\ sid\ fid\ usedNames)$

$=$ [Translate the construct Internal choice (Rule A.8 (Page 59), into mathematical definition of TA (Equation 1, Page 19)]

$traces_{TA}([\{s0, s1, s2, s3\}, s0, \{\}, \{startIDbid\_sid, startID(bid+0)\_(sid+1),$
$startID(bid+1)\_(sid+2)\}, \{(s0, startIDbid\_sid, \emptyset, \emptyset, s1), (s1, \emptyset, \emptyset, \emptyset, s2), (s1, \emptyset, \emptyset, \emptyset, s3),$
$(s3, startID(bid+0)\_(sid+1), \emptyset, \emptyset, s0), (s2, startID(bid+1)\_(sid+2), \emptyset, \emptyset, s0)\}\{\}])$
$\cup\, traces_{TA}\ (transTA\ P1\ processName\ (bid+0)\ (sid+1)\ fid\ usedNames)$
$\cup\, traces_{TA}\ (transTA\ P2\ processName\ (bid+1)\ (sid+2)\ fid\ usedNames)$

$=$ [Reversing the function $transTA()$ for the processes $P1$ and $P2$]

$traces_{TA}([\{s0, s1, s2, s3\}, s0, \{\}, \{startIDbid\_sid, startID(bid+0)\_(sid+1),$
$startID(bid+1)\_(sid+2)\}, \{(s0, startIDbid\_sid, \emptyset, \emptyset, s1), (s1, \emptyset, \emptyset, \emptyset, s2), (s1, \emptyset, \emptyset, \emptyset, s3),$
$(s3, startID(bid+0)\_(sid+1), \emptyset, \emptyset, s0), (s2, startID(bid+1)\_(sid+2), \emptyset, \emptyset, s0)\}\{\}])$
$\cup\, traces_{TA}(transTA(P1)) \cup traces_{TA}(transTA(P2))$

$=$ [Induction, assuming the traces are correct for processes $P1$ and $P2$, Equations 8 and 9, respectively.]

$traces_{TA}([\{s0, s1, s2, s3\}, s0, \{\}, \{startIDbid\_sid, startID(bid+0)\_(sid+1),$
$startID(bid+1)\_(sid+2)\}, \{(s0, startIDbid\_sid, \emptyset, \emptyset, s1), (s1, \emptyset, \emptyset, \emptyset, s2), (s1, \emptyset, \emptyset, \emptyset, s3),$
$(s3, startID(bid+0)\_(sid+1), \emptyset, \emptyset, s0), (s2, startID(bid+1)\_(sid+2), \emptyset, \emptyset, s0)\}\{\}])$
$traces_{tock-CSP}(P1) \cup traces_{tock-CSP}(P2)$

$= $ [Generate traces using the second function $traces_{TA}(TA)$ (Equation 2, Page 19)]
$\{t \setminus CoordinatingActions \mid t \in traces'_{TA}([\{s0, s1, s2, s3\}, s0, \{\},$
$\{startIDbid\_sid, startID(bid+1)\_(sid+1), startID(bid+2)\_(sid+2)\},$
$\{(s0, startIDbid\_sid, \emptyset, \emptyset, s1), (s1, \emptyset, \emptyset, \emptyset, s2), (s1, \emptyset, \emptyset, \emptyset, s3),$
$(s3, startID(bid+0)\_(sid+1), \emptyset, \emptyset, s0), (s2, startID(bid+1)\_(sid+2), \emptyset, \emptyset, s0)\}\{\}])$
$traces_{tock-CSP}(P1) \cup traces_{tock-CSP}(P2)$
$= $ [Generate traces using the first function $traces'_{TA}(TA)$ (Equation 3, Page 20)]
$\{t \setminus CoordinatingActions \mid t \in (\{\langle\rangle, \langle startIDbid\_sid\rangle\}$
$traces_{tock-CSP}(P1) \cup traces_{tock-CSP}(P2)$
$= $ [Remove the flow actions from the traces]
$traces_{tock-CSP}(P1) \cup traces_{tock-CSP}(P2)$
$= $ [Traces of $tock - CSP$ from Roscoe [33] (Section 14.2)]
$traces_{tock-CSP}(IntChoice\ P1\ P2)$

$$\therefore traces_{TA}(transTA(IntChoice\ P1\ P2)) = traces_{tock-CSP}(IntChoice\ P1\ P2)$$

This proves that, for the construct Internal Choice, the traces of the translated TA for internal choice captures traces of the internal choice correctly.

So far, the proof covers the first three rules, $STOP$, $SKIP$ and Internal choice. We plan to proof all the remaining rules to establish the correctness of the translation rules that will cover infinite traces.

5. Related Work

Timed-CSP [35] is another popular extension of CSP that provides additional notations for capturing temporal specifications. Unlike `tock-CSP`, Timed-CSP records the progress of time with a series of positive real numbers, which facilitates reasoning in a format that is suitable for verifying real-time systems.

However, the approach of Timed-CSP can not be used to specify strict progress of time, neither deadline nor urgency. Additionally, traces of Timed-CSP become infinite, which is problematic for automatic analysis and verification [33]. Besides, there is no tool support for verifying Timed-CSP models.

As a result of that, many researchers explore model transformations in supporting Timed-CSP with an automated verification tool. Thus, Timed-CSP has been translated into `tock-CSP` for using FDR in automatic verification [32].

Also, Timed-CSP has been translated into Uppaal. The work was initiated in [12] and then subsequently improved in [16]. Additionally, Timed-CSP has been translated into Constraint Logic Programming (CLP) that facilitates reasoning with the support of the constraint solver CLP(R) [13].

However, there is less attention in applying the same transformation techniques in improving `tock-CSP`. An attempt for transforming TA into `tock-CSP` was proposed in [23]. Whereas in this work, we consider the opposite direction.

Apart from CSP and TA, model transformations have been used for improving various formal modelling notations. For instance, Circus has been translated into `CSP||B` to enable using the tool ProB for automatic verification [38]. Additionally, B has been translated into TLA+ for automatic validation with TLC [18]. Also, translating TLA+ to B has been investigated for automated validation of TLA+ with ProB [17], such that both B and TLA+ benefit from the resources of each other, specifically their supporting tools ProB and TLC, respectively, for automated verification.

Model transformation is an established field, which has been used for many decades in addressing computational problems. The novelty of this translation work is traced back to early days of model translation works, such as the translation of the timed variants of LOTOS into TA [10, 7]. There is a recent systematic survey of model translation, which provides a rich collection of model transformations techniques and their supporting tools [22].

6. Conclusion

In this work, we have presented a technique for translating `tock-CSP` into TA for Uppaal, which facilitates using temporal logic and automatic support with Uppaal in verifying `tock-CSP` models. This translation provides an easier way of using TCTL in specifying liveness requirements that are difficult to specify and verify in `tock-CSP`. Also, the result of this work sheds additional insight into the complex relationship between `tock-CSP` and TA, as well as the connection between refinement model and temporal logic model.

So far, in this work, we have used trace analysis in justifying the correctness of the translation work. In the future, we are planning to complete the mathematical proof for the total correctness of the translation technique. Secondly, we are planning to explore using the translation technique in translating extensive case studies that can help us in improving the robustness of our translation work.

Additionally, currently, we translated the event `tock` into an action that is controlled by a timed clock in Uppaal. A recommended next step in this work is relating the notion of `tock` to the notion of time in TA and getting rid of *tock* as an action. This additional extension will help us in exploring additional interesting facilities of Uppaal for verifying temporal specifications.

References

[1] A repository for the translation of tock-CSP into Timed Automata for UPPAAL. Available at: https://github.com/ahagmj/Translation_tockCSP_TA

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 1994.

[3] RJR Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

[4] Gerd Behrmann, Alexandre David, Kim G Larsen, John Håkansson, Paul Pettersson, Yi Wang, and Martijn Hendriks. Uppaal 4.0. *Third Int. Conf. Quant. Eval. Syst. QEST 2006*, pages 125–126, 2006.

[5] Patricia Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323–341, 2009.

[6] Patricia Bouyer. An introduction to Timed Automata. 2011.

[7] Howard Bowman, Giorgio Faconti, J-P Katoen, Diego Latella, and Mieke Massink. Automatic verification of a lip-synchronisation protocol using UPPAAL. *Formal Aspects of Computing*, 10(5-6):550–575, 1998.

[8] Ana Cavalcanti, Augusto Sampaio, Alvaro Miyazawa, Pedro Ribeiro, Madiel Conserva Filho, André Didier, Wei Li, and Jon Timmis. Verified simulation for robotics. *Science of Computer Programming*, 174:1–37, 2019.

[9] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[10] Conrado Daws, Alfredo Olivero, and Sergio Yovine. Verifying et-lotos programs with kronos. In *Formal Description Techniques VII*, pages 227–242. Springer, 1995.

[11] Angela Figueiredo de Freitas. From Circus to Java: Implementation and verification of a translation strategy. *Master's thesis, University of York*, 2005.

[12] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.

[13] Jin Song Dong, Ping Hao, Jun Sun, and Xian Zhang. A reasoning method for timed CSP based on constraint solving. In *International Conference on Formal Engineering Methods*, pages 342–359. Springer, 2006.

[14] Neil Evans and Steve Schneider. Analysing time dependent security properties in CSP using pvs. In *European Symposium on Research in Computer Security*, pages 222–237. Springer, 2000.

[15] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *Int. J. Softw. Tools Technol. Transf.*, 2016.

[16] Thomas Göthel and Sabine Glesner. Automatic validation of infinite real-time systems. In *2013 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 57–63. IEEE, 2013.

[17] Dominik Hansen and Michael Leuschel. Translating TLA+ to B for validation with ProB. In *International Conference on Integrated Formal Methods*, pages 24–38. Springer, 2012.

[18] Dominik Hansen and Michael Leuschel. Translating B to TLA+ for validation with TLC. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 40–55. Springer, 2014.

[19] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[20] Graham Hutton. *Programming in haskell*. Cambridge University Press, 2016.

[21] Yoshinao Isobe, Faron Moller, Hoang Nga Nguyen, and Markus Roggenbach. Safety and line capacity in railways–an approach in timed CSP. In *International Conference on Integrated Formal Methods*, pages 54–68. Springer, 2012.

[22] Nafiseh Kahani, Mojtaba Bagherzadeh, James R Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, 18(4):2361–2397, 2019.

[23] Maneesh Khattri. Translating Timed Automata to tock-CSP. In *Proceedings of the 10th IASTED International Conference on Software Engineering, SE 2011*, 2011.

[24] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. pages 134–152, 1997.

[25] Birgitta Lindstrom, Paul Pettersson, and Jeff Offutt. Generating trace-sets for model-based testing. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 171–180. IEEE, 2007.

[26] Gavin Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Aspects of Computing*, 20(3):277–294, 2008.

[27] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic notes in theoretical computer science*, 152:125–142, 2006.

[28] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. RoboChart: a State-Machine Notation for Modelling and Verification of Mobile and Autonomous Robots. *Tech. Rep.*, pages 1–18, 2016.

[29] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: an appetizer*. Springer Science & Business Media, 2007.

[30] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from CSP models. In *International Colloquium on Theoretical Aspects of Computing*, pages 258–273. Springer, 2008.

[31] Marcel Vinicius Medeiros Oliveira. *Formal derivation of state-rich reactive programs using Circus*. PhD thesis, University of York, 2005.

[32] Joel Ouaknine. *Discrete analysis of continuous behaviour in real-time concurrent systems*. PhD thesis, University of Oxford, 2000.

[33] Andrew William Roscoe. *Understanding Concurrent Systems.* Springer Science & Business Media, 2010.

[34] AW Roscoe, CAR Hoare, and R Bird. The theory and practice of concurrency. 2005. *Revised edition. Only available online*.

[35] Steve Schneider. Concurrent and real time systems : the CSP approach. *Worldw. Ser. Comput. Sci.*, 2010.

[36] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In *International Conference on Computer Aided Verification*, pages 709–714. Springer, 2009.

[37] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[38] Kangfeng Ye and Jim Woodcock. Model checking of state-rich formalism by linking to $CSP \parallel B$. *International Journal on Software Tools for Technology Transfer*, 19(1):73–96, 2017.

Appendix A Details of the Translation Technique

A.1 Characterisation of tock-CSP

This section describes the characterisation of tock-CSP using BNF grammar. The following BNF in Figure 7 defines a valid syntax for constructing a tock-CSP process that we consider within the scope of this work.

$$
\begin{aligned}
NamedProc ::=\ & Name\ CSPproc \\
\mid\ & Name\ CSPexpression\ CSPproc
\end{aligned}
$$

$$
\begin{aligned}
CSPproc ::=\ & STOP \\
\mid\ & Stopu \\
\mid\ & SKIP \\
\mid\ & Skipu \\
\mid\ & Wait(Expression) \\
\mid\ & Waitu(Expression) \\
\mid\ & Event \rightarrow CSPproc \\
\mid\ & CSPproc \ \Box\ CSPproc \\
\mid\ & CSPproc\ \sqcap\ Proc \\
\mid\ & CSPproc; CSPproc \\
\mid\ & CSPproc|||CSPproc \\
\mid\ & CSPproc \underset{\{Event\}}{||} CSPproc \\
\mid\ & CSPproc \vartriangle CSPproc \\
\mid\ & CSPproc\ \Theta\ CSPproc \\
\mid\ & CSPproc \setminus \{Event\} \\
\mid\ & CSPproc[\{Event\}/\{Event\}] \\
\mid\ & EDeadline(Event, Expression)
\end{aligned}
$$

$$Event ::= eventIdentifier \mid tock$$

Figure 7: BNF of tock-CSP for the translation technique

The BNF is implemented into AST using Haskell in the following Definition A.1.

Definition A.1. Data definition of `CSPproc`

```
data CSPproc = STOP
             | Stopu
             | SKIP
             | Skipu
             | WAIT       Int
             | Waitu      Int
             | Prefix     Event      CSPproc
             | IntChoice  CSPproc    CSPproc
             | ExtChoice  CSPproc    CSPproc
             | Seq        CSPproc    CSPproc
             | Interleave CSPproc    CSPproc
             | GenPar     CSPproc    CSPproc [Event]
             | Interrupt  CSPproc    CSPproc
             | Timeout    CSPproc    CSPproc Int
             | Hiding     CSPproc    [Event]
             | Rename     CSPproc    [(Event, Event)]
             | Proc       NamedProc
             | Exception  CSPproc    CSPproc [Event]
             | EDeadline  Event      Int
             | ProcID     String
```

In the following explanation, we use two metavariables P and Q, and decorations on these names, to denote elements of the syntactic category $CSPproc$. We use the symbol e to represent an element of the set Event. Also, the symbols A and B are used to represent a set of events. Lastly, the parameter d represents a CSP expression that evaluates to a positive integer [3].

$STOP$ specifies a process at a stable state in which only the event tock is allowed to happen. This means that the process enables passage of time only, no other events are allowed to happen.

$Stopu$ specifies a process that immediately deadlocks. Unlike the previous process $STOP$, this process $Stopu$ does not allow any time to pass before the deadlock.

$SKIP$ specifies a process that reaches a successful termination point, where it can either terminate or allow time to pass using the event *tock* before termination. In essence, only two events are possible at that state, *tock* for time or *tick* for termination.

[3]Details in `https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html#csp-expressions`

$Skipu$ specifies a process that immediately terminates. Unlike the previous process $Skipu$, this process does not allow time to pass before termination. In essence, the process immediately performs the termination event *tick*.

$WAIT(d)$ specifies a delay process that remains idle for a certain amount of unit time d. After the idle time elapses, either the process terminates with the event *tick* or allows arbitrary units of times to pass before termination.

$Waitu(d)$ specifies an urgent delay process that remains idle for a fixed amount of unit time d. The process terminates immediately after the fixed delay time d.

$e \to P$ Prefix describes a process that offers to engage with an event e and then subsequently perform the behaviour of the process P.

$P \sqcap Q$ Internal choice specifies a process that has different autonomous choices of behaviour, P and Q. Independently the process $P \sqcap Q$ behaves either as P or Q, regardless of the choice of the environment. In the case of this internal choice the environment has no control over the two possible choices of P and Q.

$P \square Q$ External choice specifies a process that is ready to engage in the behaviour of either P or Q depending on the choice of the environment. The process offers to engage with the initials of both P and Q, for each chosen initials the process $P \square Q$ provides the corresponding behaviour of either process P or Q. In the case of this external choice, the environment has control in choosing the behaviour of the process. This is the complement of the previous internal choice where the process has control over the choice of the behaviour.

Well-formedness In the case of external choice, there is a restriction that the event tock is not allowed to appear in the initials of either of the processes. That is $tock \notin (initials(P) \cup initials(P))$. This is because having the event tock as part of the initials will cause non-determinism between the process behaviour and progress of time.

$P; Q$ Sequential Composition specifies a composition of two processes P and Q that run one process after the other. The first process P begins until it terminates, then follow with the behaviour of the subsequent process Q.

$P|||Q$ Interleaving specifies a parallel composition where both the processes run independently without any interaction. In this case, the processes have no common interaction points except for the termination point. Interleaving processes do not synchronise in any of their events.

Well-formedness Implicitly, the processes P and Q have to match the flow of time. If both the two processes perform the time event *tock*, they synchronise with the flow of time on the event *tock*, which implies that the two processes implicitly synchronise on the flow of time and the time event *tock*.

$P \underset{A}{||} Q$ Generalised parallel specifies a parallel composition of two processes P and Q that run in parallel and synchronise on specified set of events A. Independently, each of the processes performs its events that are outside the set A

Well-formedness The set A implicitly contains the event *tock*.

$P \overset{d}{\rhd} Q$ Timeout delay specifies a composition of two processes P and Q, where a deadline d is specified for the first process P to engage with performing an event from it initials $initials(P)$. If the first process P engagee, then the whole process behaves as the process P. After the deadline d time unit, if the first process P did not engages, the second process Q take over the control, and the whole process behaves like the second process Q.

Well-formedness The expression d should be an expression that evaluates to a natural number. This is because tock-CSP is based on a discrete-time model that records time-progress with discrete-event tock. Also, both the two processes P and Q are not allowed to begins with the timed event *tock*.

$P \bigtriangleup Q$ Interrupt operator describes a process P that can be interrupted by another process Q at any time during the execution of P. The first process P runs until the second process Q performs a visible event. Whenever the second process performs an external action, it interrupts the execution of the first process. The interrupted process is blocked, and the second process takes over the control, then the whole process behaves as the second process Q. If the second process Q did not interrupt the whole process behave as the first process P.

Well-formedness There is restriction that the event *tock* is not allowed to be in the initials of the second (interrupt) process. This means that an interruption cannot begin with the event tock. This is because time event *tock* can cause non-determinism between the interrupt and the process of time.

$P \setminus A$ Hiding specifies the behaviour of a process P which hide all the events in set A. The hidden events A becomes special event *tau* that are not visible to the environment, as such the environment has no control over the hidden events.

Well-formedness In the case of hidden, there is a restriction that hidden events should not include the time event *tock*. This is because a process should not control the progress of time.

$P[A/B]$ Renaming specifies a process that renames a list of its events A with corresponding names of events in list B, in one to one mapping. The renaming operator transforms a process into another process with the same structure but appears with different names of the renamed events A.

Well-formedness There is restriction that the event *tock* cannot be renamed to another event, and no other event can be renamed to be *tock*. This is because the time-event tock is a special event dedicated for recording the progress of time.

$Edeadline(e, d)$ specifies a process that must perform the event e within the deadline d. So the event e must happen within the deadline d.

A.2 Translation Rules

This section discusses the details of the translation rules. The section describes the translation rules in functional style and provides examples that illustrates using each of the translation rules in translating a tock-CSP process. We begin with describing the components of the translation rules. The input is a valid `tock-CSP` within the scope of the presented BNF. The output is a valid TA for Uppaal.

Example A.1. Here, we use a simple example of TA to illustrate the definition of TA provided in Figure 8, which is defined in Listing 1 using the syntax of Haskell, which we used in the translation work.

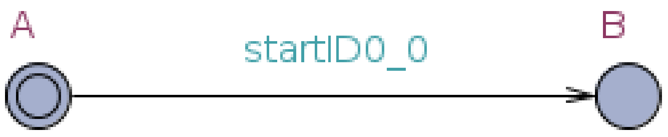


Figure 8: An illustration of an output TA with two location and one transition.

```
1  TA  idTA   []  []  [loc1, loc2] (Init loc1) [tran1]
2    where
3      idTA  = "ta1" + 0  + "_" + 0
4        --  = Location  ID    Name  Label       LocType
5      loc1  = Location "idA"  "A"   EmptyLabel  None
6      loc2  = Location "idB"  "B"   EmptyLabel  None
7
8         -- = Transition  Source  Target  [Label]
9      tran1 = Transition  loc1    loc2    [lab1]
10
11     lab1  = Sync (VariableID  "start")  Excl
12     .
```

Listing 1: An abstract definition of a TA that has two locations and one transition.

Line 1 defines a TA using the data definition of TA with the required 6 arguments. First parameter `"idTA"` specifies an identifier for the TA, similar to the naming format we use in the translation work. Second and third parameters are empty lists for both parameters of the TA itself and its local definitions. Fourth, `[loc1, loc2]` is a list of locations for the TA that contains 2 locations, `loc1` and `loc2`. Fifth, `(init loc1)` specifies loc1 as the initial location of the TA. Lastly, `[tran1]` is a list of transitions that has one transition for the TA.

Line 3 highlights a data definition of location, described below in the Definition A.2. Then, Line 4 defines `Loc1` as an instance of location with an identifier `"idA"` and name `"A"`, with an empty label that indicates no constraint in the location, and also specify the type of the location to be None. In the like manner, Line 5 defines `loc2` as the second location with an identifier `"idB"`, name `"B"`, also label with empty that specify no constraint in the location, and specify a type for the location to be None.

Line 7 is a comment that highlights a definition of a transition. Then, Line 8 defines `tran1` as a transition that connects two locations `loc1` and `loc2` with `[lab1]` as a label of the transition. `lab1` is defined in Line 10 using the definition of label from Definition A.2 as an UPPAAL action with identifier "start" that has direction Excl which specifies the acation as a sender.

The above Example A.1 illustrates a simple form of the output TA produced by the translation function `transTA`. However, in the translation rule we will have a TA that has more than two locations and multiple transitions. The upcoming translation rules define the function `transTA`. Each rule defines a translation of one of the constructors of the BNF previously presented in Section A.1. In the next section, we discuss details of each of the translation rules together with an example for illustrating using the rule in translating a process.

Location

```
data Location = Location ID Name Label LocType
```

Definition A.2 defines a location with a constructor that has 4 parameters, of types `ID, Name, Label and LocType.` First parameter of type `ID` is an identifier for the location. Second parameter of type `Name` is a tag for the location. Third parameter of type `Label` is a constraint label for the location, defined below in Definition A.2. Last parameter of type `LocType` is a format of the location, which can be one of these three: `urgent`, `committed`, `None` (which means neither urgent nor committed, just normal location with no constraint).

Transition

```
data Transition = Transition Source Target [Label]
```

Also, Definition A.2 defines a data type for Transition, which has a constructor with 3 parameters, of type `Source`, `Target` and `[Label]`. First parameter of type `Source`, is a starting location for the transition. Second paramter of type `Target` is a destination location for the transition. Third parameter of type `[Label]` is a list of labels for the transition.

