A Report for the Translation of tock-CSP into Timed Automata for UPPAAL

V010220

1st February 2020

Abstract A process algebra `tock-CSP` provides textual notations for modelling discrete-time behaviours. Automatic verification of tock-CSP is supported by FDR. Similarly, automatic verification of Timed Automata (TA) is supported by real-time verification toolbox UPPAAL. However, `tock-CSP` differs from TA in terms of both modelling and verification as well as the kind of requirements that can be addressed with each of them. In this work, we translate `tock-CSP` into TA so that we can take advantage of TA and its supporting tool UPPAAL. We developed a translation technique that enables us to work out a list of translation rules for translating the constructs of `tock-CSP` into a network of small TA, which address the complexity of capturing the compositionality of `tock-CSP` that is not available in TA. Correctness of the translation is justified by testing based on finite approximations to trace sets. The results show significant performance increase in using UPPAAL for the verification of deadlock freedom.

1. Introduction

Communicating Sequential Processes (CSP) is an established process algebra that provides notations for both modelling and verification of concurrent systems [22, 23]. Using CSP for verification has been supported with an automatic model-checker FDR [9]. CSP and its modelling approaches have significantly contribute in understanding the complexity of concurrent systems [23].

Afterwards, `tock-CSP` provides an additional event *tock* to record a progress of time. This facilitates using CSP for modelling temporal specifications in a suitable format for formal analysis and verification. tock-CSP retains the structure of CSP and utilises the existing works and tools build for CSP. Thus, automatic verification is supported by FDR [9, 22]. As a result of that, `tock-CSP` has been used in the verification of case

studies such as security protocols [8] and Railways system [14]. Recently tock-CSP has been used for capturing the semantics of RoboChart, a DSML for modelling robotics system. Additionally, a recent extension of tock-CSP provides additional notations for specifying budgets and deadlines for effective modelling and verification of temporal specifications [2].

Based on this foundation, we add a step forward by developing a technique for translating tock-CSP into TA that enable using the facilities of UPPAAL for verification of tock-CSP models. UPPAAL is a tool-suite for modelling and verification of real-time systems [3]. So, in this work we develop both a translation technique and a supporting tool for automatically translating tock-CSP into a network of TA that capture the behaviour of the input tock-CSP models.

The results of this translation work show a significant increased performance for using UPPAAL in the verification of deadlock freedom in the translated tock-CSP. In comparing the performance, first, we formulate various tock-CSP processes for the possible pairs of the tock-CSP operators that we consider for the scope of this work. Second, we use the developed translation tool to translate the formulated processes into TA for UPPAAL. Third, we used both FDR and UPPAAL to compare the performance of checking deadlock freedom in both tock-CSP models and translated TA, respectively.

The structure of the paper is as follows. Next, Section 2 provides brief backgrounds that are essential for understanding the translation technique. Section **??** describes the translation technique and provides sample of the translation rules. Section 4 discusses an evaluation of the translation technique. Section **??** highlights related work and also provides brief comparison with this work. Finally, Section 6 highlights future work and concludes the paper.

2. Background

This section discusses the required background for understanding the translation work. We begin with discussing the notations of tock-CSP and then discuss the target outcome of the translation work, that is a list of suitable TA for UPPAAL.

2.0.1. tock-CSP

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

Additionally, there are various binary operators such as sequential composition (;), which composes two processes one after the other. So a process P1;P2 is a process that behaves as P1, and then after P1 terminates, the process behaves as P2. For example,

the process `(move->SKIP);(turn->SKIP)`, specifies a system that moves, turns and then terminates.

There are other binary operators for composing processes in different shapes, such as concurrency, choice and interrupt. The collection of the operators provides a rich set of constructs for modelling untimed systems. Details of the CSP operators is available in these books [22, 23].

tock-CSP enriches the notations with an additional event `tock` for recording the progress of time [22]. Each event `tock` indicates a passage of a single time unit. For example, `move->tock->tock->turn->SKIP` specifies a behaviour of a system that moves, and then after at least two time units, the system turns and then terminates. This additional event tock enables specifying temporal specification in a format that is suitable for formal analysis [9].

For this work, we provide a BNF that defines the `tock-CSP` operators we consider. Part of the BNF is presented here in Definition A encoded in the syntax of Haskell data definition. The complete BNF is available in Appendix A. The BNF provides a basic foundation for the translation work.

Data definition of `CSPproc`

```
1 data CSPproc = STOP
2              | SKIP
3              | Prefix     Event     CSPproc
4              | GenPar     CSPproc   CSPproc [Event]
```

2.0.2. Timed Automata for UPPAAL

TA provides graphical notations for modelling Hybrid systems. In UPPAAL, systems are modelled as a network of TA. Mathematically, TA is defined as a tuple (L, l_0, C, A, E, I), where L defines a set of locations such that l_0 is the initial location, C is the set of clocks, A is the set of actions, E is the set of possible edges that connects the locations