Label

```
data Label = EmptyLabel
           | Invariant     Expression
           | Guard         Expression
           | Update       [Expression]
           | Sync          Identifier   Direction
```

Finally, label is an expression (or list of expressions) that is associated with either a location or a transition. For a location, a label can be empty or invariant that specifies the constraint condition of the location. While for a transition the label can be either empty for silent transition or any combination of these three types: `Guard`, `Update` and `Sync`. Where `Sync` is a type for an UPPAAL action that has an identifier and direction, which is either sender (with question mark) or receiver (with exclamation mark).

Function `transTA`

```
transTA :: CSPproc -> ProcName  -> BranchID -> StartID -> FinishID
                 -> UsedNames -> ([TA], [Event], [SyncPoint])
```

The function `transTA` has 6 parameters. The type of the parameters are `CSPproc` , `ProcName`, `BranchID`, `StartID`, `FinishID` and `UsedNames`. First parameter of type `CSPproc`, is the input CSP process to be translated. Second paramter is a name for the process, of type `ProcName`; an alias for `String`. While third and fourth paramter are of type `BranchID` and `StartID`; also alias of String and Int respectively. We use the two parameters to generate an identifier for each small TA in the list of the output TA. In generating the identifiers, we consider the structure of binary tree for the AST of the input process, which has branch and depth. So a combination of these two parameters branch and depth identifies a position of each TA in the list of the output TA. Fifth parameter of type `FinishID`, is a termination ID. Last parameter of type `UsedNames` is a collection of names, which we used in defining the translation function, mainly for passing translation information from one recursive call to another. We will explain the purpose of these names as we introduce them in the translation rules. The first 3 parameters `ProcName`, `BranchID` and `StartID` are essential for each translation rule.

A.2.1 Translation of STOP

This section describes a translation of a constant process `STOP`. The section begins with presenting a rule for translating STOP and then follows with an example that illustrates using the rule in translating a process.

Rule A.1 expresses the translation of the construct STOP, which produces an output TA depicted in Figure 9. The figure illustrates the structure of the output TA that has 2 locations and 2 transitions as defined in Lines 7–9 and lines 11–13 respectively.

Starting from the beginning of the translation rule, Line 1 provides a definition of the function `transTA` for the construct `STOP` and the 3 essential parameters for translating the construct STOP, `processName`, `bid` and `sid` . While the remaining 3 underscores represent unused arguments for this translation rule. In Haskell, an underscore indicates a position of unused arguments. For conciseness, we use the underscore to omit unused arguments and provide only the required arguments for each translation rule.

Line 2 defines the output tuple which contains 3 elements, a list of output TA, and the remaining two elements for translating multi-synchronisation. For this translation rule, there is no multi-synchronisation, so the remaining two elements are both empty for the synchronisation actions and their corresponding identifiers.

Also, in the output tuples, the first element (non-empty element) is a definition of the output TA for the translation of the constant process STOP, which has 6 parameters. First, `idTA` is an identifier for the TA, which is define subsequently in Line 4, as concatenation of the keyword `"taSTOP__"` with the 2nd and 3rd arguments of the function `transTA`, that is `bid` and `sid` respectively. Additionally, still in Line 2, in the definition of the output TA, the 2nd, 3rd and 5th parameters are empty for the output TA. While the 4th parameter `locs` is a list of locations for the output TA defined in Lines 7–9. The 6th parameter `(Init loc1)` specifies `loc1` as the initial location of the output TA. Lastly, `trans` describes a list of transitions that connect the two locations as defined in line 12–13. Lines defines interrupt transitions in the case of translating a process that is composed with an operator interrupt.

Rule A.1. Translation of STOP

```
transTA STOP processName bid sid _ _ _ =
     (([(TA idTA [] [] locs [] (Init loc1) trans)]), [], [] )
     where
        idTA = "taSTOP__" ++ bid ++ show sid

        --   = Location ID      Name    Label       LocType
        loc1 = Location "id1"   "s1"    EmptyLabel  None
        loc2 = Location "id2"   "s2"    EmptyLabel  None
        locs = [loc1, loc2]

        --    = Transition Source   Target  [Label]  [Edge]
        tran1 = Transition loc1     loc2    [lab1]   []
        tran2 = Transition loc2     loc2    [lab2]   []
        intrp = transIntrpt intrptsInits loc1 loc2
        trans = [tran1, tran2] ++ intrp

        lab1 = Sync (VariableID
                        (startEvent processName (bid ++ sid)) [])
                    Ques
        lab2 = Sync (VariableID "tock" []) Ques

        -- Get initial events for possible interrupting process
        (_, _, _, _, _, intrptsInits, _, _) = usedNames

```

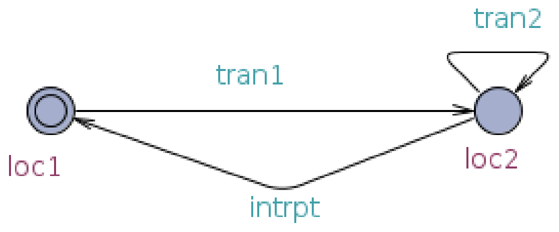


Figure 9: A structure of TA for the translation of STOP.

Finally, Line 17 – 20 defines the labels of the two transitions in the output TA. Label `lab1` defines a label for the first transition as a first flow action, which we generate its name using the function `startEvent`, defined in the following Definition A.2.1. `Lab2` is label for the second transition, which is defined as an UPPAAL action ”tock” with

Ques that indicates a receiver.

Function `startEvent`

```
startEvent :: String      -> String -> Int      -> String
startEvent    processName    bid       sid      =
                             if    notNull      processName
                             then "startID" ++  processName
                             else "startID" ++  bid ++ show sid
```

In developing the translation rules, we use a function `startEvent` Definition A.2.1 for generating a name for the first starting flow action of the first TA in each list of the translated TA. If the input process has an identifier, we used the identifier in the translated TA. Otherwise, the function `startEvent` generates a name for the list of the translated TA. The name is a combination of the keyword `"startID"` with the two identifiers of the first TA in the list of the translated output, that is a combination of the parameters `BranchID` and `StartID`.

Line 14 define an interrupt transitions, which is provided for the case of translating a process that involves interrupt. Details of translating interrupt will be provided in Section A.2.13. Here, we highlight a declaration of the function `transIntrpt` below, due to its first appearance in Line 14.

Declaration of the function `transIntrpt`

```
transIntrpt :: [Event] -> Location -> Location -> [Transition]
```

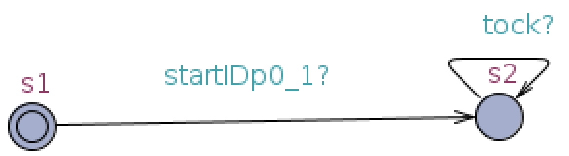
The function `transIntrpt` generates transitions for interrupt using the initials of an interrupting process, as highlighted in the the overview of the translation technique. The function `transIntrpt` has 3 parameters. The types of the parameters are list of event (initials of an interrupting process) and two locations that connect the transition for interrupt. The first argument `intrpts` is the initials of an interrupting process, and generates a transition for each of the initials.

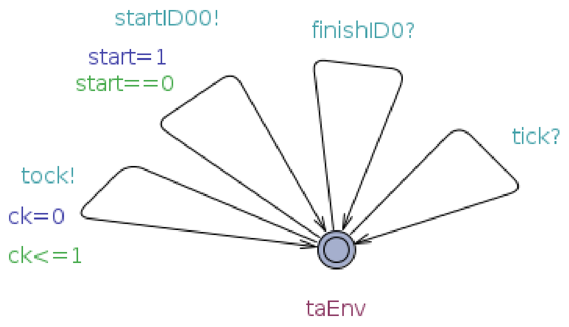
The list of the initials `intrpts` comes from the tuple `usedNames` (Line 23). Previously, we mentioned that we will explain the names in the point where we start using the names. Here, we start using the name `intrpts` from the names `usedNames`. The name `intrpts` is use to collect the initials of interrupting processes for constructing interrupting transitions that enables a translated process to interrupt the behaviour of another process.

The behaviour of the output TA begins with the first flow action (line 17), which is constructed using a function `startEvent`, previously defined in Definition A.2.1. After that, the TA performs the action `tock` (line 18), repeatedly, which allows time to progress. An illustration of using this translation rule is provided in the following Example A.2.

Example A.2. An example of translating a process STOP produces a list of TA that contains two TA is illustrated below.

```
transTA STOP "p0_1" 1 0 ([], [], [], [], [], [], [], ([],[])
    ) = [
```





```
    ]
```

Example A.2 demonstrates a translation of the process STOP using Rule A.1, which produces a list of translated TA that has contains two TA, a small TA and its corresponding environment TA as shown in the above figures. The behaviour of the output TA begins with the environment TA that performs its first flow action `startID00!` with the cooperation of the small TA using its corresponding co-action `startID00?` . Then, the small TA continues performing the event `tock` for the progress of time, and remains in location `s2`. This concludes the behaviour of the translated TA for the constant process `STOP`.

A.2.2 Translation of Stopu (Timelock)

This section describes a translation of constant process `Stopu`, an urgent deadlock that does not allow time to pass. The section begins with presenting a rule for translating the process `Stopu`. Then, follows with an example that illustrates using the rule in translating a process.

Rule A.2. Translation of Stop*u*

```
1  transTA Stopu processName bid sid _ _ _ =
2      ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
3      where
4          idTA = ("taSTOP__" ++ bid ++ show sid)
5
6           --  = Location  ID    Name Label       LocType
7          loc1 = Location "id1" "s1"  EmptyLabel  None
8          loc2 = Location "id2" "s2"  EmptyLabel  None
9          locs = [loc1, loc2]
10
11          --    =  Transition  Source   Target   [Label] [Edge]
12         trans = [Transition  loc1     loc2     [lab1]  []   ]
13
14         lab1  = Sync (VariableID
15                          (startEvent processName (bid ++ sid))  [])
16                      Ques
```

Rule A.2 expresses the translation of the constant process `Stopu` which produces an output TA that is depicted in the following Figure 10, which is annotated with the names used in the translation rule. The figure illustrates the structure of the output TA, which has 2 locations and 1 transition as defined in Lines 7–9 and Line 12 respectively.

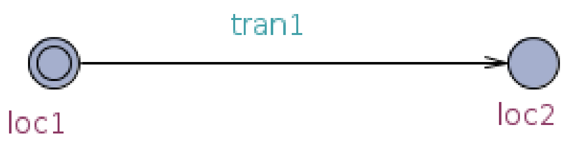
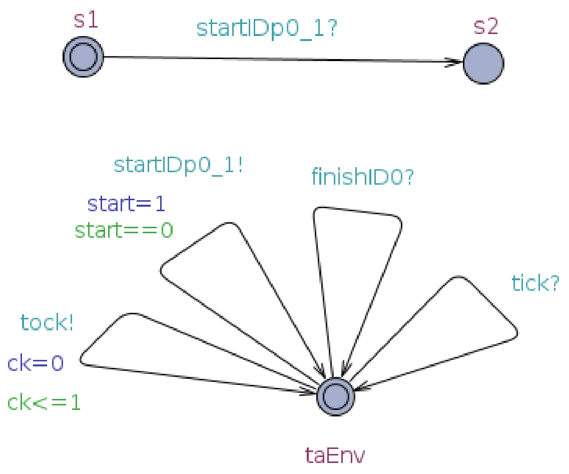


Figure 10: A structure of TA for the translation of `Stopu`

This description of Rule A.2 resembles the previous description of Rule A.1 (translation of STOP), except that the output TA of this rule does not perform the event `tock` that allows time to progress in the previous rule. The structure of this output TA has two locations `loc1` and `loc2` define in Lines 7 and 8, respectively, and only one transition for the coordinating start event (Line 11). This behaviour of the output TA begins with with synchronising on the first coordinating start event and then deadlock immediately. This is illustrated in the following example for translating the constant process `Stopu`.

Example A.3. An example for translating an urgent process Stopu.

```
-- transTA :: CSPproc -> procName -> BranchID  -> StartID
--                     -> FinishID -> UsedNames ->
--                             ([TA], [Event], [SyncPoint])
transTA Stopu "p0_1" "0" 1 0
          ([], [], [], [], [], [], [], ([],[])) =
               (outPutTA, [], [])
where
     outPutTA = [
```



```
          ]
```

The above Example A.5 illustrates a translation of the constant process `Stopu` according to Rule A.2. Also, this example resembles the previous Example A.2, except that the behaviour of this TA terminates immediately without performing the event `tock`. In this example, the output TA synchronises on the coordinating start event `startID00` and then deadlocks immediately.

A.2.3 Translation of SKIP

This section describes the translation of process SKIP. The section begins with presenting a rule for translating the process SKIP. Then, follows with an example that illustrates using the rule in translating a process. However, the behaviour of SKIP can be interrupted just before termination, so we have to consider a possible interrupt in translating interrupt.

Rule A.3 describes a translation of process `SKIP` into a single output TA, which is depicted in the following Figure 11. The figure is annotated with the names used in the translation rule. The structure of the output TA has 3 locations: `loc1`, `loc2` and `loc3` (define in Lines 6 – 9) and 4 transitions `tran1`, `tran2` and `tran3` (define in Lines 11 – 16).

The behaviour of the TA begins on `tran1` for a flow action that is define using the function `startEvent` (Definition A.2.1). Then, the TA follows one of the 3 transitions: `tran1, tran3 or intrp`. On transition `tran2`, the output TA performs the event `tock`

Rule A.3. Translation of SKIP

```
1  transTA SKIP procName bid sid fid usedNames =
2       ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
3       where
4          idTA = "taWait_n" ++ bid ++ show sid
5
6          loc1 = Location "id1" "s1" EmptyLabel None
7          loc2 = Location "id2" "s2" EmptyLabel None
8          loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
9          locs = [loc1, loc2, loc3]
10
11         tran1 = Transition loc1 loc2 [lab1]  []
12         tran2 = Transition loc2 loc2 [lab2]  []
13         tran3 = Transition loc2 loc3 [lab3]  []
14         tran4 = Transition loc3 loc1 [lab4]  []
15         intrp = transIntrpt intrptsInits loc1 loc2
16         trans = [tran1, tran2, tran3, tran4] ++ intrpt
17
18         lab1 = Sync (VariableID (startEvent processName
19                     (bid ++ sid)) []) Ques
20         lab2 = Sync (VariableID "tock" []) Ques
21         lab3 = Sync (VariableID "tick" []) Excl
22         lab4 = Sync (VariableID  finishLab []) Excl
23
24         finishLab = ("finishID" ++ show fid)
25
26         -- Get initial events for possible interrupting process
27         (_, _, _, _, _, intrptsInits, _, _) = usedNames
28
```

to record the progress of time, and remains in the same location `loc2`. On transition `tran3`, the output TA performs the event `tick` and then immediately follows the subsequent transition `tran4` to perform the termination event `finishID0!`. Finally, on transition `intrp`, the TA is interrupted by another process.

Line 15 define an interrupt transitions, which is provided for the case of translating a process that involves interrupt. Details of translating interrupt will be provided in Section A.2.13. Here, we highlight a declaration of the function `transIntrpt` below, due to its first appearance in this translation rule.

The list of the initials `intrpts` comes from the tuple `usedNames` (Line 27). Previously, we mentioned that we will explain the names in the point where we start using

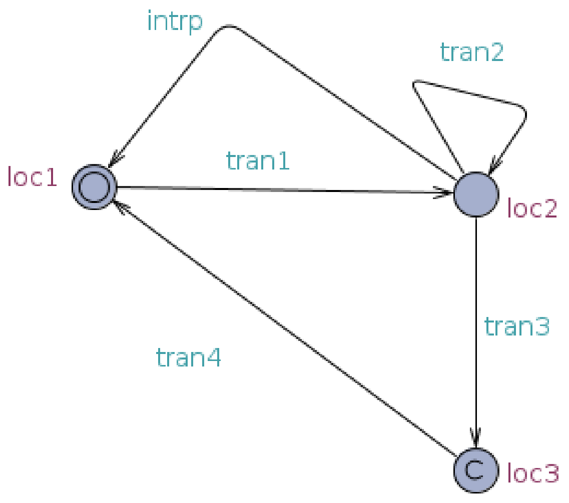
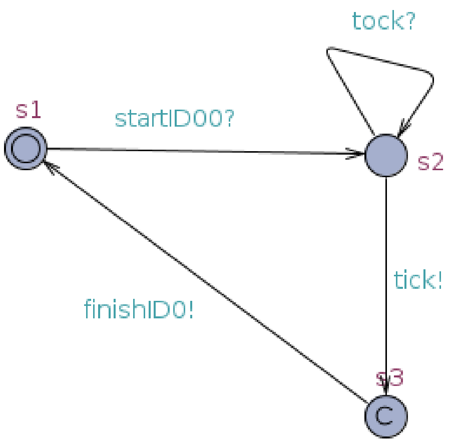


Figure 11: A structure of a TA for the translation of the process SKIP.

the names. In this rule (Line 27), we start using the name `intrpts` from the names `usedNames`. We used the name `intrpts` to collect the initials of interrupting processes for constructing interrupting transitions. This completes the description of Rule A.3. This is illustrated in the following Example A.4 for translating constant process `SKIP`.

Example A.4. An example for translating a constant process `SKIP`.

```
transTA SKIP "" "0" 0 0
        ([], [], [], [], [], [], [], ([],[])) =
        [
```



```
        ]
```

Example A.4 illustrates using Rule A.4 in a translating a constant process `SKIP`. The example uses the definition of the translation function `transTA` for the construct `SKIP` and the required parameters: process `SKIP`, empty name, ”0” for `BranchID`, 0 for `StartID`, 0 for `finishID`, and a tuple of empty lists for the `usedNames`. We used empty name to illustrate the translation of a process that has an empty name.

Details of the output TA are as follows. Initially, the output TA synchronises on the start event `startID00?`. In this example, the translated process does not have a name, so the start event is a concatenation of the keyword ”startID” with 0 for `BranchID` and another 0 for `StartID`. After that, on location `s2` either the TA performs the event `tock` and returns to the same location; or performs the event `tick` and then immediately performs the termination event `finishID0`, which notifies the TA environment for a successful termination of the output TA.

In this example, there is no interrupting process, so the function `transIntrpt` produces an empty list for the interrupting transitions. Section A.2.13 provides an example that has interrupting transitions. For better understanding, we will discuss the example with interrupt transitions after discussing the translation of the construct `interrupt`. This completes the description of an example for translating a constant process `SKIP`.

A.2.4 Translation of Skipu (Urgent termination)

This section describes the translation of another constant process `Skipu`, which specifies an urgent termination that does not allows time to pass before the termination. The section begins with presenting a rule for translating the process `Skipu`. And then, follows with an example that illustrates using the rule in translating a process.

Rule A.4. Translation of Skipu

```
transTA Skipu procName bid sid fid usedNames =
      ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
      where
        idTA = "taSkipu_" ++ bid ++ show sid

        loc1 = Location "id1" "s1" EmptyLabel None
        loc2 = Location "id2" "s2" EmptyLabel None
        loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
        locs = [loc1, loc2, loc3]

        tran1 = Transition loc1 loc2 [lab1]  []
        tran3 = Transition loc2 loc3 [lab3]  []
        tran4 = Transition loc3 loc1 [lab4]  []
        trans = [tran1, tran3, tran4]

        lab1 = Sync (VariableID (startEvent procName bid sid) [])
                    Ques

        lab3 = Sync (VariableID "tick"         []) Excl
        lab4 = Sync (VariableID ("finishID" ++ show fid) []) Excl
```

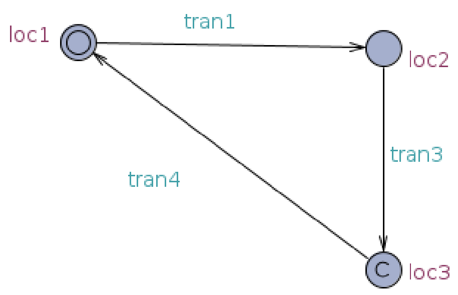
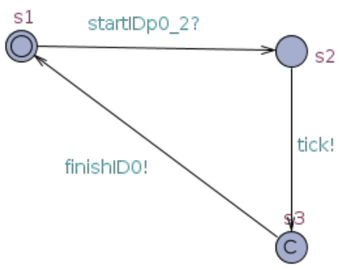


Figure 12: A structure of TA that maps the names used in Rule A.4

Rule A.4 resembles the previous Rule A.3, except that on location s2 the output TA does not perform event tock. This means that the TA terminates immediately as illustrated in Figure 12. The following example demonstrates using the rule in translating a process.

Example A.5. An example for translating a process for an immediate termination.

```
transTA Skipu "" "0" 0 0
        ([], [], [], [], [], [], [], ([],[]))
        = [
```



```
          ]
```

Similarly Example A.5 resembles Example A.4, except that the output TA terminates immediately. Initially, the TA synchronises on the coordinating start event `startID00`, then either performs the event tock or performs the event `tick`. On performing the event `tock` the TA remains in the same location `s2`. While on performing the event `tick` the TA proceeds immediately to perform a termination action `finishID0`, which indicates a successful termination. There is no interrupting process in this example, so the interrupting transitions are empty. This completes the description of Example A.5, which illustrates a translation of the constant process `Skipu` for urgent termination.

A.2.5 Translation of Prefix

This section describes the translation of operator `Prefix`. The section begins with presenting a rule for translating the operator `Prefix`, and then follows with an example that illustrates using the rule in translating a process.

This rule for translating prefix happens to be the largest translation rule because we can not translate an event without knowing whether the event used in the prefix is, in the overall process, being hidden, renamed, or used as part of synchronisation, initial of external choice or initial of interrupt. In each of these cases the TA has different behaviour.

So, first the translation rule defines a function for checking both hidden and renamed events. And then defines eight cases for capturing the possible behaviour of an event in a process that participates in synchronisation, external choice or interrupt.

Rule A.5. Translation of Prefix

```
transTA (Prefix e1 p) procName bid sid fid usedNames =
        (([(TA idTA [] [] locs1 [] (Init loc1) trans1)] ++ ta1),
                sync1, syncMapUpdate)
  where
    idTA = "taPrefix" ++ bid ++ show sid
    (syncs, syncMaps, hides, renames, exChs, intrpts, initIntrpts,
            excps) = usedNames

    -- Checking hiding or renaming
    e = checkHidingAndRenaming e1 hides renames

    -- High level definition of locations and transitions for the
    -- eight possible combination of synchronisation, choice and
    -- interrupt, 000, 001, 010, 011, 100, 101, 110, 111
    (locs1, trans1)
        |((not synch) && (not exChoice) && (not interupt)) = case1
        |((not synch) && (not exChoice) && (    interupt)) = case2
        |((not synch) && (    exChoice) && (not interupt)) = case3
        |((not synch) && (    exChoice) && (    interupt)) = case4
        |((    synch) && (not exChoice) && (not interupt)) = case5
        |((    synch) && (not exChoice) && (    interupt)) = case6
        |((    synch) && (    exChoice) && (not interupt)) = case7
        |((    synch) && (    exChoice) && (    interupt)) = case8


    case1 = ([loc1, loc2, loc5],
             [t12, t25,  t51] ++ addTran ++ transIntrpt')
    case2 = ([loc1, loc2, loc3c, loc5],
             if not $ null intrpts
             then [t12G, t23ci, t3c5, t51] ++ addTran
             else [t12,  t23ci, t23cgi, t3c5, t51] ++ addTran
                  ++ transIntrpt')
    -- if a process can interrupt and also be interrupted,
    -- then it can only be interrupted after initiating its interrupt
    case3 = ([loc1, loc2,  loc3c,  loc5],
             [t12,  t23c,  t3c5,    t51] ++ t23e ++ addTran
             ++ transIntrpt')
    case4 = ([loc1, loc2,  loc3c,  loc4c,   loc5],
             [t12G, t23c,  t3c4ci, t3c4cgi, t4c5e,  t51] ++
             addTran ++ transIntrpt')
    case5 = ([loc1, loc2, loc3, loc5],
             [t12,  t23,  t35,  t51] ++ addTran ++ transIntrpt')

```

(Cont. 1) Translation of Prefix

```
case6 = ([loc1, loc2,  loc3,   loc4c,   loc5],
         [t12G, t23,   t34c,   t4c5i,   t4c5gi, t51]  ++
         addTran ++ transIntrpt')
case7 = ([loc1, loc2,  loc3,   loc4,    loc5],
         [t12,  t23ech, t34,   t45,     t51] ++
         addTran ++ transIntrpt')
case8 = ([loc1, loc2,  loc3,   loc4,    loc5,   loc6],
         [t12G, t23,   t34c,   t4c6,    t65, t65gi, t51] ++
         addTran ++ transIntrpt')

--     =  Location  ID    Name  Label      LocType
loc1   =  Location "id1"  "s1"  EmptyLabel None
loc2   =  Location "id2"  "s2"  EmptyLabel None
loc2c  =  Location "id2"  "s2"  EmptyLabel CommittedLoc
loc3   =  Location "id3"  "s3"  EmptyLabel None
loc3c  =  Location "id3"  "s3"  EmptyLabel CommittedLoc
loc4   =  Location "id4"  "s4"  EmptyLabel None
loc4c  =  Location "id4"  "s4"  EmptyLabel CommittedLoc
loc5   =  Location "id5"  "s5"  EmptyLabel CommittedLoc
loc6   =  Location "id6"  "s6"  EmptyLabel CommittedLoc

transIntrpt' = (transIntrpt intrpts loc1 loc2)

-- Additional transitions for tock, external choice
addTran | ((not  $ elem e syncs) && (null exChs))       = [t22]
        | ((elem e syncs)        && (null exChs))       = [t22]
        | ((not  $ elem e syncs) && (not $ null exChs))  = [t22]
                                                          ++ t21
        | otherwise = [t22, t33, t44] ++ t21


t23ci   = Transition  loc2     loc3c   l23ci           []
t23cgi  = Transition  loc2     loc3c   altIntrpt       []
l23ci   = [(Sync
             (VariableID   ((show e) ++ "_intrpt")  []) Excl),
             (Update  [(AssgExp  (ExpID ((show e) ++
             "_intrpt_guard")) ASSIGNMENT TrueExp )])]

--  An alternative transition in case another event has
-- already initiates the interrupt.
-- Guard other possible interrupts, such that any of
-- the interrupt can enable the alternative transition
altIntrpt = [(Guard
               (ExpID (intercalate " || "  [l ++
               "_intrpt_guard" | (ID l) <- initIntrpts])))]
```

(Cont. 2) Translation of Prefix

```

    -- reset the guards in case of recursive process
    resetG  = [Update  [(AssgExp  (ExpID (l ++ "_intrpt_guard"))
                    ASSIGNMENT FalseExp)| (ID l) <- initIntrpts]]
    t3c5    = Transition  loc3c    loc5    lab4e            []
    t3c4ci  = Transition  loc3c    loc4c   l23ci            []
    t3c4cgi = Transition  loc3c    loc4c   altIntrpt        []
    t4c5    = Transition  loc4c    loc5   (lab2i ++ lab2d)  []
    t4c5e   = Transition  loc4c    loc5    lab4e            []
    t34c    = Transition  loc3     loc4c   lab4             []
    t3c4c   = Transition  loc3c    loc4c   lab4e            []
    t4c5i   = Transition  loc4c    loc5    l23ci            []
    t4c5gi  = Transition  loc4c    loc5    altIntrpt        []
    t4c6    = Transition  loc4c    loc6   (lab2i ++ lab2d)  []
    t65     = Transition  loc6     loc5    l23ci            []
    t65gi   = Transition  loc6     loc5    altIntrpt        []
    t12     = Transition  loc1     loc2    lab1             []
    t12G    = Transition  loc1     loc2   (lab1 ++ resetG)  []
    t23     = Transition  loc2     loc3    lab2i            []
    t2c3    = Transition  loc2c    loc3    lab2i            []
    t23c    = Transition  loc2     loc3c  (lab2i ++ lab2d)  []
    t23ech  = Transition  loc2     loc3   (lab2i ++ lab2d)  []

    t25     = if   elem e hides
              then Transition    loc2  loc5
                ([(Sync (VariableID "itau" []) Excl)]) [] -- hiding
              else Transition    loc2  loc5   lab2i    []
    t25r    =  Transition  loc2  loc5  ([(Sync (VariableID
                   (show new_e) []) Excl)] ++ labpath) [] -- renaming
    new_e   =  head [newname | (oldname, newname) <- renames,
                                                oldname == e]
    t51     =  Transition  loc5  loc1  [lab3]          [] -- startTA
    t33     =  Transition  loc3  loc3  [labTock]       [] -- tock
    t44     =  Transition  loc4  loc4  [labTock]       [] -- tock
    t35     =  Transition  loc3  loc5   lab4           []
    t22     =  Transition  loc2  loc2  [labTock]       [] -- tock
    t21     = [(Transition loc2  loc1  [(Sync  (VariableID
                ((show ch) ++ "_exch")  []) Ques)] [])|ch <- exChs']
    t23e    = [(Transition loc2  loc3  [(Guard (ExpID ((show ch) ++
                                  "_exch_ready")) )] [])|ch <- exChs']
    t34     =  Transition  loc3  loc4   lab6  []
    t45     =  Transition  loc4  loc5   lab4  []

    lab1    = [Sync (VariableID (startEvent procName bid sid) []) Ques]

```

(Cont. 3) Translation of Prefix

```
    lab2i   | (elem e syncs) && (null exChs')  =    -- check sync
                [(Guard (BinaryExp (ExpID ("g_" ++
                    (eTag e syncMaps' ""))) Equal (Val 0))),
                (Update [(AssgExp (ExpID ("g_" ++
                    (eTag e syncMaps' ""))) AddAssg (Val 1))])]
            | (not $ null exChs') =
                    if (elem e hides)
                    then [(Sync (VariableID "itau_exch"  []) Excl)]
                    else [(Sync (VariableID ((show e ) ++
                      "_exch")     []) Excl)]
            | otherwise    =  lab4e

    labpath = [(Update  [(AssgExp (ExpID "dp") AddAssg (Val 1)),
              (AssgExp  ( ExpID   ("ep_" ++ bid ++ "_" ++ show sid))
                            ASSIGNMENT TrueExp )])]
                            -- Attaching path variable transition
    -- Checks for exception
    lab3    =  if elem e (fst excps)
               then Sync (VariableID ("startExcp" ++ (show fid )) [])
                         Excl
               else Sync (VariableID ("startID" ++ bid ++ "_" ++
                         show (sid+1)) []) Excl
    lab4    = [(Sync (VariableID ((show e) ++ "___sync") []) Ques)]

    lab4e   | e == Tock    = [(Sync (VariableID (show e) []) Ques)]
                             ++ labpath  -- Sync on tocks
            | elem e hides = [(Sync (VariableID ("itau") []) Excl)]
                            -- itau for hiding event
            | otherwise    = [(Sync (VariableID (show e) []) Excl)]
                                   ++ labpath  -- Fire normal event

    lab6    = [(Guard (BinaryExp (ExpID ("g_" ++ (eTag e syncMaps'
                        ""))) Equal (Val 0))),
              (Update [(AssgExp (ExpID ("g_" ++ (eTag e syncMaps'
                        ""))) AddAssg (Val 1))])]

    lab2d   = [(Update [(AssgExp  (ExpID ((show ch) ++ "_exch_ready"))
                AddAssg   (Val 1)) | ch <- exChs'])]

    gIntrpt = [(Guard (BinaryExp (ExpID "gIntrpt") Equal   (Val 1)))]

    uIntrpt = [(Update [(AssgExp (ExpID "gIntrpt") AddAssg (Val 1))])]

    labTock = Sync (VariableID  "tock" [])  Ques
```

(Cont. 4) Translation of Prefix

```
    synch    = elem e syncs
    exChoice = null exChs
    interupt = null initIntrpts

    -- Update sync points
    syncMaps' = if elem e syncs
                then [(e, (show e) ++ bid ++ "_" ++ show sid )]
                else [] -- syncMaps_

    -- Combine the synchronisations together
    syncMapUpdate = syncMaps' ++ syncMap1

    -- Replace renamed event with the new name
    exChs' =  if  ( null crs ) then exChs
              else  (exChs \\ [es']) ++ [nn']

    -- rename all events for blocking external choice
    crs  = [(es, nn) | (es, nn) <- renames, ch <- exChs, ch == es]
    (es', nn') = head crs

    -- Update used names and then remove external choice and
    -- interrupt if any, after the first event.
    usedNames' =
        (syncs, syncMaps, hides, renames, [], intrpts, [], excps)

    -- finally recursive call for subsequent translation.
    (ta1, sync1, syncMap1) = transTA p []  bid (sid+1) fid usedNames'

```

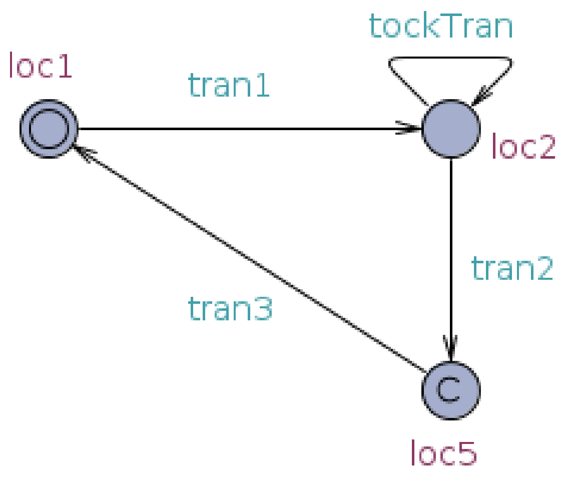


Figure 13: A structure of a TA for the translation of an event from the translation Rule A.2.5, for a translation an event in the case1.

As discussed in Section A.1, the operator prefix is a binary operator that combines an event with a process, syntactically in the form of `event->Process`. The prefix event is translated according to one of the eight possible cases for a process that takes part in synchronisation, external choice and interrupt. Each case defines a separate behaviour for the prefix event in the translation rule.

In Figure 13, we annotate the structure of the TA for translation an event in case 1. The TA has 3 locations and 4 transitions, as defined in Lines 27–29 and Lines 31–37 respectively. The TA begins with transition `tran1` for performing flow action, and then on location `loc1` either the TA performs the action `tock` to record the progress of time and return to the same location, or perform the translated action on transition `tran2` that leads to location `loc3`. Then, on transition `tran2` the TA performs another flow action to activate the subsequent TA. The remaining 7 cases follow similar pattern.

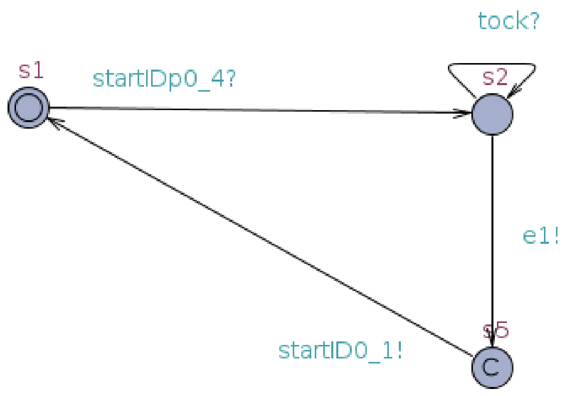
Cases 1 to 4 are cases that did not involve synchronisations. Case 1 is the simple case where a prefix event is not part of any of one the 3 operators. Case 2 defines a translation of an event that is part of the initials of an interrupting process, which means that the event is the kind of event that interrupts the behaviour of another process. Case 3 is for an event that is part of initials of a process that participate in external choice only. Case 4 is for an event that is part of a process that participate in both external choice and interruption.

Cases 5 to 8 are cases that involve synchronisations. Case 5 defines a translation of an event that is part of synchronisation only, which means that multiple processes synchronise on performing the event. Case 6 is for the translation of an event that is part of both synchronisation and interrupt. Case 7 is for the translation of an event that is part of both synchronisation and external choice. Finally, case 8 defines a translation of an event that is part of the 3 operators: synchronisation, external choice and interrupt.

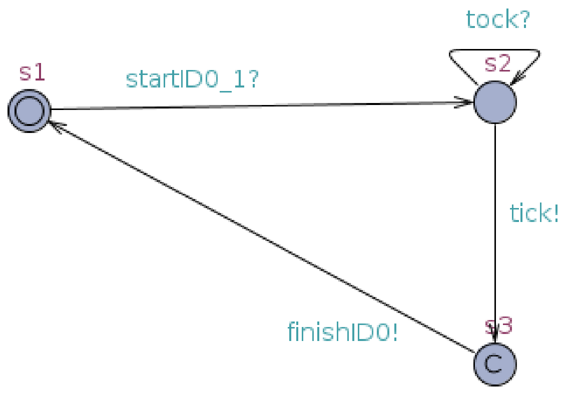
For each of these 8 cases, Rule A.2.5 defines a separate TA for translating the behaviour of a prefix event. Definition of all the locations and transitions of these 8 possible TA generate a long list of definitions that makes the rule to be very large. Here, we present a high-level definition of the rule for the main part, which omits the details description of locations and transitions of all the possible output TA for this translation rule. Figure 13 maps the named with the structure of TA define in case 1 of the translation rule. And Example A.6 illustrates using the translation rule A.2.5 in translating a process.

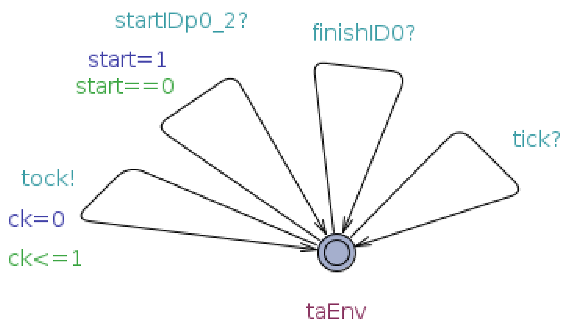
Example A.6. An example that demonstrates using Rule A.2.5 in translating a process `e1->SKIP`

```
transTA e1->SKIP "p04" "0" 0 0 usedNames =
    [
```



```
    ] ++ transTA(SKIP)
   = [
```





```
        ]
```

Example A.6 illustrates using Rule A.2.5 in translating a process `c1->SKIP`, which is translated into a list containing three TA. The first Ta captures the translation of the event `e1` using Rule A.2.5. The second TA captures the translation of the subsequent process `SKIP` using Rule A.3. The last TA is an environment TA for the list of the translated TA.

The details behaviour of the output TA is as follows. Initially, the first TA synchronises on the coordination action `startIDp04`. Then, on location `s2` either the TA performs the event `tock` and remains in the same location `s2` or the TA performs the prefix event `e1` that leads to performing the subsequent flow action `startID01` to activate to activate the second TA, which synchronises on the flow action `startID01`. Then, either the second TA performs the action `tock` and remains in the same location; or `TA1` performs the action `tick` which leads to performing the termination action `finishID` for a successful termination. These two TA describe the translation of process `c1->SKIP`.

A.2.6 Translation of WAIT n

This section describes a translation of process `(WAIT n)`, which defines a delay of at least `n` units time. The section begins with presenting a rule for translating the process `(WAIT n)`, and then follows with an example that illustrates using the rule in translating a process.

Rule A.6. Translation of WAIT n

```
transTA (WAIT 0) processName bid sid fid usedNames =
    transTA SKIP processName bid sid fid usedNames

transTA (WAIT n) processName bid sid fid usedNames =
    transTA (Prefix Tock Wait (n - 1)) [] bid sid fid usedNames
```

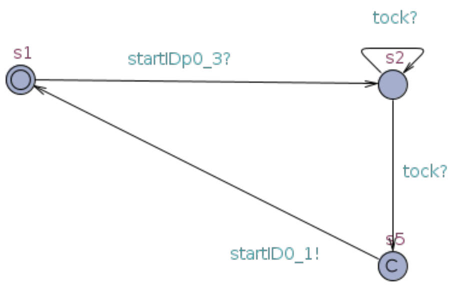
Rule A.6 describes a translation of delay process `WAIT(n)`, which is translated in terms of two constructs: `Prefix` and `SKIP`, previously defined in Rule A.3 and Rule A.2.5, respectively. In the syntax of tock-CSP, this is express as :

```
Wait(n)= tock -> Wait (n-1).
```

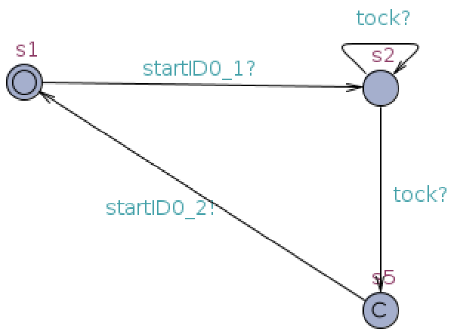
The process `WAIT(n)` is translated into a list of TA, which performs the event `tock` `n` times until the value `n` becomes `0` and the the TA behaves as `SKIP`. The base case is translated according to Rule A.4. While the remaining cases are translated according to Rule A.2.5. The following example illustrates using the rule in translating a process.

Example A.7. An example for translating a delay process `WAIT(2)`, which expresses a delay of 2 units time. The process is translated as follows.

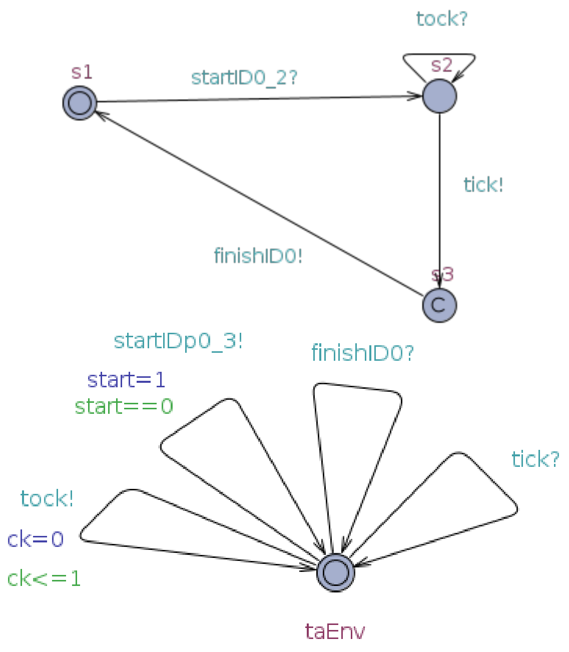
```

transTA (WAIT 2) "p0_3" "0" 0 0
        ([], [], [], [], [], [], [], ([],[])) =
        [
```



```
        ] ++
        transTA (WAIT 1) "" "0" 1 0
                ([], [], [], [], [], [], [], ([],[])) =
                [
```



```
        ] ++
        transTA (WAIT 0) "" "0" 2 0
                ([], [], [], [], [], [], [], ([],[])) =
```



12]

Example A.7 illustrates using Rule A.6 in translating the process `WAIT(2)`. Initially, the example defines the function `transTA` for the construct `(WAIT n)` and its required arguments: process is `WAIT(2)`, process name is ”p0_3”, `branchID` is ”0”, `startID` is 0, `finishID` is 0 and `usedNames` is the remaining empty lists for the collection of names that are empty at the beginning. The translation produces a list of TA `TA0, TA1 and TA2` in the example.

Details behaviour of the output TA is as follows. First, `TA0` synchronises on the flow action `startIDp0_3`, which connects the environment with the first TA in the list of the translated TA. Then, on location `s2`, `TA0` performs the time event `tock` at least ones and then performs the subsequent flow action `startID01`, which connects two `TA0` and `TA1`. In this case, `TA1` is similar to the previous `TA0` because both of them capture the translation of the event `tock`. `TA1` performs the second action `tock` and then performs another flow action `startID02`, which connects `TA1` and `TA2`. Lastly, `TA2` synchronises on the flow action `startID02` and then either `TA2` performs the action `tock` and remains in the same location; or `TA2` performs the action `tick` and then immediately proceeds to a terminating action `finishID0`, which indicates a successful termination of the translated process. These 3 TA describe the translation of the process `WAIT(2)`.

A.2.7 Translation of Waitu n (Strict delay)

This section describes the translation of process `Waitu n`, a strict delay of `n` units time. The section begins with presenting a rule for translating the process `Waitu n`, and then follows with an example that illustrates using the rule in translating a process.

Rule A.7. Translation of Waitu n

```
transTA (Waitu n) procName bid sid fid usedNames =
  ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
  where
    idTA = "taWait_u" ++ bid ++ show sid

    loc1 = Location "id1" "s1" EmptyLabel None
    loc2 = Location "id2" "s2" EmptyLabel None
    locs = [loc1, loc2]

    tran1 = Transition loc1 loc2 ([lab1] ++ t_reset)       []
    tran2 = Transition loc2 loc2 ([lab2] ++ dlguard ++ dlupdate) []
    tran3 = Transition loc2 loc1 ([lab4] ++ dlguard2 ++ t_reset) []
    trans = [tran1, tran2, tran3] ++
            (transIntrpt intrpts loc1 loc2)

    (_, _, _, _, _, intrpts, _, _) = usedNames

    lab1 = Sync (VariableID (startEvent procName bid sid) []) Ques
    lab2 = Sync (VariableID "tock"  []) Ques
    lab4 = Sync (VariableID ("finishID" ++ show fid)   []) Excl

    dlguard  = [(Guard (BinaryExp (ExpID "tdeadline") Lth (Val n)))]
    dlupdate = [(Update
                 [(AssgExp (ExpID "tdeadline") AddAssg (Val 1))] )]

    -- A guard for exiting a strict delay
    dlguard2 = [(Guard (BinaryExp (ExpID "tdeadline") Equal (Val n)))]

    -- reset deadline time
    t_reset = [(Update [(AssgExp (ExpID "tdeadline")
                             ASSIGNMENT (Val 0)) ] )  ]
```

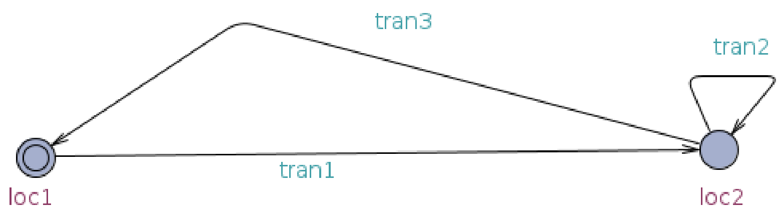


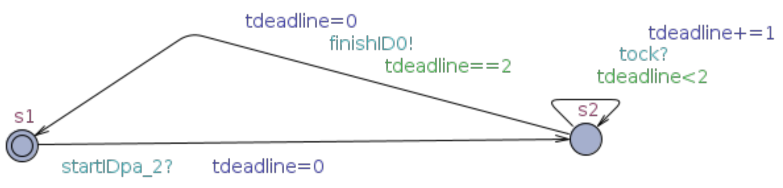
Figure 14: A structure of a TA for a translation of strict delay.

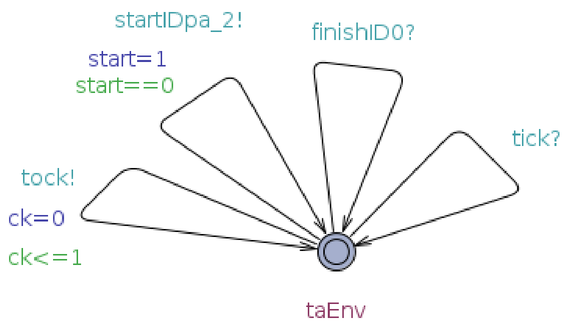
Rule A.7 describes a translation of strict delay. In Figure 14, we annotates the structure of the output TA with the names used in the translation rule. The structure of the TA has 2 locations and 3 transitions as defined in Lines 6–8 and Lines 10–14. Then, Line 16 extracts the used names for interrupt. And Lines 18–20 defines the labels of the transitions. Lines 22–27 defines the guards for controlling the deadlines. Finally, Lines 30–31 reset the deadline in case of translating process that has recursive calls.

The behaviour of the output TA begins on transition `tran1`, where the TA synchronises on a flow action (define in Line 18). Then, on Location `loc2` (define in Line 6), either `TA` follows transition `tran2` or `tran3`. On transition `tran2` (define in Line 10), the TA checks the delay guard `dlguard` (defined in Line 22), if it is true the TA performs the time event `tock` and update the delay timer with the expression `dlupdate` (Lines 23–24). Alternatively, if the guard `dlguard` is false, the second guard `dlguard2` becomes true (Line 27), which enables the `TA` to perform the next flow action on transition `tran3`, as well as resetting the timer in the expression `t_reset` (Lines 30–31). This transition completes the behaviour of the translated TA. The following Example A.8 illustrates using the rule in translating a process.

Example A.8. An example for translating a process `Waitu(2)` a strict delay of 2 units time.

```
transTA (Waitu 2) "p0_3" "0" 0 0
        ([], [], [], [], [], [], [], ([],[]))
        ⇝ [
```





]

Example A.8 illustrates using Rule A.7 in translating a process `Waitu 2`. The example translates the process `Waitu 2` into a list of TA shown in the above list of TA. In the beginning, the example applies the function `transTA` on the required parameters:

process is `Waitu 2`, process name is `p0_3`, `branchID` is "0", `startID` is 0, `finishID` is 0, `usedNames` is a tuple of empty elements, each rule begins with empty used names. As the translation goes on we build a collection of the names used in the translation.

The resulting output TA for the translation is shown in the above figures, which illusrate two TAs. Initially, the TA synchronises on the coordinating flow action `startIDp0_3` and then performs the action `tock` twice, which disables the first guard `(tdeadline<2)` and enables the second guard `(tdeadline==2)`. Finally, the TA performs the termination action `finishID0`. This completes the description of an example for translating the process `Waitu 2` into TA.

A.2.8 Translation of Internal Choice

This section describes a translation of operator for Internal choice. The section begins with presenting a rule for translating the operator internal choice, and then follows with an example that illustrates using the rule in translating a process.

Rule A.8. Translation of Internal Choice

```
transTA (IntChoice p1 p2) procName bid sid fid usedNames =
   ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
       (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
   where
      idTA  = "taIntCho" ++ bid ++ show sid

      loc1 = Location "id1" "s1" EmptyLabel None
      loc2 = Location "id2" "s2" EmptyLabel CommittedLoc
      loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
      loc4 = Location "id4" "s4" EmptyLabel CommittedLoc
      locs = [loc1, loc2, loc3, loc4]

      tran1 = Transition loc1 loc2 [lab1] []
      tran2 = Transition loc2 loc3 []     []
      tran3 = Transition loc2 loc4 []     []
      tran4 = Transition loc3 loc1 [lab4] []
      tran5 = Transition loc4 loc1 [lab5] []
      trans = [tran1, tran2, tran3, tran4, tran5]

      lab1 = Sync (VariableID (startEvent procName bid sid) [])
                  Ques
      lab4 = Sync (VariableID ("startID" ++ (bid ++ "0") ++
                                show (sid+1)) [])
                  Excl
      lab5 = Sync (VariableID ("startID" ++ (bid ++ "1") ++
             show (sid+2)) []) Excl

      -- translation of RHS and LHS processes
      (ta1, sync1, syncMap1) =
            transTA p1 [] (bid ++ "0") (sid+1) fid usedNames
      (ta2, sync2, syncMap2) =
            transTA p2 [] (bid ++ "1") (sid+2) fid usedNames

```

Internal choice is a binary operator that combines two processes P1 and P2. Rule A.8 translates the operator of internal choice into a TA that coordinates a list of translated TA Tp1 and Tp2 for the translation of the two processes P1 and P2 respectively, that are composed with internal choice operator.

In Figure 15, we annotate the structure of the output TA with the names used in

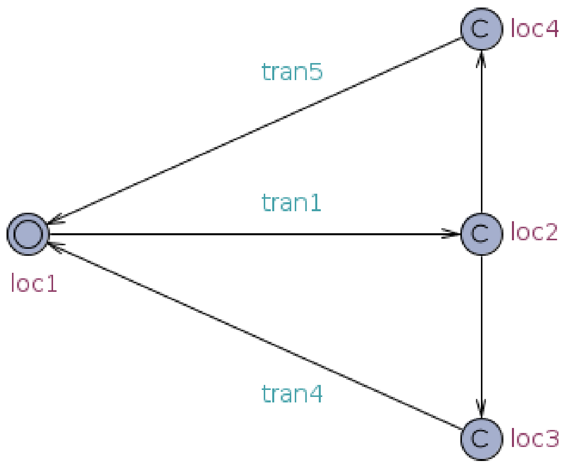


Figure 15: A structure of a TA for translating Internal choice.

the translation rule. The output TA begins on transition `tran1` for performing a flow action that connects the TA with the network of the TA. Then, the output TA follows one of the two silent transitions that lead to transition `tran4` and `tran5` respectively. On transition `tran4` the TA activates the list of TA `Tp1` and on transition `tran5` the TA activates the list of TA `Tp2`.

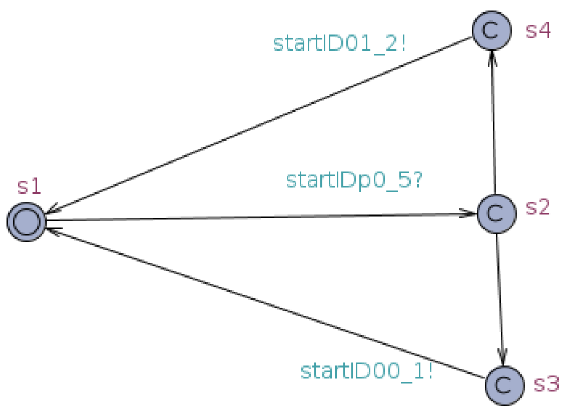
Details of Rule A.8 is as follows. Line 1 defines the function `transTA` for the construct `IntChoice` and the 5 required parameters for this rule. Line 2 describes the output tuple that contains 3 elements, a list of translated TA, a list of synchronisation actions and a list of identifiers for identifying each synchronisation action.

The output TA has 4 locations and 5 transitions, as define Lines 7–11 and Lines 13–18 respectively. Lines 20–26 define the label of the transitions. Lines 28–31 defined the subsequent translation of the processes `P1` and `P2`.

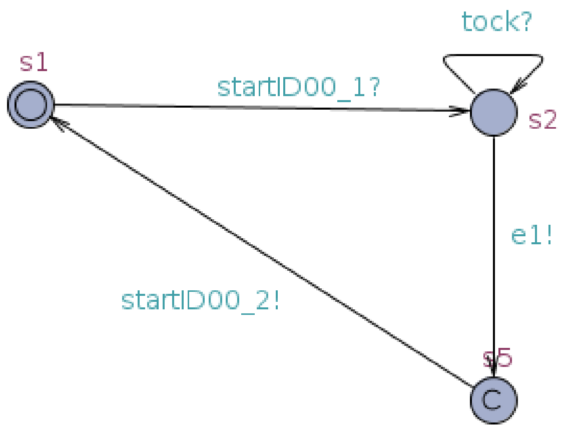
The behaviour of the output TA begins with a flow action (defined in Line 20). Then, on location `loc2`, the TA follows one of the two silent transitions, that is either `tran2` or `tran3`. Transition `tran2` leads to transition `tran4`, where the TA performs another flow action (Line 22) that activates `Tp1`. While transition `tran3` leads to transition `tran5`, where the TA performs a flow action (define Line 25) that activates `Tp2`. The following Example A.9 illustrates using this Rule A.8 in translating a process.

Example A.9. An example for translating a process that compose two process with internal choice.

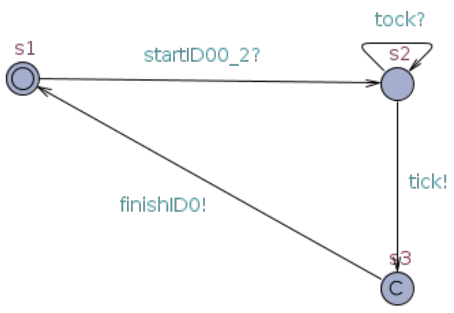
```
1 transTA((e1->SKIP)|-|(e2->SKIP)) =
2     [
```



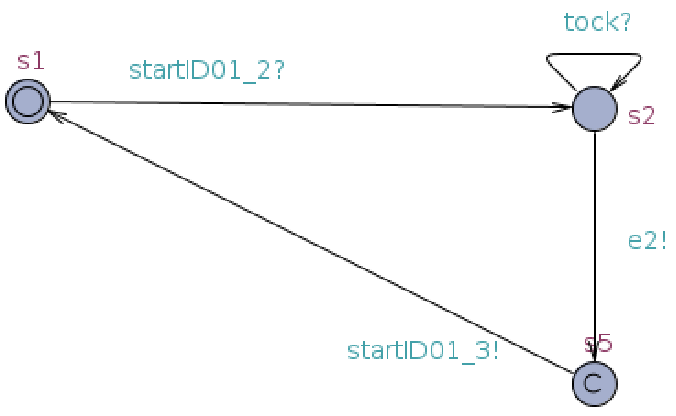
```
    ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



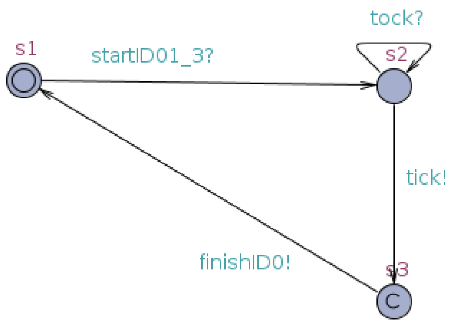
```
     ] ++ transTA(SKIP)
         = [
```

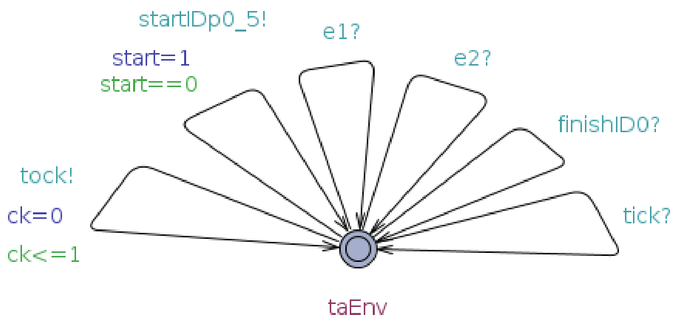


```
      ]

ta2 = transTA(e2->SKIP)
      = [
```



```
      ] ++ = transTA(SKIP)
           = [
```





```
    ]
```

Example A.9 translates the process `((e1->SKIP)|-|(e2->SKIP))` into a list containing 5 TA. The first TA is a translation of the operator internal choice. Second and third TA are translations of the LHS process `(c1 -> SKIP)`. Where the second TA is a translation of the prefix event `e1` using Rule A.2.5. And the third TA is a translation of the subsequent process `SKIP` using Rule A.3. Fourth and fifth TA are translation of the RHS process `(e2 -> SKIP)`. Fourth TA is a translation of the prefix event `e2` using Rule A.2.5. While, the fifth TA is a translation of the subsequent process `SKIP` using Rule A.3. Finally the last TA is an environment TA for list of the translated TA. This completes the list of the output TA that capture the behaviour of the process `((e1->SKIP)|-|(e2->SKIP))`.

A.2.9 Translation of External Choice

This section describes the translation of the construct External Choice. The section begins with presenting a rule for translating the operator for external choice and then follows with an example that illustrates using the rule in translating a process.

Rule A.9. Translation of External Choice

```
transTA (ExtChoice p1 p2) procName bid sid fid usedNames =
   ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
   (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
   where
      idTA  = "taIntCho" ++ bid ++ show sid

      loc1  = Location "id1" "s1" EmptyLabel None
      loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
      loc3  = Location "id3" "s3" EmptyLabel CommittedLoc
      locs  = [loc1, loc2, loc3]

      tran1 = Transition loc1 loc2 [lab1]
      tran2 = Transition loc2 loc3 [lab2]
      tran3 = Transition loc3 loc1 [lab3]
      trans = [tran1, tran2, tran3]

      lab1  = Sync (VariableID (startEvent procName bid sid) []) Ques
      lab2  = Sync (VariableID
                     ("startID" ++ (bid ++ "0") ++ show (sid+1)) [])
                   Excl
      lab3  = Sync (VariableID
                     ("startID" ++ (bid ++ "1") ++ show (sid+2)) [])
                   Excl

      -- Extract a list of names for external choice from the
      -- parameter usedNames.
      (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
               excps) = usedNames

     -- Updates the used names for subsequent translation
      exChs'  = exChs ++ (initials p2)
      exChs'' = exChs ++ (initials p1)
      usedNames'  = (syncEv, syncPoint, hide, rename, exChs', intrr,
                     iniIntrr, excps)
      usedNames'' = (syncEv, syncPoint, hide, rename, exChs'', intrr,
                     iniIntrr, excps)

      -- translation of RHS and LHS processes p1 and p2
      (ta1, sync1, syncMap1) =
                transTA p1 [] (bid ++ "0") (sid+1) fid usedNames'
      (ta2, sync2, syncMap2) =
                transTA p2 [] (bid ++ "1") (sid+2) fid usedNames''
```

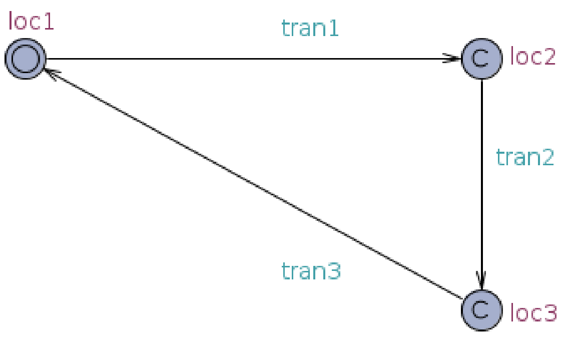


Figure 16: A structure of the control TA for the translation of external choice.

Rule A.9 defines a translation of external choice. The operator of external choice ([]) is another binary operator that combines two processes `P1` and `P2`. Rule A.9 translates the operator external choice into a TA that coordinates a lists of translated TA: `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2`, respectively.

In Figure 16, we annotate the structure of the output TA with the names used in the translation rule. The output TA has 3 transitions and 3 locations define in Lines 7–10 and 12–15, respectively. Then, Lines 17–21 define the corresponding labels of the transitions. Lines 27 extracts the initials of the external choice, and then updates the initials in Lines 31–36. Finally, Lines 39–42 define the subsequent translation of processes `P1` and `P2`, which produces list of TA `Tp1` and `Tp2`, respectively.

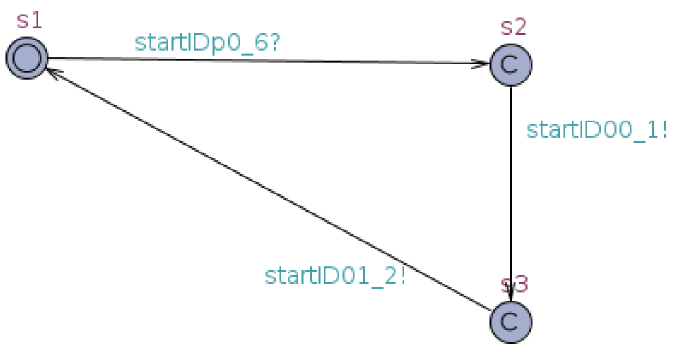
The behaviour of the output TA begins on transition `tran1` with performing a flow action (define in Line 12). Then, on both transition `tran2` (Line 13) and `tran3` (Line 14) the TA performs two additional flow actions that activate two list of TA: `Tp1` and `Tp2` define in Lines 39–34 and 41–42 respectively. Thus, the output TA activates both `Tp1` and `Tp2` simultaneously, which makes the behaviour of both `Tp1` and `Tp2` available to the environment, such that choosing one of the translated list of TA blocks the other alternative list of TA. That is, choosing `Tp1` blocks `Tp2`, likewise choosing `Tp2` blocks `Tp1`.

An important part of translating external choice is translating both processes such that choosing one process blocks the behaviour of the other process. This is achieved with additional transitions in the first TA of both processes for the external choice as discussed in the overview of the translation technique. In the parameters of the translation function `transTA`, the parameter `usedNames` has a name for the initials of the processes for external choices, which is updated in Lines 31–36, and the used in translating each process, specifically in constructing the transition of blocking external choice that has co-actions (initials of the other process) for blocking the process that is not chosen by the environment. The following Example A.10 illustrates using this rule in translating a process that composes processes with the operator of external choice.

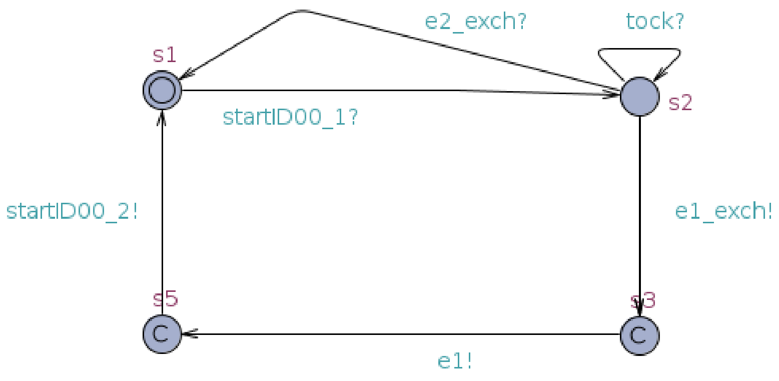
Example A.10. An example of translating a process that composes two processes with the operator of external choice.

```
transTA((e1->SKIP)[](e2->SKIP))
```

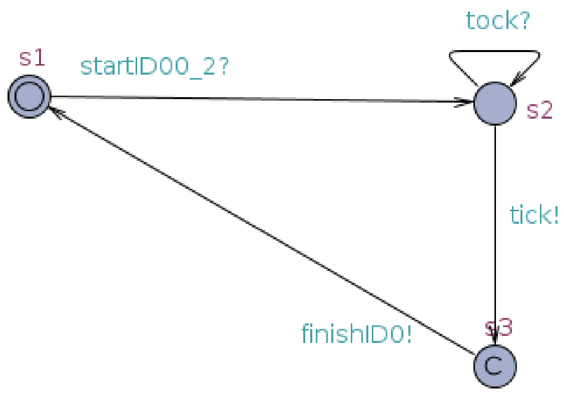
```
  = [
```



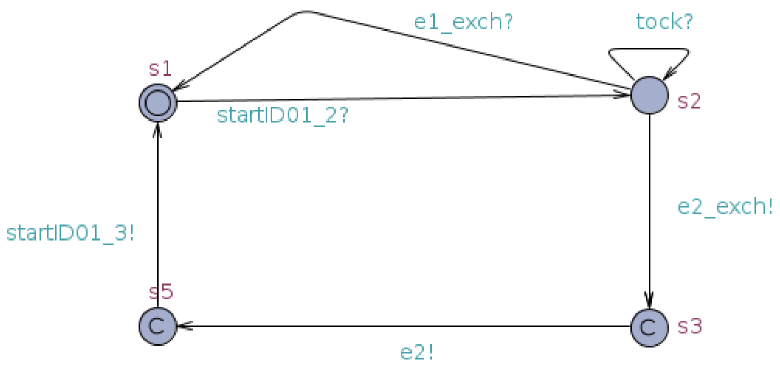
```
         ] ++ ta1 ++ ta2
where
ta1 = transTA(e1->SKIP)
    = [
```



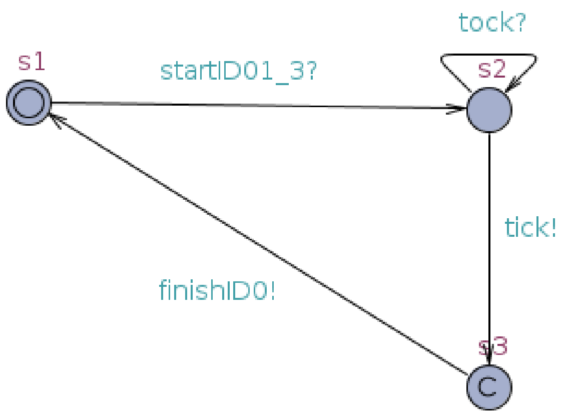
```
        ] ++ transTA(SKIP)
          = [
```

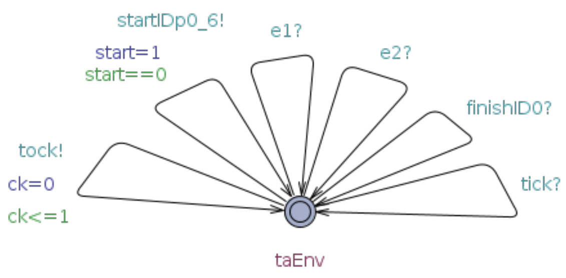


```
         ]

  ta2 = transTA(e2->SKIP)
      = [
```



```
         ] ++ transTA(SKIP)
            = [
```





```
]
```

Example A.10 illustrates using Rule A.9 in translating a process ((e1->SKIP)[](e2->SKIP)) into a list of TA that contains 5 TA. The first TA is a translation of the operator external choice. The TA has 3 transitions, each label with a flow action, startIDp0_6 startID00_1 and startId01_2. Initially, the behaviour of the TA synchronises on the first flow action startIDp0_6 and then immediately performs the two subsequent flow actions startID00_1 and startId01_2 that activate the translations of the LHS and RHS processes, (e1->SKIP) and (e2->SKIP) respectively.

Second and third TA are translations of the LHS process e1->SKIP. Second TA is a translation the event e1. which synchronises on the flow action startID00_1 and moves to location s2 where the TA has 3 possible transitions: e1_exch? e2_exch? and tock?. On transition tock? the TA performs the action tock for the progress of time. On transition e2_exch? the TA performs a blocking event when the environment chooses the other action e2. Lastly, on transition e1_exch! the TA performs the action e1_exch! when the environment chooses the action e1 for the behaviour Tp1. First, the action e1_exch! synchronise with its co-action e1_exch? to block the alternative behaviour of Tp2, and then immediately proceeds with performing the chosen action e1 that leads to the subsequent flow action startID00_2, which activates the subsequent TA third TA. The third TA is a translation of the subsequent process SKIP for the LHS process e1->SKIP which is translated with Rule A.3.

Fourth and Fifth TA are translations of the RHS process (e2->SKIP). Fourth TA is a translation the event e2 using Rule A.2.5, similar to the previous translation of the first TA. While, the fifth TA is a translation of the remaining process SKIP using Rule A.3. Finally, the last TA is an environment TA for the list of the translated TA, as defined in the overview of translation technique. This completes the description of translating the process (e1->SKIP)[](e2->SKIP) into a list of TA.

A.2.10 Translation of Sequential Composition

This section describes a translation of operator sequential composition. The section begins with presenting a rule for translating the operator sequential composition. And then follows with an example that illustrates using the rule in translating a process.

Rule A.10. Translation of Sequential Composition

```
transTA (Seq p1 p2) procName bid sid fid usedNames =
    ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
    (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
    where
      idTA = "taSequen" ++ bid ++ show sid

      loc1  = Location "id1" "s1" EmptyLabel None
      loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
      loc3  = Location "id3" "s3" EmptyLabel None
      loc4  = Location "id4" "s4" EmptyLabel CommittedLoc
      locs  = [loc1, loc2, loc3, loc4]

      tran1 = Transition loc1 loc2 [lab1] []
      tran2 = Transition loc2 loc3 [lab2] []
      tran3 = Transition loc3 loc4 [lab3] []
      tran4 = Transition loc4 loc1 [lab4] []
      trans = [tran1, tran2, tran3, tran4]

      lab1  = Sync (VariableID
                      (startEvent procName bid sid) [])
                   Ques
      lab2  = Sync (VariableID
                       ("startID" ++ (bid ++ "0") ++ show (sid+1)) [])
                   Excl
      lab3  = Sync (VariableID ("finishID" ++ show (fid+1)) [])
                   Ques
      lab4  = Sync (VariableID
                       ("startID" ++ (bid ++ "1") ++ show (sid+2)) [])
                   Excl

        -- translation of the LHS process
      (ta1, sync1, syncMap1) =
            transTA p1 [] (bid ++ "0") (sid+1) (fid+1) usedNames

       -- translation of the RHS process
      (ta2, sync2, syncMap2) =
             transTA p2 [] (bid ++ "1") (sid+2) fid usedNames
```

This operator sequential composition is another binary operator that composes two processes `P1` and `P2` sequentially. Like the previous translation rules, this rule translates the operator sequential composition into a control TA, which coordinates the list

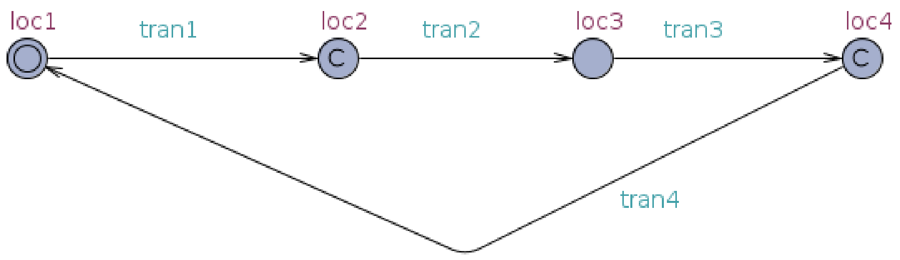


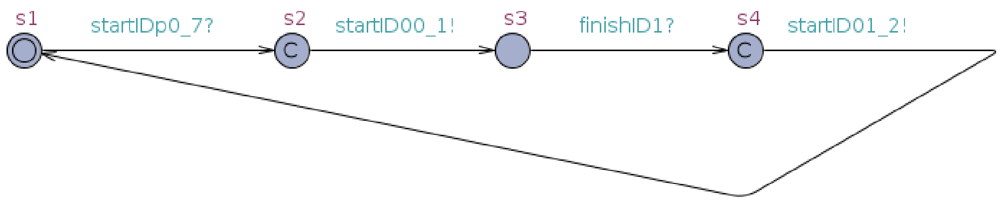
Figure 17: A structure of the control TA for the translation of the operator for sequential composition.

of TA `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2`.

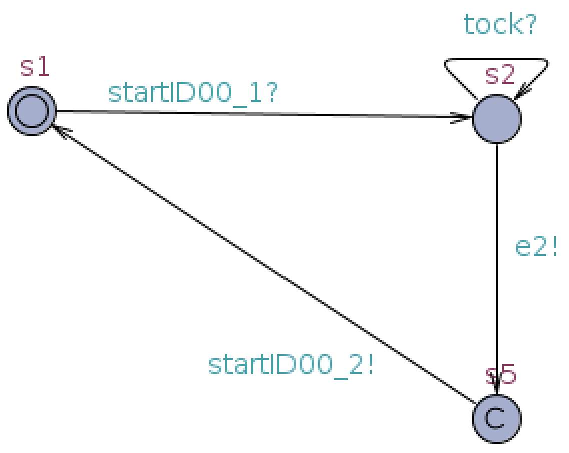
In Figure 17, we annotate the structure of the output TA with the names used in the translation Rule A.10. The TA has four locations and four transitions that are defined in Lines 7–11 and Lines 13–17, respectively. In Rule A.10, the behaviour of the output TA begins with synchronising on the first flow action (Line 13) and then immediately performs another two flow action on transition `tran2` (Line 14) to activate the translation of the LHS process `Tp1`. After that, the control TA waits on location `loc3` until the TA synchronises on a terminating action on transition `tran3` (Line 15), which indicates the termination of the first process `Tp1`, and then immediately activates `Tp2` which proceeds up to its termination point. The following Example A.11 illustrates using this rule in translating a process.

Example A.11. An example for translating a process that composes two processes with the operator for sequential composition.

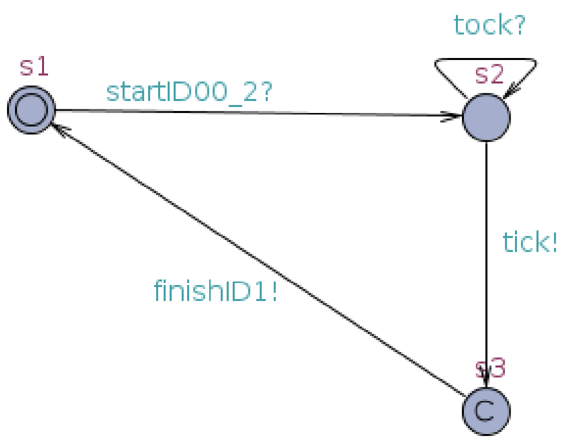
```
transTA((e2->SKIP);(e1->SKIP)) = [
```



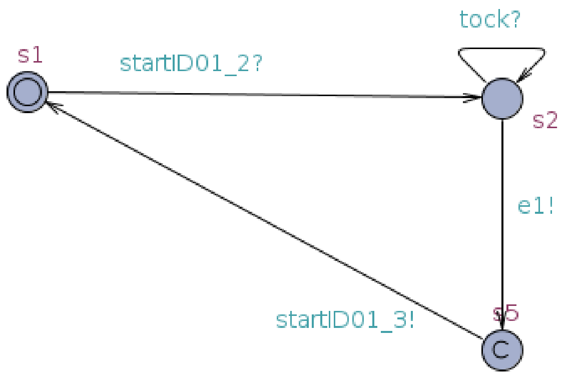
```
    ] ++ ta1 ++ ta2

where
ta1 = transTA(e2->SKIP)
    = [
```



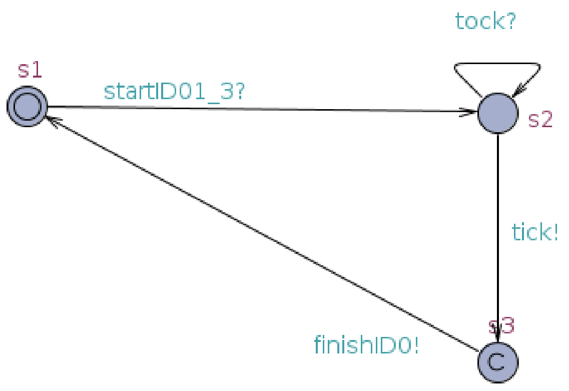
```
  ] ++ transTA(SKIP)
      = [
```

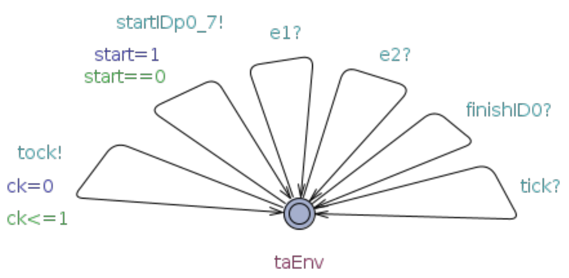


```
             ]

ta2 = transTA(e1->SKIP)
      = [
```



```
      ] ++ = transTA(SKIP)
           = [
```





```
             ]
```

Example A.11 illustrates using Rule A.10 in translating the process ((e2-> SKIP);(e1-> SKIP)) into a list of TA that contains 5 TA. The first TA is a translation of the operator sequential composition using Rule A.10. The second and third TA are translation of the LHS process (e2-> SKIP), while the fourth and fifth TA are translations of the RHS process. Finally, the last TA is an environment TA for the list of the translated TA.

Details behaviour of the list of the translated TA is as follows. The first TA synchronises on the flow action `startIDp0_7?` and then immediately performs another flow action `startID00_1` to activate the second TA, and then waits on location `s3` until the TA synchronises on the termination action `finishID1?`; which indicates the termination of the LHS process (`(e2-> SKIP)`), and then immediately the TA performs the subsequent flow action `startID01_2` that activates the fourth TA for the translation of the RHS process. In the like manner, the fourth TA synchronises on the flow action and then performs the action `e1`, which follows with the subsequent flow action `startID01_3` that activates the fifth TA, which synchronises on the flow action and then performs the action `tick`, and then follows with a termination action `finishID0`, which synchronises with the co-action in the environment TA to indicate a successful termination of the whole process. This completes the description of translating the process `((e2-> SKIP);(e1-> SKIP))` into a list of TA.

A.2.11 Translation of Generallised Parallel

This section describes a translation of operator generallised parallel. The section begins with a rule for translating the operator generallised parallel. And then follows with an example that illustrates using the rule in translating a process.

Rule A.11. Translation of Generallised Parallel

```
transTA (GenPar p1 p2 es) procName bid sid fid usedNames =
    ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
    (es ++ sync1 ++ sync2), (syncMap1 ++ syncMap2) )
    where
      idTA   = "taGenPar" ++ bid ++ show sid
      loc1   = Location  "id1" "s1" EmptyLabel None
      loc2   = Location  "id2" "s2" EmptyLabel CommittedLoc
      loc3   = Location  "id3" "s3" EmptyLabel CommittedLoc
      loc4   = Location  "id4" "s4" EmptyLabel CommittedLoc
      loc5   = Location  "id5" "s5" EmptyLabel None
      loc6   = Location  "id6" "s6" EmptyLabel None
      loc7   = Location  "id7" "s7" EmptyLabel None
      loc8   = Location  "id8" "s8" EmptyLabel CommittedLoc
      locs   = [loc1, loc2, loc3, loc4, loc5, loc6, loc7, loc8]
      tran1  = Transition loc1 loc2 [lab1] []
      tran2  = Transition loc2 loc3 [lab3] []
      tran3  = Transition loc3 loc5 [lab2] []
      tran4  = Transition loc2 loc4 [lab2] []
      tran5  = Transition loc4 loc5 [lab3] []
      tran6  = Transition loc5 loc6 [lab4] []
      tran7  = Transition loc5 loc7 [lab5] []
      tran8  = Transition loc6 loc8 [lab5] []
      tran9  = Transition loc7 loc8 [lab4] []
      tran10 = Transition loc8 loc1 [lab6] []
      trans  = [tran1, tran2, tran3, tran4, tran5,
                tran6, tran7, tran8, tran9, tran10]
      lab1 = Sync (VariableID (startEvent procName bid sid) []) Ques
      lab2 = Sync (VariableID ("startID" ++ (bid ++ "0") ++
                               show (sid+1              )) []) Excl
      lab3 = Sync (VariableID ("startID" ++ (bid ++ "1") ++
                               show (sid+2              )) []) Excl
      lab4 = Sync (VariableID ("finishID" ++ show (fid+1 )) []) Ques
      lab5 = Sync (VariableID ("finishID" ++ show (fid+2 )) []) Ques
      lab6 = Sync (VariableID ("finishID" ++ show  fid    ) []) Excl

      (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
               excps) = usedNames
      syncEv'    = es ++ syncEv               -- Update synch name
      usedNames' = (syncEv', syncPoint, hide, rename, exChs, intrr,
                    iniIntrr, excps)
      (ta1, sync1, syncMap1) =
               transTA p1 [] (bid ++ "0") (sid+1) (fid+1) usedNames'
      (ta2, sync2, syncMap2) =
               transTA p2 [] (bid ++ "1") (sid+2) (fid+2) usedNames'
```

The operator generalised parallel is another binary operator, which composes two processes P1 and P2 that run in parallel and synchronise on a specified set of synchronisation events. The construct GenPar is translated into two TA: a control TA and synchronisation TA. The synchronisation TA (Definition A.2.11) coordinates the synchronisation of the translated processes Tp1 and Tp2. While, the control TA coordinates the behaviour of the translated processes Tp1 and Tp2.

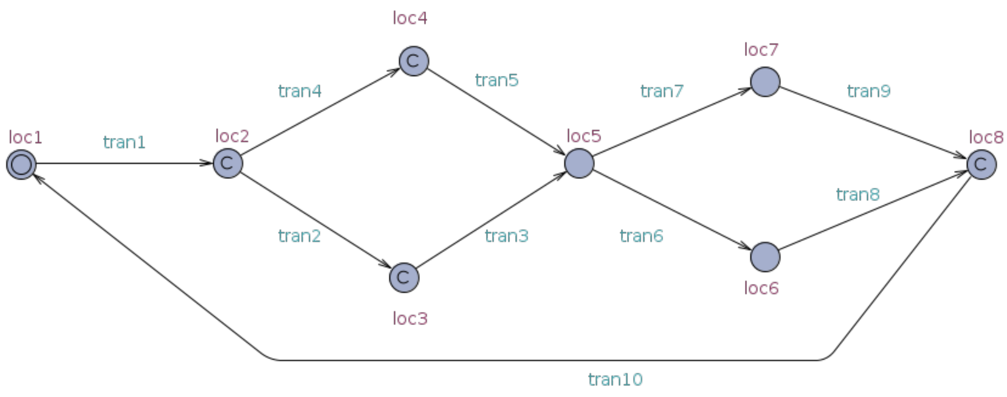


Figure 18: A structure of the control TA for the translation of operator Generallised Parallel

In Figure 18, we annotate the structure of the control TA with the names used in the translation rule. The structure of the control TA has 8 locations and 10 transitions define in Lines 6–14 and Lines 15–26 respectively. Lines 27–34 define the labels of the transitions. Lines 36–37 extracts the used names from the parameter usedNames for updating the list of synchronisation names syncEv in Lines 38–40. Then, Lines 41–44 is a recursive call for the translation of the processes P1 and P2, which produces the list of TA, Tp1 and Tp2.

The behaviour of the control TA begins on tran1 for synchronising on a flow action and then immediately performs another two flow actions to activate both Tp1 and Tp2 simultaneously, that is on tran2 and tran3 or tran3 and tran4 in both two possible orders depending on the environment, that is either Tp1 simultaneously with Tp2 or Tp2 simultaneously with Tp1. Then, the control TA waits on location s4 until either Tp1 or Tp2 terminates and then waits for the other TA to terminate, depending on the first process that terminates, either Tp1 and then Tp2 or Tp2 and then Tp1. After that, the control TA immediately performs another termination action, which records the termination of the whole process.

The translated processes synchronise on a multi-synchronisation action, which synchronises more than two translated TA: at least two translated processes and the Environment TA. As highlighted in the overview of the translation technique, for describing the strategy we use in handling multi-synchronisation. We adopt a centralised approach [31, 11] for using a separate controller in handling multi-synchronisation. In this work, we implement the approach in a functional style with Haskell. Definition

A.2.11 defines a function the control TA `syncTA` that coordinates multi-synchronisation action for synchronising multi-synchronisation action.

In translating the multi-synchronisation, each translated process that participates in multi-synchronisation has a client TA, which synchronises on a multi-synchronisation action. The client TA sends a synchronisation request to the synchronisation controller TA `syncTA` and then waits for a synchronisation response.

When the synchronisation controller receive all the synchronisation request, which indicates that all the synchronisation client are ready for the synchronisation, then a guard for performing the multi-synchronisation action becomes enable, and `syncTA` communicates the multi-synchronisation action to the environment and then also immediately broadcast the multi-synchronisation action that responds to all the awaiting client TA.

On receiving the broadcast multi-synchronisation response, all the awaiting client TA synchronise and proceed. An example of using this rule in translating a process is illustrated in the following Example A.12, which demonstrates using the rule in translating a concurrent processes that are composed with the operator generalised parallel `GenPar`.

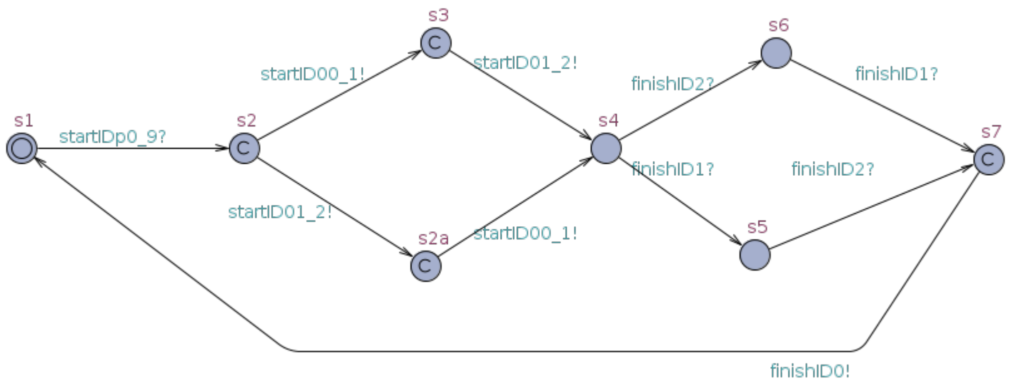
Synchronisation TA

```
syncTA :: [Event] -> [SyncPoint] -> TA
syncTA     events     syncMaps   =
    TA "SyncTA" [] [] (loc:locs) [] (Init loc) trans
    where
      loc  = Location "SyncPoint" "SyncPoint" EmptyLabel None
      locs = [(Location ("s"++ show e) ("s"++ show e) EmptyLabel
                        CommittedLoc) | e <- uniq events]
      trans = transGen  loc (uniq events) syncMaps events

-- Generates transitions for the sync controller
    transGen :: Location->[Event]->[SyncPoint]->[Event]->[Transition]
    transGen    l0        []        _            _     = []
    transGen    l0        (e:es)   syncMaps     syncs  =
        [(Transition
           l0 l
           (Sync (VariableID (show e) []) Excl),
           (Guard
              (ExpID
                ((addExpr
                  [("g_" ++ tag)|(e1, tag) <- syncMaps, e == e1 ])
                  ++ " == " ++
                  show ((length [e1 | e1 <- syncs, e == e1 ]) + 1)
                )
              )
           ),
           (Update ([ AssgExp (ExpID ("g_" ++ tag))
                     ASSIGNMENT (Val 0) |(e1, tag) <- syncMaps,
                     e == e1] )) ] [])]
        ++
        [Transition
            l l0
            [(Sync (VariableID ((show e) ++ "___sync") []) Excl)] []
        ] ++ (transGen l0 es syncMaps syncs)
        where
          l = (Location ("s"++ show e) ("s"++ show e)
                        EmptyLabel CommittedLoc)

```

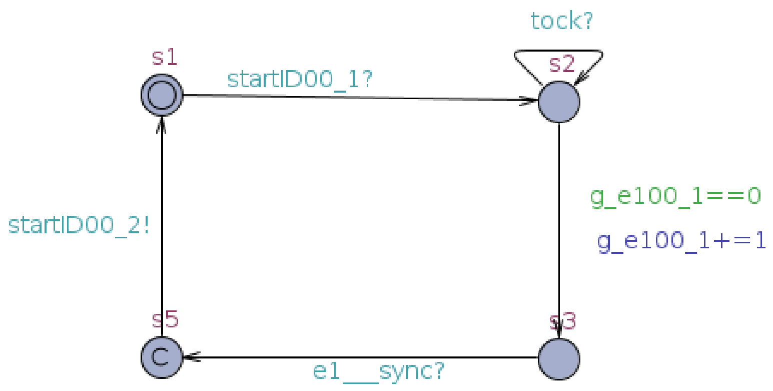
In Definition A.2.11, we define a function for the synchronisation TA `syncTA` which takes 2 parameters, a list of synchronisation actions and a list of pairs that assign an identifier to each synchronisation action. The output of the function is a TA with an identifier "syncTA" (Line 3). The output TA has one starting location and one location for each synchronisation action as defined in Lines 4–7. Then, Line 8 defines a function for generating the transitions of the TA. Each synchronisation action has two transitions one from the initial location and the second transition back to the initial location. The first transition has a guard that is only enabled when all the synchronisation TA becomes ready for the synchronisation. This is illustrated in Example A.12.

Example A.12. An example that illustrates using both Rule A.11 and the definition of synchronisation controller in translating a process.

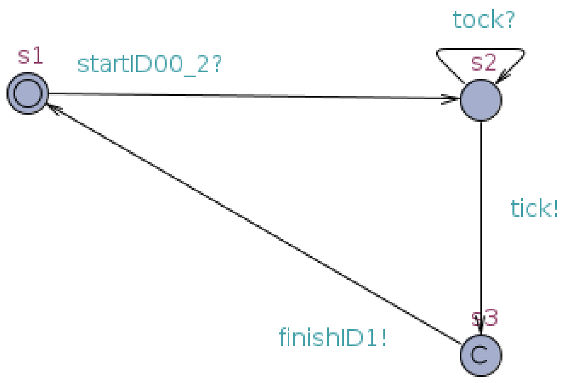
```
transTA((e1->SKIP)[|{e1}|](e1->SKIP)) = [
```



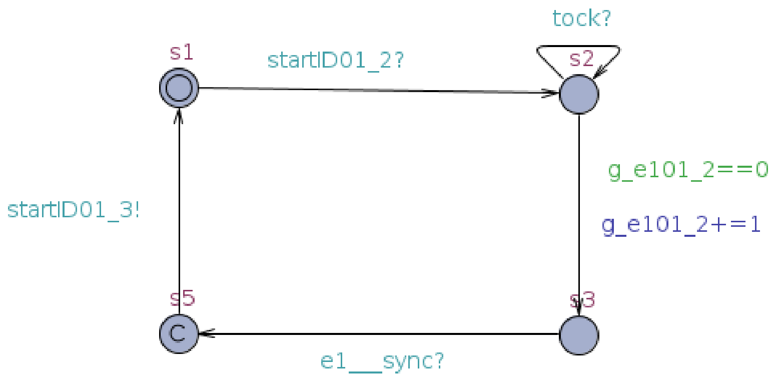
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



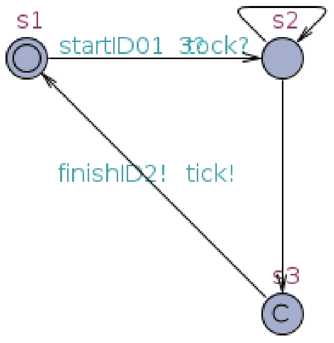
```
  ] ++ transTA(SKIP)
     = [
```

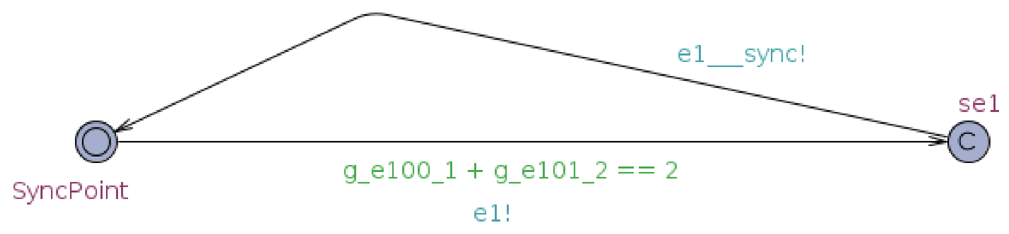


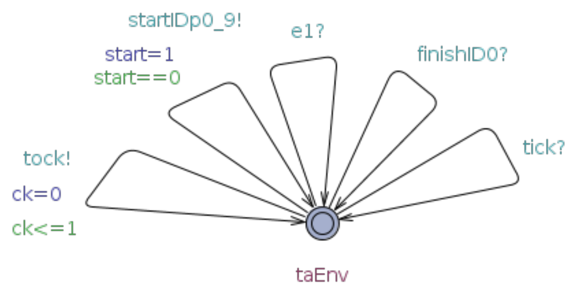
```
              ]

ta2 = transTA(e2->SKIP)
    = [
```



```
         ] ++ transTA(SKIP)
              = [
```







]

Example A.12 illustrates using Rule A.11 in translating the process `(e1->SKIP)[|{e1}|](e1->SKIP)` into a list of TA that contains 7 TA. The first TA captures the translation of the operator generalised parallel. The second and third TA captures the translation of the LHS process. The fourth and fifth TA captures the translation of the RHS process. The sixth TA is a synchronisation TA that coordinates the synchronisation of the action `e1`. Finally, the last TA is an environment TA for the list of the translated TA for the process `(e1->SKIP)[|{e1}|](e1->SKIP)`.

The behaviour of the translated TA is as follows. The first TA is the control TA that initially synchronises on the first flow action `startIDp0_9` and then immediately performs two flow actions in two possible orders, either `startID00_1` and then `startID01_2` or `startID01_2` and then `startID00_1`, depending on the environment. Then the control TA waits on location `s4` until the control TA synchronises on the termination action `finishID2` and then synchronise on the second termination action `finishID1`, for the LHS and RHS processes respectively. Alternatively the control TA synchronises first on `finishID1` and then synchronise on the termination action `finishID2`, depending on the process that terminates first, either the LHS process or the RHS process.

The second TA synchronises on the flow action `startID00_1` and then updates its guard to indicates its readiness to synchronise on the multi-synchronisation action `e1` . Then, on receiving a response for the synchronisation, the TA synchronises on the broadcast multi-synchronisation action `e1___sync?`, which enables the TA to proceed with performing another flow action that activated the third TA, which captures the translation of the subsequent process `SKIP` as described in Rule A.3. In similar manner, the fourth and fifth TA captures the translation of the RHS process `(e1->SKIP)`

A.2.12 Translation of Interleaving

This section describes the translation of operator interleaving. The section begins with presenting a rule for translating the operator interleaving and then follows with an example that illustrates using the rule in translating a process.

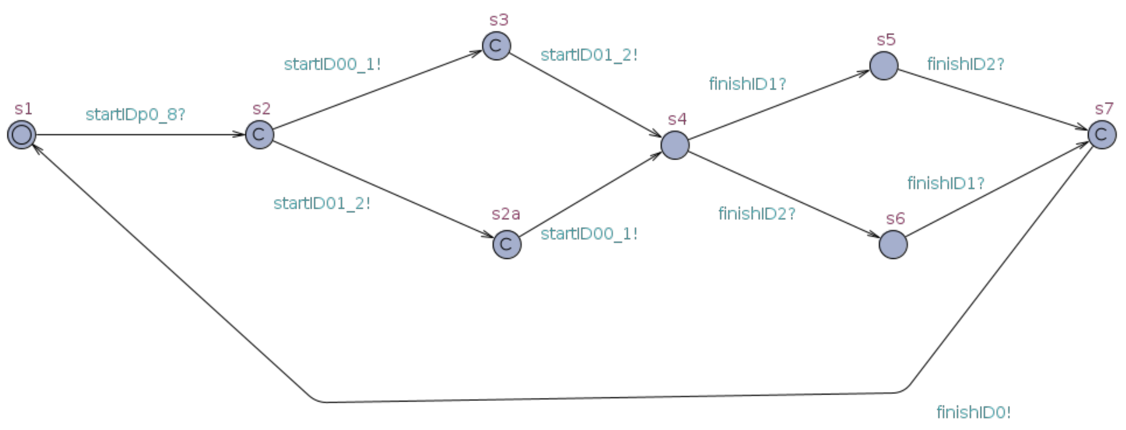
Rule A.12. Translation of Interleaving

```
1 transTA (Interleave p1 p2)     procName bid sid fid usedNames
2   = transTA (GenPar  p1 p2 []) procName bid sid fid usedNames
3             -- As generalised parallel with empty synch events
4
```

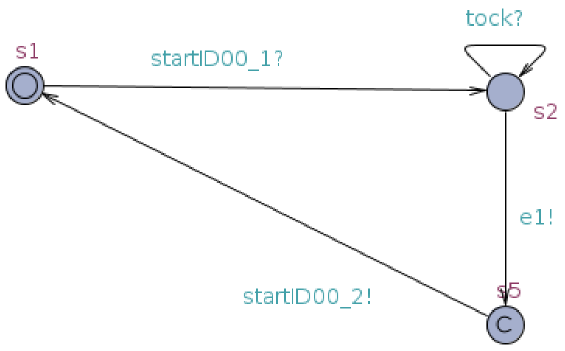
The operator interleaving is translated in terms of the constructor for generalised parallel with empty synchronisation events. In tock-CSP, this is expressed as `(P1 ||| P2) = (P1 |[{}]| P2)`. Line 1 defines the function `transTA` for the construct `Interleave` and the required parameters. While line 2 defines the output TA in terms of the construct for the generalised parallel `GenPar` with empty synchronisation events. The following example illustrates using the rule in translating a process.

Example A.13. An example of translating a tock-CSP process that composes processes with the operator interleaving using Rule A.12.

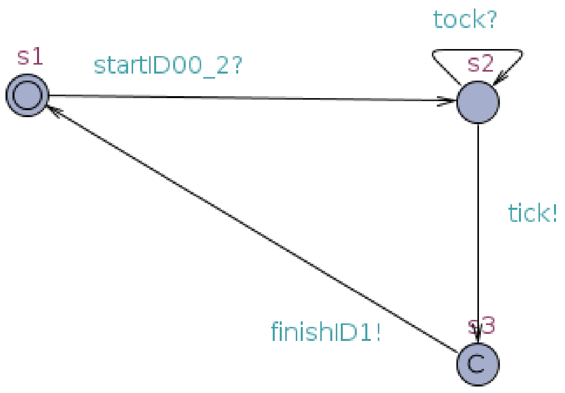
```
   transTA((e1->SKIP)|||(e2->SKIP)) = [
```



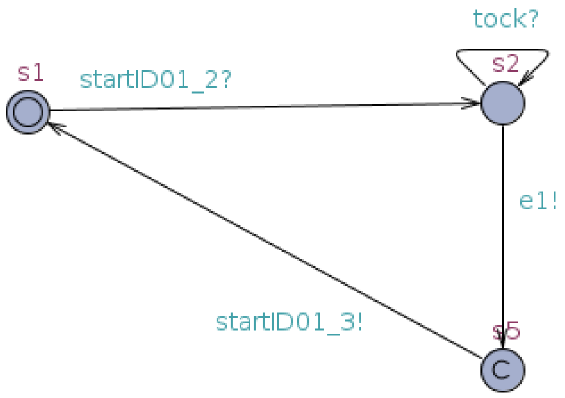
```
       ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



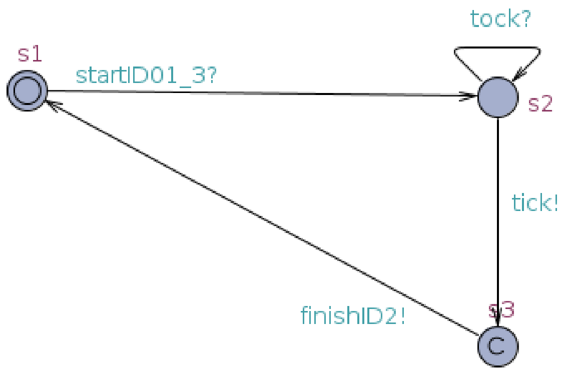
```
 ] ++ transTA(SKIP)
     = [
```

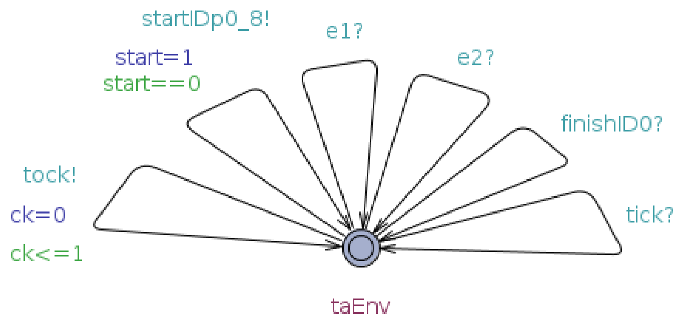


```
              ]

ta2 = transTA(e2->STOP)
    = [
```



```
        ] ++ transTA(STOP)
          = [
```





```
]
```

Example A.13 illustrates using Rule A.12 in translating a process `(e1->SKIP)|||(e2->SKIP))` into a list containing 6 TA. The first TA is a translation of the operator Interleaving. Second and third TA are translations of the LHS process `(e1->SKIP)`. Fourth and fifth TA are translations of the RHS process `!e2->STOP)`. The last TA is an environment TA for the list of the translated TA.

The behaviour of the TA is as follows. The first TA synchronises on the coordinating start event `startID1` and then immediately performs two flow actions `starID2` and `startID3` simultaneously that activate the translation of the LHS and RHS processes respectively. And then the first TA waits on location `s3` until it synchronises on a termination action, either `finishID1` or `finishID2`, and then waits for the second termination action `finishID1` or `finishID2` depending on the first terminating process. The action `finishID1` is termination action of the translated LHS process `(e1->SKIP)`, and `finishID1` is the termination action of the translated RHS process `(e2->SKIP)`. Then, the first TA performs another termination action to record the termination of the whole process.

The second TA is a translation of the event `e1` using Rule A.2.5. While, the third TA is a translation of the subsequent process `SKIP` using Rule A.3. Also, TA3 is a translation of the event `e2` using Rule A.2.5. While TA4 is a translation of the subsequent process `SKIP` using Rule A.3. This completes the description the translated TA in Example A.13 that demonstrates using Rule A.12 in translating the process `(e1->SKIP)|||(e2->SKIP)` into a list of TA.

A.2.13 Translation of Interrupt

This section describes the translation of operator Interrupt. The section begins with presenting a rule for translating the operator `Interrupt` and then follows with an example that illustrates using the rule in translating a process.

Rule A.13. Translation of Interrupt

```
transTA (Interrupt p1 p2 ) procName bid sid fid usedNames =
    ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
        (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
    where
       idTA  = "taIntrpt" ++ bid ++ show sid

       loc1  = Location "id1" "s1" EmptyLabel None
       loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
       loc3  = Location "id3" "s3" EmptyLabel CommittedLoc
       locs  = [loc1, loc2, loc3]

       tran1 = Transition loc1 loc2 [lab1] []
       tran2 = Transition loc2 loc3 [lab2] []
       tran3 = Transition loc3 loc1 [lab3] []
       trans = [tran1, tran2, tran3]

       lab1  = Sync (VariableID (startEvent procName bid sid) [])
                   Ques
       lab2  = Sync (VariableID ("startID" ++ (bid ++ "0") ++
              show (sid+1)) []) Excl
       lab3  = Sync (VariableID ("startID" ++ (bid ++ "1") ++
              show (sid+2)) []) Excl

       (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
                excps) = usedNames

       -- Updates the parameters for interrupts
       intrr'    = intrr ++ (initials p2)
       iniIntrr' = iniIntrr ++ (initials p2)

       usedNames'  = (syncEv, syncPoint, hide, rename, exChs,
                      intrr', iniIntrr, excps)
       usedNames'' = (syncEv, syncPoint, hide, rename, exChs,
                      intrr, iniIntrr', excps)

       (ta1, sync1, syncMap1) =
             transTA p1 [] (bid ++ "0") (sid+1) fid usedNames'
       (ta2, sync2, syncMap2) =
             transTA p2 [] (bid ++ "1") (sid+2) fid usedNames''
```

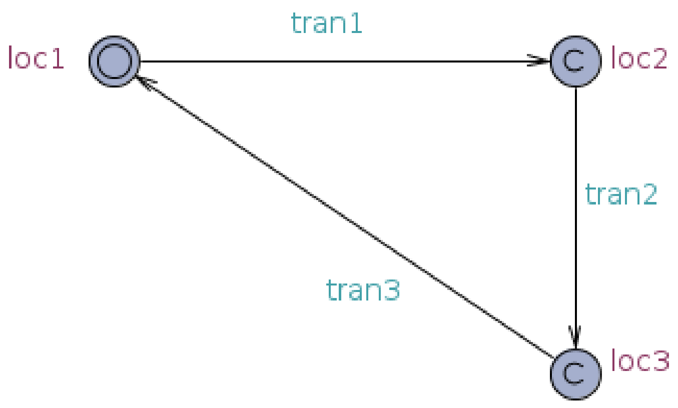


Figure 19: A TA for the structure of the translation of the operator interrupt.

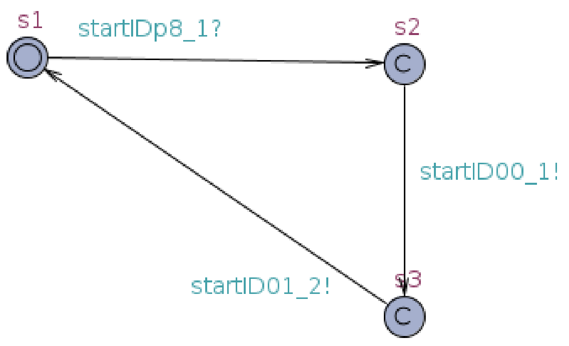
The operator Interrupt is also another binary operator that composes two processes `P1` and `P2` in such a way that the process `P1` begins its behaviour that can be interrupted by process `P2`, whenever process `P2` performs an event. The operator Interrupt is translated into a TA that coordinates the list of TA for the translated processes `P1` and `P2` into `Tp1` and `Tp2`, respectively.

In Figure 19, we annotate the structure of the translated TA with the names used in the translation rule. The translated TA has 3 locations and 3 transitions defined in Lines 7–10 and Lines 12–15. Lines 17–22 define labels for the transitions. Line 24 extracts the list of used names for interrupt. Lines 28–29 updates the used names with the initials of the interrupting process `p2`. Lines 31–34 updates the tuples of the used names `usedNames`. Lines 36–40 define the subsequent translation of the LHS and RHS processes, that is `Tp1` and `Tp2` respectively.

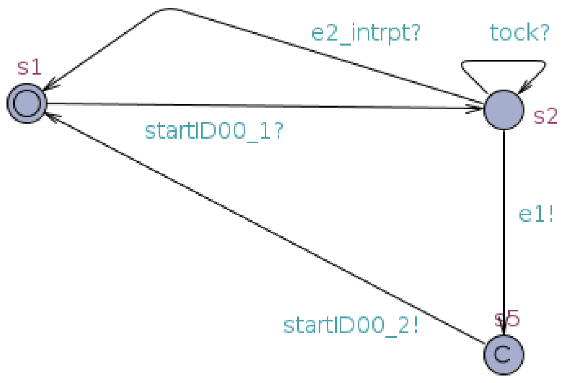
The behaviour of the control TA begins on transition `tran1` for performing a flow action. And then immediately activates the translation of the processes `Tp1` and `Tp2`. For the translation of the interrupted process `Tp1`, each TA in the list `Tp1` has an additional interrupting transition in each stable location, as described in the Translation strategy. The additional transition provides a co-action of the initials of the interrupting process `Tp2`, which enables `Tp2` to interrupt `Tp1` at any stable location. The following example A.14 demonstrates using the rule in translating a process.

Example A.14. An example that illustrates using Rule A.13 in translating the process `((e1-> SKIP)/\(e2-> SKIP))` into a list of TA as follows.

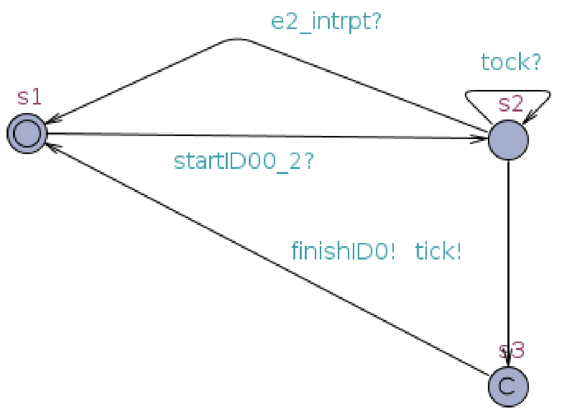
```
transTA((e1-> SKIP)/\(e2-> SKIP)) = [
```



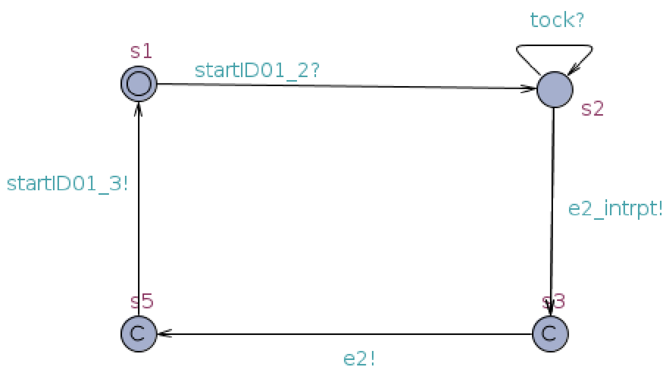
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



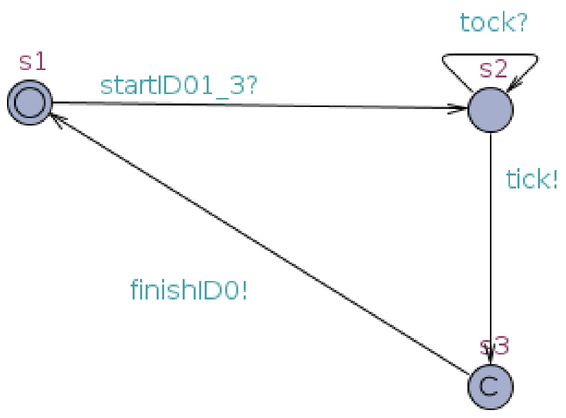
```
  ] ++ transTA(SKIP)
     = [
```



```
            ]

ta2 = transTA(e2->SKIP)
    = [
```



```
        ] = transTA(SKIP)
            = [
```



]

Example A.14 demonstrates a translation of the process ((e1->SKIP)/\(e2->SKIP)) into a list containing 5 TA. The first TA is a translation of the operator interrupt. Second and third TA are translations of the LHS process (e1->SKIP). The second TA is a translation of the event e1 using Rule A.2.5. And the third TA2 is a translation of the subsequent process SKIP using Rule A.4. While the fourth and fifth TA are translation of the RHS process (e2->SKIP). Fourth TA is a translation of the event e2. Fifth TA is a translation of the subsequent process SKIP. The last TA is an environment TA for the list of translated TA.

The behaviour of the translated TA begins with the first TA that synchronises on the flow action startIDp8_1, and then immediately performs two subsequent flow actions startID00_1 and startID01_2 that activate the second and third TA. The second TA synchronises on the action startID00_1, then on location s2, the second TA can be interrupted with co-action of e2_intrpt, or perform the action tock to record the progress of time or proceeds to perform the action e1! and then immediately performs another flow action startID00_2 to activate the third TA.

The third TA synchronises on the flow action startID00_2, and then on location s2, also, the third TA can be interrupted with the co-action e2_intrpt, or perform the action tock to record the progress of time or progress to perform the action tick and then immediately perform the termination action finishID0 to record a successful termination of the LHS process without interrupt.

The fourth TA initiates the behaviour of the RHS process that interrupts the behaviour of the LHS process. The fourth TA begin with synchronising on a flow action stardID01_2 initiated by the first TA. Then, on location s2, either the TA performs the action tock or performs an interrupt action e2_intrpt to interrupt the behaviour of the LHS process, then proceeds to perform the action e2, and then performs another flow action startID01_3 to activate the fifth TA, which performs the action tick, and then performs the termination action finishID0, which records a successful termination of the process. This completes the description of the list of TA that capture the behaviour of the process ((e1->SKIP)/\(e2->SKIP))

A.2.14 Translation of Exception

This section discussed the translation of the operator Exception. The section begins with presenting a rule for translating the operator Exception and then follows with an example that illustrates using the rule in translating a process.

Rule A.14. Translation of Exception

```
transTA (Exception p1 p2 es) procName bid sid fid usedNames =
  ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
    (sync1 ++ sync2), (syncMap1 ++ syncMap2) )
  where
     idTA  = "taException" ++ bid ++ show sid
     loc1  = Location "id1" "s1" EmptyLabel None
     loc2  = Location "id2" "s2" EmptyLabel CommittedLoc
     loc3  = Location "id3" "s3" EmptyLabel None
     loc4  = Location "id4" "s4" EmptyLabel CommittedLoc
     loc6  = Location "id6" "s6" EmptyLabel CommittedLoc
     loc7  = Location "id7" "s7" EmptyLabel None
     loc8  = Location "id8" "s8" EmptyLabel CommittedLoc
     locs  = [loc1, loc2, loc3, loc4, loc6, loc7, loc8]
     tran1 = Transition loc1 loc2 [lab1] []
     tran2 = Transition loc2 loc3 [lab2] []
     tran3 = Transition loc3 loc4 [lab3] []
     tran4 = Transition loc4 loc1 [lab4] []
     tran5 = Transition loc3 loc6 [lab5] []
     tran6 = Transition loc6 loc7 [lab6] []
     tran7 = Transition loc7 loc8 [lab7] []
     tran8 = Transition loc8 loc1 [lab4] []
     trans = [tran1, tran2, tran3, tran4, tran5, tran6, tran7, tran8]
     lab1  = Sync (VariableID (startEvent procName bid sid) []) Ques
     lab2  = Sync (VariableID ("startID" ++ (bid ++ "0") ++
                                        show (sid+1)) []) Excl
     lab3  = Sync (VariableID ("finishID"  ++ show (fid+1)) []) Ques
     lab4  = Sync (VariableID ("finishID"  ++ show (fid))   []) Excl
     lab5  = Sync (VariableID ("startExcp" ++ show (fid+1)) []) Ques
     lab6  = Sync (VariableID ("startID"   ++ (bid ++ "1") ++
                                        show (sid+2)) []) Excl
     lab7  = Sync (VariableID ("finishID"  ++ show (fid+1)) []) Ques

     -- Extracts and updates the list of names used for exception
     (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
               excps) = usedNames
     excps' = (((fst excps) ++ es), snd excps)
     usedNames' = (syncEv, syncPoint, hide, rename, exChs,
                    intrr, iniIntrr, excps')

      -- Subsequent translation of the remaining processes
     (ta1, sync1, syncMap1) =
             transTA p1 [] (bid ++ "0") (sid+1) (fid+1) usedNames'
     (ta2, sync2, syncMap2) =
             transTA p2 [] (bid ++ "1") (sid+2) (fid+1) usedNames'
```

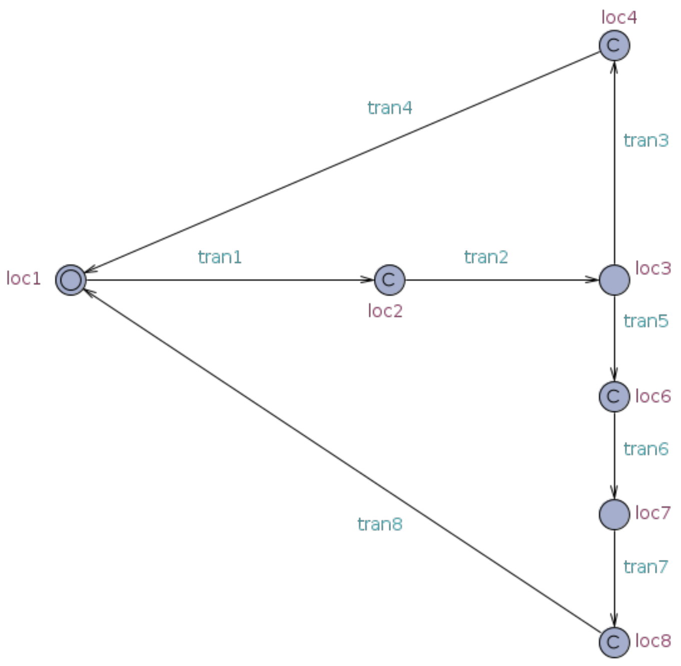


Figure 20: A structure of the control TA for the translation of the operator Exception.

The operator Exception is also another binary operator that combines two processes `p1` and `p2`. Initially the process behaves as `p1` until either `p1` terminates or performs an exception event from the list `es` which terminates the process `p1` and initiates the process `p2`. Like the previous binary operators, the operator `Exception` is translated into a control TA that coordinates the translation of the two processes `p1` and `p2` into `Tp1` and `Tp2`, respectively.

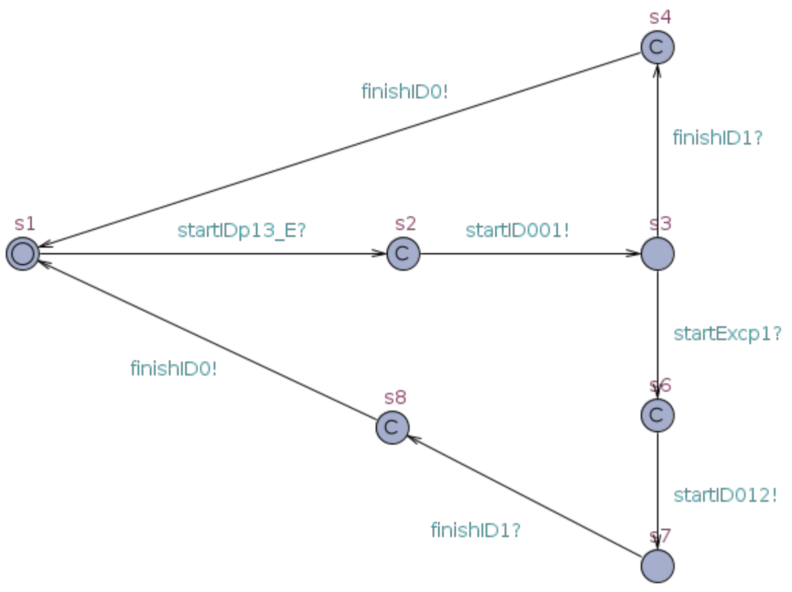
In Figure 20, we annotate the structure of the control TA for the operator exception with the names used in the translation Rule A.14. The translated TA has 8 locations and 8 transitions defined in Lines 6–13 and Lines 14–22 respectively. Then, Lines 23–31 define the labels of the transitions. Lines 33–38 extracts and update the list of names `excps`, which we used for handling exceptions in the tuples `usedNames`. Finally, Lines 41–44 define a recursive call for the subsequent translation of the remaining processes.

The behaviour of the control TA begins on transition `tran1` for performing a flow action. Then, on transition `tran2` the control TA performs another flow action that activates `Tp1`. After that, the control TA remains on location `loc3` until either `Tp1` terminates successfully or performs an exception action from the list `es`. If `Tp1` terminates with performing a termination action the translated TA synchronises with the corresponding co-action on transition `tran3`, and then performs another termination action on transition `tran4` for terminating the whole process.

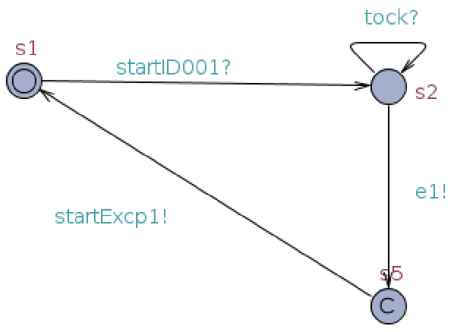
Alternatively, if `Tp1` performs an exception action that raises an exception action, the control TA synchronises with its co-action on transition `tran5`, and then immediately initiates the translation of `Tp2` on transition `tran6`, and then waits on location`loc7` until the translated list of TA `Tp2` performs a termination action and the control TA synchronises with the corresponding co-action on transition `tran7` and then on transition `tran8`, the control TA immediately performs another termination action for terminating the whole process. The following Example A.15 illustrates using the Rule A.14 in translating a process.

Example A.15. An example that illustrates using Rule A.14 in translating a process. This example translates the process `((e1->SKIP)[|{e1}|>(e2->SKIP))` into a list of TA as follows.

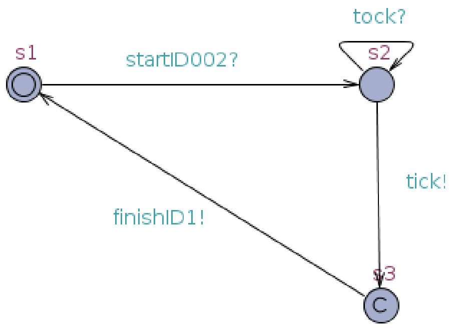
```
transTA((e1->SKIP)[|{e1}|>(e2->SKIP)) = [
```



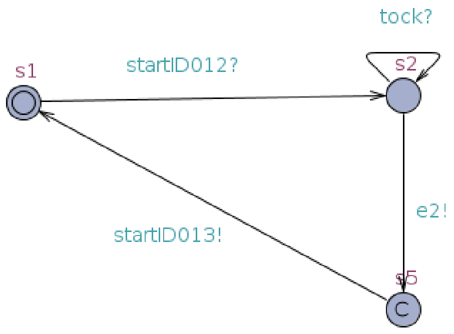
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(e1->SKIP)
    = [
```



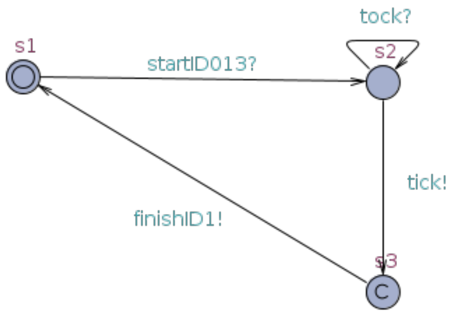
```
] ++ transTA(SKIP)
   = [
```

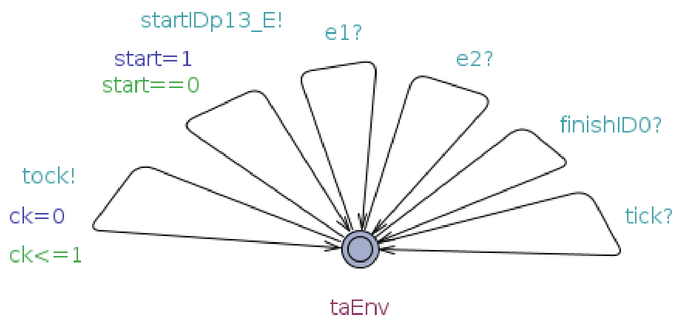


```
         ]

ta2 = transTA(e2->SKIP)
    = [
```



```
      ] = transTA(SKIP)
          = [
```





]

Example A.15 translates the process `(e1->SKIP)[|{e1}|>(e2->SKIP)` into a list containing 5 TA. The first TA is a translation of the operator interrupt using Rule A.14. Second and third TA are translation of the LHS process `(e1->SKIP)`. Second TA captures the translation of the event `e1` using Rule A.2.5. While the third TA captures the translation of the subsequent process `SKIP` using Rule A.3. Similarly, the fourth and fifth TA are translations of the RHS process `(e2->SKIP)`. The fourth TA capture the translation of the event `e2` using Rule A.2.5. The fifth TA captures the translation of the subsequent process `SKIP` using Rule A.3.

The behaviour of the first TA begins with a synchronising on a flow action `startID13_E` that comes from the environment and then immediately performs the subsequent coordination action `startID001` which activates the second TA that initiates the the behaviour of the LHS process `(e1->SKIP)`. After that, the first TA waits on locations3 until it receives either a termination action `finishID1` or an exception action `startExcp1`.

If the first TA receives a termination action `finishID1` which indicates a successful termination of the LHS process `(e1->SKIP)`, then the first TA performs another subsequent termination action `finishID0` to signal a termination of the whole process. Alternatively, if the first TA receives an exception action `startExcp1`, then the first TA immediately performs the flow action `startID012` which activates the RHS process `(e2->SKIP)`. Then, the first TA waits on locations7 until it receives a termination action `finishID1` and then immediately performs the subsequent termination action `finishID0` to signal the termination of the whole process. This completes the behaviour of the first TA for the translation of operator Exception in the process `(e1 ->SKIP)[|{e1}|>(e2->SKIP)`.

A.2.15 Translation of Timeout

This section describes the translation of operator Timeout. The section begins with presenting a rule for translating the operator timeout and then follows with an example that illustrates using the rule in translating a process.

According Roscoe [33], the operator timeout specifies a deadline for the LHS process to perform an event before the deadline or the process the RHS process begins its behaviour and the whole process behaves as the RHS process. In `tock-CSP`, this is express in term of internal choice and delay process as follows:

```
(P1 [d> P2 = P1)|~| (WAIT(2);P2)
```

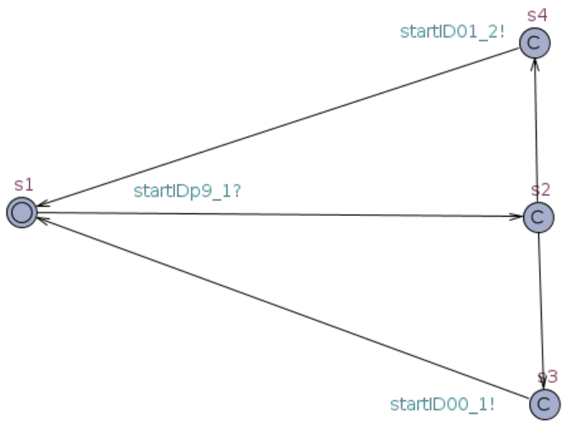
We follow a similar pattern in translating the operator timeout. We translate the operator in term of the two previous rules for for translating internal choice (`|-|`) and a process dalay (`WAIT(d)`. Rule A.15 expresses the translation rule and Example A.16 demonstrates using the rule in translating a tock-CSP process.

Rule A.15. Translation of Timeout

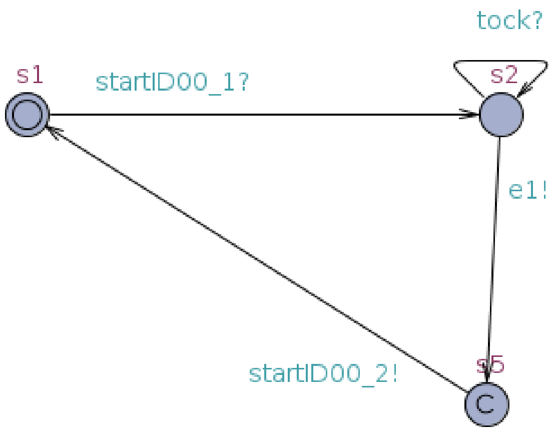
```
transTA (Timeout p1 p2 d) procName bid sid fid usedNames =
        transTA (IntChoice p1 (Seq (WAIT d) p2 )) procName bid sid
                                                  fid usedNames
```

Example A.16. An example that illustrates a translation of process using Rule A.14. This example translates the process `((e1->SKIP)[2>(e2->SKIP))` into a list of TA as follows.

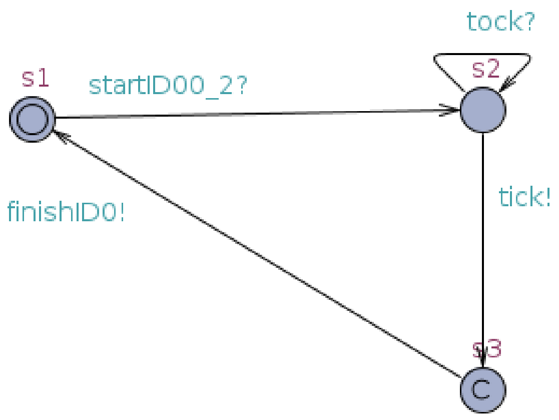
```
  transTA ((e1->SKIP)[2>(e2->SKIP))=
     transTA ((e1->SKIP)|~|(WAIT(2);(e2->SKIP)))= [
```



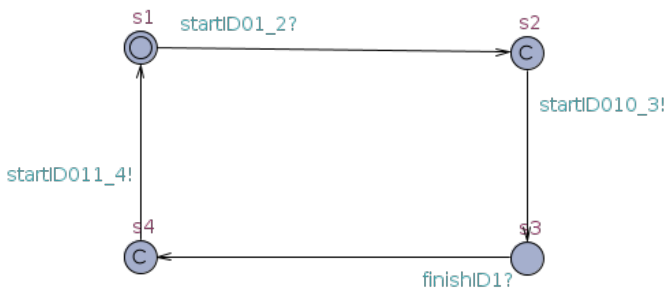
```
        ] ++ ta1 ++ ta2

where
ta1 = transTA(c1->SKIP)
    = [
```



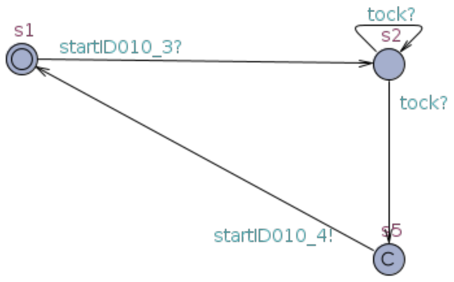
```
 ] ++ transTA(SKIP)
      = [
```



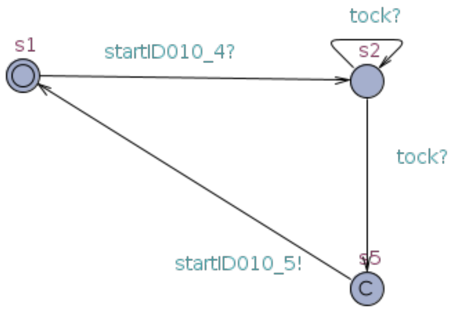
```
              ]

ta2 = transTA ((WAIT 2);(e2->SKIP))
    = [
```



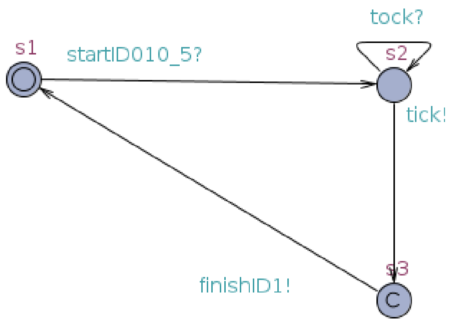
```
         ] = ta21 ++ ta22
ta21 = transTA(WAIT 2)
       = [
```



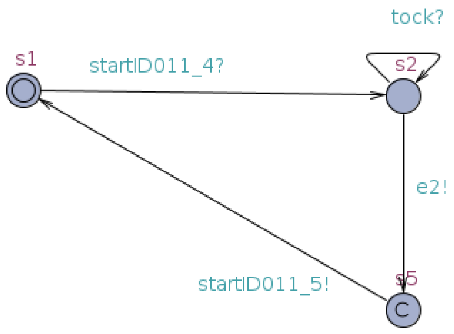
```
    ] ++ transTA(WAIT 1)
       = [
```



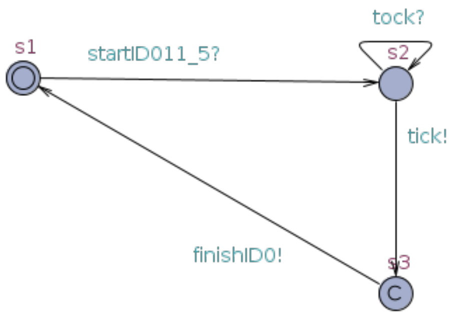
```
    ] ++ transTA(SKIP)
       = [
```

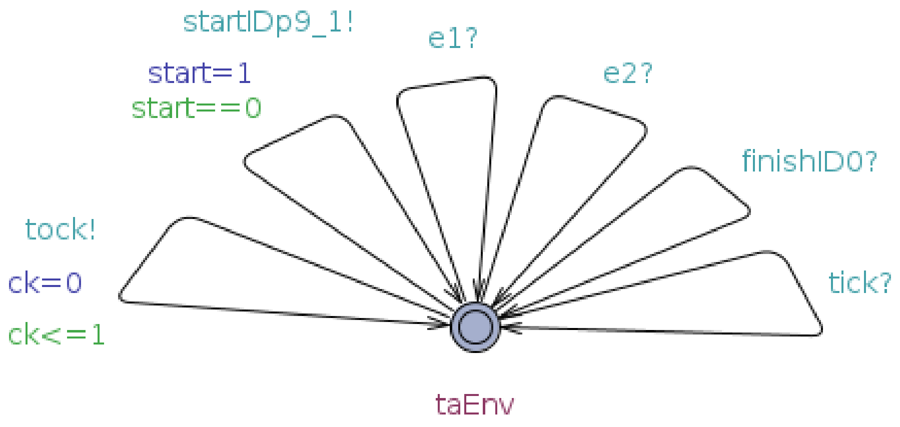


```
                ]

ta22 = transTA(e2->SKIP)
       = [
```



```
] ++ transTA(SKIP)
    = [
```





```
]
```

Example A.16 illustrates using Rule A.15 in translating the process `((e1->SKIP) [2>(e2->SKIP))` into a list of TA containing 10 TA. The first TA is translation of the operator for internal choice. The second and third is translation of the LHS process `(e1->SKIP)`. The second TA captures the translation of the event `e1` according to Rule A.2.5. The third TA captures the translation of the subsequent process `SKIP`. From the fourth to the ninth TA are translation of the RHS process `(WAIT(2);(e2->SKIP))`. The fourth TA is a translation of the operator sequential composition according to Rule A.10. The fifth and sixth TA are translation of the delay process `WAIT(2)` according to Rule A.6. The seventh TA captures the translation of the event `e2` according to Rule A.2.5. The eighth TA captures the translation of the subsequent process `SKIP`. The last TA is an environment TA for the list of translated TA of the process `((e1->SKIP)[2>( e2->SKIP))`. This completes the description of an example that illustrates using Rule A.15 in translating a process.

A.2.16 Translation of EDeadline (Event Deadline)

This section describes the translation of the construct `Edeadline` for a process that assigns a deadline to an event. The section begins with presenting a translation rule for the construct `Edeadline` and then follows with an example that illustrates using this rule in translating a process.

Rule A.16. Translation of EDeadline (Event Deadline)

```
1  transTA (EDeadline e n) procName bid sid fid usedNames =
2    ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
3    where
4      idTA = "taDeadln" ++ bid ++ show sid
5
6      loc1 = Location "id1" "s1" EmptyLabel None
7      loc2 = Location "id2" "s2" EmptyLabel None
8      loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
9      locs = [loc1, loc2, loc3]
10
11     tran1 = Transition loc1 loc2 ([lab1] ++ t_reset)    []
12     tran2 = Transition loc2 loc2 ([lab2] ++ dlguard ++ dlupdate) []
13     tran3 = Transition loc2 loc3 ([lab3] ++ dlguard2)   []
14     tran4 = Transition loc3 loc1 [lab4] []
15     trans = [tran1, tran2, tran3, tran4] ++
16             (transIntrpt intrpts loc1 loc2)
17
18     lab1 = Sync (VariableID (startEvent procName bid sid) []) Ques
19     lab2 = Sync (VariableID "tock"    []) Ques
20     lab3 = Sync (VariableID (show e)           []) Excl
21     lab4 = Sync (VariableID ("finishID" ++ show fid)   []) Excl
22
23     -- reset timer
24     t_reset = [(Update [(AssgExp (ExpID "tdeadline")
25                                  ASSIGNMENT (Val 0))] )]
26
27     dlupdate = [(Update [(AssgExp (ExpID "tdeadline")
28                                  AddAssg (Val 1) ) ] ) ]
29
30     dlguard  = [(Guard (BinaryExp (ExpID "tdeadline") Lth (Val n)))]
31     dlguard2 = [(Guard (BinaryExp (ExpID "tdeadline") Lte (Val n)))]
32
33     (_, _, _, _, _, intrpts, _, _) = usedNames
34
```

Rule A.16 defines a translation of the construct `Edeadline` into a TA. In Figure 21 we annotates the structure of the TA with the names used in the translation rule. The TA has 3 locations and 4 transitions defined in lines 6–9 and lines 11–16 respectively. Lines 18 – 32 define the labels of the transitions. Line 24–25 define a label for resetting the timer. Lines 27–28 update the time with one time unit after every action `tock`. Line 30–31 define guards for controlling the deadline. The following example illustrates

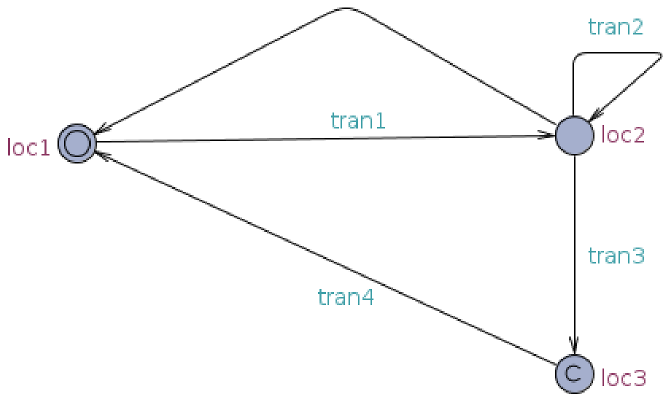
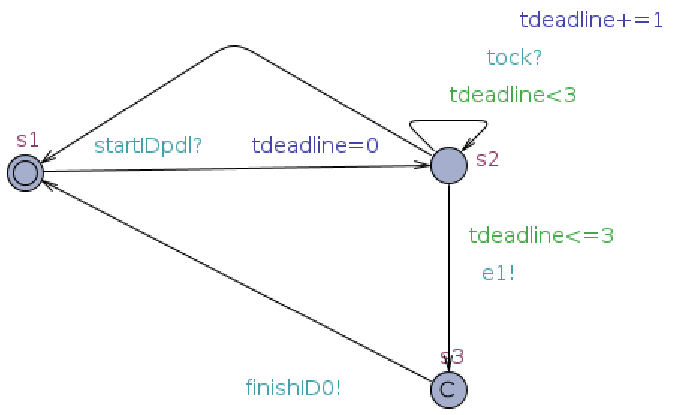


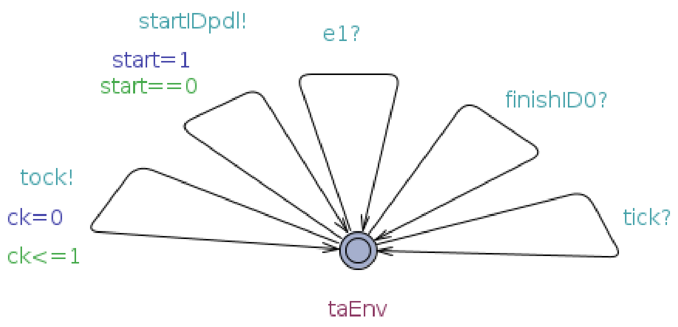
Figure 21: A structure of the control TA for the translation of the process Edeadline.

using this Rule A.16 in translating a process.

Example A.17. This example illustrates using Rule A.16 in translating the process (`EDeadline (e1, 3))` into a list of TA as follows.

```
transTA (EDeadline (e1, 3)) "pdl" "0" 0 0
                   ([], [], [], [], [], [], [], ([],[])) ⇝ [
```





The behaviour of the first TA begins with synchronising on a flow action `startIDpd ?` from the environment and then reset deadline `tdeadline` to zero. After that, on location `s2` either the TA performs the action `tock` to record the progress of time or the TA performs the event `e1` within a deadline of 3 time units After the deadline `tdeadline` the guard `tdeadline<=3` blocks the event `e1` and the TA follows a silent transition to the initial location `s0`. The second TA is an environment TA for the list of translated TA. This completes the behaviour of the TA for the translation of the process `Edeadline(e1, 3)`.

A.2.17 Translation of Hiding

This section describes the translation operator Hiding. The section begins with presenting a rule for translating the operator `Hiding`, and then follows with an example that illustrates using the rule in translating a process.

Rule A.17. Translation of Hiding

```
transTA (Hiding p es ) procName bid sid fid usedNames =
        transTA p      procName bid sid fid usedNames'
        where
           (syncs, syncPoints, hides, renames, exChs, intrrs,
                   iniIntrrs, excps) = usedNames

           -- Updates the parameter for hiding
           usedNames' = (syncs, syncPoints, (es ++ hides), renames,
                          exChs, intrrs, iniIntrrs, excps)
```

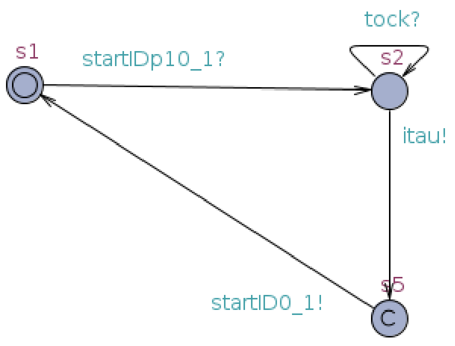
Rule A.17 updates the used name for hiding `hides`, which is used in developing a rule for translating prefix. Line 1 defines the function `transTA` for the construct `Hiding`. Line 2 describes the output in terms of the function `transTA` with an updated name `usedNames'`, which contains an updated name `hides`. Lines 4–5 extract the name `hides` from the tuples of used names `usedNames`. Lines 8–9 updates the name `usedNames` with hiding events for subsequent translation.

Rule A.2.5 checks the used name `hides` in translating each event. If an event is in the list of hiding events, Rule A.2.5 translates the event into a special name `itau`. While, if an event is not part of the used name `hides`, Rule A.2.5 translates the event with its name in the output TA. The following Example A.18 illustrates using this rule in translating a process.

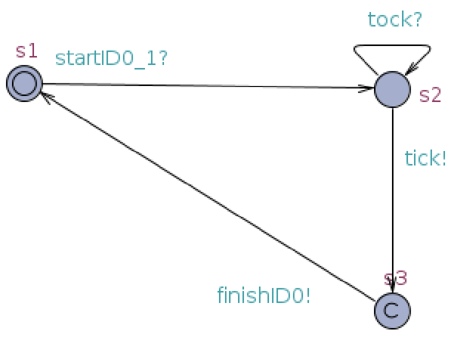
Example A.18. This example demonstrates using Rule A.17 in translating the process `((e1->SKIP)\{e1})` into a list of TA as follows.

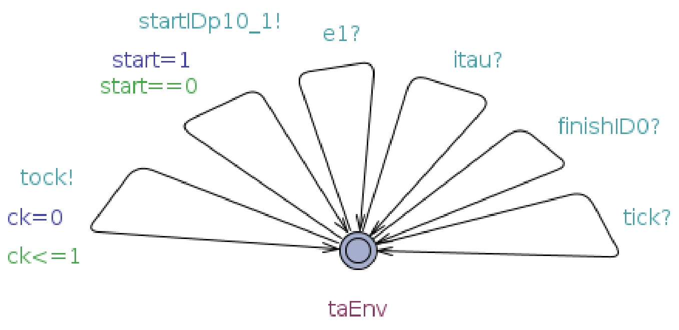
```
transTA((e1->SKIP)\{e1}   "p10_1" _ 0 0 usedNames ) =
      transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
```

```
    where
    (syncs, syncMap, hides, rename, chs, intrpt, initIntrpt)
        = usedNames

    usedNames' = (syncs, syncMap, [e1]++hides, rename,
                        chs, intrpt, initIntrpt)

    transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
        = [
```



```
   ] ++ transTA(SKIP)
          = [
```





]

Example A.18 illustrates a translation of a process using A.17. The example translates the process `((e1->SKIP)\{e1})` into a list of TA that contains 3 TA. The first TA is a translation of the hiding event `e1` into a special name `itau`. The second TA is a translation of the subsequent process `SKIP` according to Rule A.3. The third TA is an environment TA for the list of translated TA.s

The behaviour of the translated TA begins with the first TA, which synchronises on a flow action `startIDp10_1` from the environment TA, then performs the hiding action `itau`, and then immediately performs another flow action `startID0_1!` to activate the second TA. The second TA synchronises on the flow action `startID0_1?`, then performs the flow action `tick`, and then immediately performs the termination action `finishID0` that records a successful termination of the whole process. This completes the translation of the process `((e1->SKIP)\{e1})`.

A.2.18 Translation of Renaming

This section describes the translation of operator Renaming. The section begins with presenting a rule for translating the operator renaming and then follows with an example that illustrates using the rule in translating a process.

Rule A.18. Translation of Renaming

```
transTA (Rename p pes) procName bid sid fid usedNames
 =  transTA    p       procName bid sid fid usedNames'
        where
            (syncs, syncPoints, hides, renames, exChs, intrrs,
                    iniIntrrs, excps) = usedNames

           -- Updates the name renames in the list of usedNames
           usedNames' = (syncs, syncPoints, hides, (renames ++ pes),
                              exChs, intrrs, iniIntrrs, excps)
```

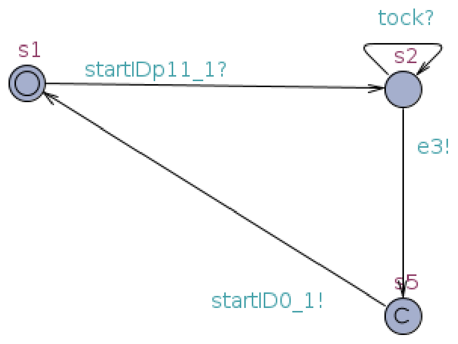
Translation of operator renaming follows similar patterns with the previous Rule A.17, except that, the parameter for renaming is a list of pairs, an event with its corresponding new name. This rule updates the used name `renames` from the tuples `usedNames`. Then in translating prefix, we check the updated used names `renames` for translating each event. If an event is in the list `renames`, Rule A.2.5 replaces the event name with its corresponding new name, such that it appears with its new name in the translated TA.

Example A.19. An example for translating a process that demonstra a translation of the operator Renaming in translating the process `((e1->SKIP)[[e1<-e3]])`.

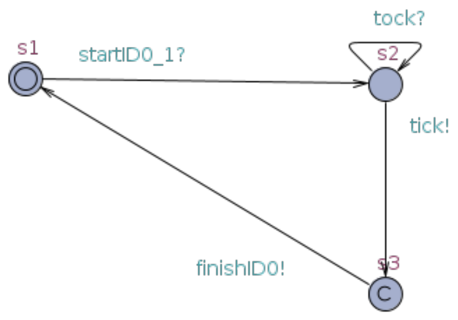
```
transTA(((e1-> SKIP)[[e1<-e3]]) "p11_1" "" 0 0 usedNames)
```

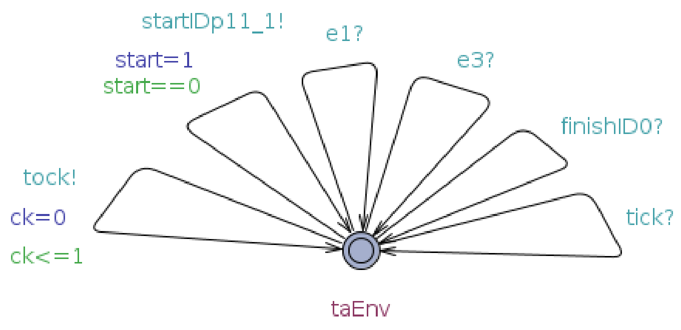
```
    = transTA((e1->SKIP) "p11_1" "" 0 0 usedNames')
where
    (syncs, syncMap, hides, rename, chs, intrpt, initIntrpt)
            = usedNames

    usedNames' = (syncs, syncMap, hides, rename ++ [e1, e3],
                        chs, intrpt, initIntrpt)
```

```
transTA((e1->SKIP) "p11_1" "" 0 0 usedNames')
    = [
```



```
   ] ++ = transTA(SKIP)
          = [
```





]
Example A.19 illustrates using Rule A.18 in translating the process `((e1->SKIP)[[ e1<-e3]])` into a list of TA that contains 3 TA. The first TA is a translation of the event `e2`, which appears with its new name `e3` in the translated TA. The second TA is a translation of the subsequent process `SKIP`. The last TA is an environment TA for the list of translated TA. This completes the description of translating the process `((e1->SKIP)[[e1<-e3]])` into a list of TA.

A.2.19 Definition of Environment TA

This section describes the defines an explicit environment TA for the translated TA of the UPPAAL models. Definition A.2.19 provides a Haskell function that express the structure of an explicit environment TA for the translated UPPAAL system. We have seen various examples of environment TA in the provided examples accompanying each translation rule.

Function for an Environment TA

```
1  env :: String -> [Event] -> Template
2  env    pid       es      =
3    Template "Env" [] [] [loc] [] (Init loc) trans
4    where
5     loc = Location  "taEnv" "taEnv" EmptyLabel None
6     tll = Transition loc     loc   -- a common name used for defining
7                                    -- the transitions, from loc to loc
8     trans =
9       [(tll [(Sync    (VariableID id []) Ques)] [])|(ID id) <- es] ++
10      [ tll [(Sync    (VariableID "startID0_0"  []) Excl),
11             (Guard   (BinaryExp (ExpID "start"   ) Equal (Val 0))),
12             (Update [(AssgExp   (ExpID "start"   ) ASSIGNMENT
13                                                      (Val 1))])] [],
14      tll [(Sync    (VariableID ("startID" ++ pid) []) Excl),
15           (Guard   (BinaryExp  (ExpID "start"   ) Equal (Val 0))),
16           (Update [(AssgExp    (ExpID "start"   ) ASSIGNMENT (Val 1))
17                   ]
18           )
19          ] [],
20      tll [(Sync  (VariableID "finishID0" []) Ques)] [],
21      tll [(Sync  (VariableID "tick"      []) Ques)] [],
22      tll [(Sync  (VariableID "itau"      []) Ques)] [],
23      tll [(Sync  (VariableID "tock"      []) Excl),
24           (Guard (BinaryExp (ExpID "ck") Lte (Val 1))),
25           (Update [AssgExp  (ExpID "ck") ASSIGNMENT (Val 0)])
26          ] []
27     ]
```

As highlighted in the translation strategy, each process is translated into a list of TA that includes an environment TA. The function defines the environment TA that has one location as defined in Line 5. While the remaining Lines 6–27 defines the transitions of the TA. Line 6 defines a common name used in defining the source and targets of all the transitions. Line 9 defines a transition for each action from the translated process. Lines 10–12 defines the first starting flow action `startID0_0`, which initiates the behaviour of the translated TA (for the case of anonymous function). This starting transition has guards that blocks the environment from starting the behaviour multiple times. In the case of a named process, Lines 14–16 defines another transition for starting flow action that has the process name (for the case of a named process). Line 20 defines the final termination co-action for terminating the whole process. Line 21 defines a transition for a co-action of the translated action `tick`. Line 22 defines a transition for a co-action of hiding events `itau`. Lines 23–26 define a transition for the translated action `tock`, which is associated with clocks variable for recording the progress of time. This completes the definition of the environment TA. And also completes the descriptions of the translation rules developed for translating `tock-CSP` into a list of TA.

Appendix B Specification of the Case Studies

B.1 Cash Machine (ATM)

A machine that goes through cycles of accepting a card, requiring its PIN, and servicing one request before returning the card to the customer. The request can be cash withdrawal, transferring cash and checking a balance of the account. While if the PIN is not correct, the machine simply returns the card, and continue an operation that prepares the machine for accepting another card [34].

In tock-CSP, the operation of the machine is as follows:

```
channel tock, card, acceptPIN, correctPIN, incorrectPIN,
   returnCard, withdrawCash, amount, dispenseCash,
   returnCArd, checkBalance, displayBalance, transfer,
   acceptAccountNo, acceptAmount

OneStep(_) = 0

Timed(OneStep){

ATM = card -> acceptPIN -> (correctPIN ->  Options)
                        |~| (incorretPIN -> returnCard ->
                           ATM)
Options =
    (withdrawCash -> amount -> dispenseCash -> returnCard ->
        ATM)
    [] (checkBalance -> displayBalance -> returnCard  -> ATM
       )
    [] (transfer -> acceptAccountNo -> acceptAmount ->
        returnCard -> ATM)

assert ATM :[deadlock-free]
}
```

Specification:

1. E ◇ returnCard
   Can the machine eventually return the card, if there is a card in the machine?

2. `card --> returnCard`
   Whenever the machine accepts a card, eventually the machine will return the card.

3. `withdrawCash --> returnCard`
   After withdrawing cash, eventually the machine will return the card.

4. `checkBalance --> returnCard`
   After checking account balance, eventually the machine will return the card.

5. `transfer --> returnCard`
   Whenever the machine accepts a card, eventually the machine will return the card.

B.2 Automated Barrier

An automated barrier that accepts tickets and raises the barrier exactly two time units after dispensing the ticket. It lowers the barrier one second after receiving a through signal. If no such signal has been received after 20 time units of raising the barrier, it emits a beep once per time unit until through or reset is received. The barrier is lowered one second after the occurrence of either of these events [35].

In tock-CSP, the operation of the automated barrier is as follows:

```

channel tock, acceptTicket, dispenseTicket, raiseBarrier,
   beep,
      through, reset, receivedSignal, lowerBarrier

OneStep(_) = 0

Timed(OneStep) {

AutoBarrier = acceptTicket -> dispenseTicket -> tock -> tock
    ->
            raiseBarrier -> Timeout(Response, NoResponse,
               20)

Timeout(P, Q, d) = P [] (WAIT(d);Q)

Response = receivedSignal -> tock -> lowerBarrier ->
   AutoBarrier

NoResponse = Beeping /\ Action

Beeping = beep -> tock -> Beeping

Action = ((through -> SKIP) [] (reset -> SKIP))
            ;(lowerBarrier -> AutoBarrier)

-- Alternatively:
-- Action = ((through -> lowerBarrier -> SKIP)
            [] (reset -> lowerBarrier -> SKIP))

assert AutoBarrier :[deadlock-free]

}
```

Specifications:

1. $(E \diamond \; raiseBarrier)$
   Can the system eventually raise the barrier
2. $(E \diamond \; lowerBarrier)$
   Can the system eventually lower the barrier
3. `raiseBarrier --> lowerBarrier`
   Whenever the system raises the barrier, eventually the system will lower the barrier.
4. `receivedSignal --> lowerBarrier`
   After receiving a signal, eventually the system will lower the barrier.
5. `acceptTicket --> dispenseTicket`
   After the system accept ticket, eventually the system will dispense the ticket.
6. `beep --> lowerBarrier`
   Can beep leads to lowering the barrier.
7. `beep` $\mathcal{U}$ `(reset or through)`
   Can the system continues beeping until the system receive an action either reset or through.

B.3 Bookshop Payment System

A book shop operates a system whereby customers pay for books at a cashier's counter and collect them at a different counter where they had previously been lodged. A customer may lodge a chosen book with the counter and have a chit issued in return [35].

In tock CSP, the operation of the book counter is described as follows:

```
BOOK = lodge -> issueChit -> paymentSystem
        [] receiveReceipt -> claim -> paymentSystem
```

A customer may lodge a chosen book with the counter and have a chit issued in return. In order to claim the book to take away, a receipt must be provided. This may be obtained from the cashier, described as follows:

```
CASHIER = receiveChit -> payment -> issueReceipt ->
    paymentSystem
```

Book chits must be issued by the book counter before they can be received by the cashier:

```
CHIT = (issueChit -> paymentSystem)
        |~| (chit -> receiveChit -> paymentSystem)
```

Similarly, receipts must be issued before they can be exchanged for books.

```
RECEIPT =  (issueReceipt -> paymentSystem)
            |~| (receiveReceipt -> paymentSystem)
```

Each of these components imposes some constraint on the customer; and they are all in place together. The complete payment system which the customer must navigate is captured as the combination of the components of the system as follows.

```
paymentSystem = CASHIER [] BOOK [] CHIT [] RECEIPT
```

The complete tock-CSP specification is as follows:

```

channel tock, lodge, issueChit, receiveReceipt, claim,
    receiveChit,
        payment, issueReceipt

OneStep(_) = 0

Timed(OneStep){

paymentSystem = CASHIER
                [] BOOK
                [] CHIT
                [] RECEIPT

BOOK = lodge -> issueChit -> paymentSystem
        [] receiveReceipt -> claim -> paymentSystem

CASHIER = receiveChit -> payment -> issueReceipt ->
    paymentSystem


CHIT = (issueChit -> paymentSystem)
       [] (chit -> receiveChit -> paymentSystem)


RECEIPT =  (issueReceipt -> paymentSystem)
           [] (receiveReceipt -> paymentSystem)

assert paymentSystem :[deadlock-free]

}
```

Note that the initial specification in S10, uses interleaving operator, highlighted below.

```
    paymentSystem = CASHIER ||| BOOK ||| CHIT |||RECEIPT
```

However, FDR does not support recursion thru interleaving. So I replaced the interleaving operator with the choice operator in the specification.

Specification:

1. `receiveChit --> issueReceipt`
   Whenever the system accept a chit, eventually the machine will issue a receipt.

2. ($E\diamond$ `chit`)
   Can the machine eventually issue a chit

3. ($E\diamond$ `receipt`)
   Can the machine eventually issue a receipt

B.4 Railway Crossing System

The system consists of three components: a train, a gate, and a gate controller. The gate should be up to allow traffic to pass when no train is approaching but should be lowered to obstruct traffic when a train is close to reaching the crossing. It is the task of the controller to monitor the approach of a train and to instruct the gate to be lowered within the appropriate time. The train is modelled at a high level of abstraction: the only relevant aspects of the train's behaviour are when it is nearing the crossing when it is entering it, when it is leaving it; and the minimum delays between these events [34, 35].

The gate controller receives two types of signal from the crossing sensors: near ind, which informs the controller that the train is approaching, and out ind, which indicates that the train has left the crossing. It sends two types of signal to the crossing gate mechanism: down command, and up command, which instruct the gate to go down and up respectively. It also receives a confirmation confirm from the gate. These five events form the alphabet C of the controller. The gate, modelled by GATE, responds to the commands sent by the controller. The additional events up and down are included to model the position of the gate. These two events, together with up command and down command and the confirmation confirm, form the alphabet G of the gate [34, 35].

```
channel tock, nearInd, outInd, confirm, upCommand,
   downCommand,
        down, up, trainNear, enterCrossing, leaveCrossing,
           pass

OneStep(_) = 0

Timed(OneStep){

-- The crossing system, in conjunction with the train, is
   described as follows.
System = Train [|{nearInd, outInd}|]  Crossing

Crossing = Controller [|{downCommand, upCommand}|] Gate

Controller = nearInd -> downCommand -> confirm -> Controller
             [] outInd -> upCommand -> confirm -> Controller

-- The gate process responds to the controllers signals
-- by raising and lowering the gate
Gate =  downCommand  -> down -> confirm -> Gate
        [] upCommand -> up   -> confirm -> Gate

-- The process TRAIN will be used to model the approach of
   the train,
-- and its effect upon the crossing system

Train = trainNear -> nearInd -> enterCrossing ->
        leaveCrossing -> outInd -> pass -> Train

assert System :[deadlock-free]

}
```

Specification:

1. $E\diamond$ pass
   The train eventually pass the gate.

2. `trainNear --> down`
   When the train is near eventually the gate will go down

3. `leaveCrossing --> up`
   Leaving the crossing eventually leads to opening the gate.

B.5 Thermostat

A thermostat that monitors ambient temperature and controls a valve to enable or disable a heating system. It is triggered by the input of the events too.hot and too.cold, and is required to send a signal turn.off or turn.on in response to these respective inputs, after two time units [35].

```
channel tock, tooHot, tooCold, turnOff, turnOn

OneStep(_) = 0

Timed(OneStep){

Thermostat = (tooHot -> tock -> tock -> turnOff ->
    Thermostat)
              [] (tooCold -> tock -> tock -> turnOn ->
                  Thermostat)

assert Thermostat :[deadlock-free]

}
```

Specification:

1. `tooHot --> turnOff`
   Whenever the temperature is too hot, eventually the system will turn off,

2. `tooCold --> turnOn`
   Whenever the temperature is too cold eventually the system will turn on

3. `turnOn` $\mathcal{U}$ `tooHot`
   Can the system continues to remain on until it receives a signal for too hot.

4. `turnOff` $\mathcal{U}$ `tooCold`
   Can the system continues to remain off until it receives a signal for too cold.

B.6 A Simple Mobile System

The provided model is a simple model of a mobile system that moves at a specified linear velocity lv and observes the presence of an obstacle. When the system detects an obstacle, the system turns at a specified angular velocity av and then continue moving, repeating the procedure again [8].

```
channel obstacle, tock, moveRet, moveCall, turnCall, turnReturn

OneStep(_) = 0

Timed(OneStep) {

mSystem = EntryMoving /\ (obstacle-> SKIP);EntryTurning;
                                 WAIT(3);mSystem

EntryMoving  = EDeadline(moveCall, 0);EDeadline(moveRet, 0);
                        WAIT(1);EntryMoving

EntryTurning = EDeadline(turnCall, 0); EDeadline(turnReturn, 0)

EDeadline(e, t) = (e -> SKIP) [|{e}|] (WAIT(t) /\ e->SKIP)

assert mSystem :[deadlock-free]

}
```

Specification:

1. `E` $\diamond$ `obstacle`
   Can the system eventually detects an obstacle, if an obstacle exists.

2. `moveCall --> obstacle`
   Whenever the system moves, it will eventually detect an obstacle, if an obstacle exists.

3. `turnCall --> moveCall`
   Whenever the system moves, it will eventually detect an obstacle, if an obstacle exists.

4. `moveCall` $\mathcal{U}$ `obstacle`
   Can the system continues to move until it detects an obstacle.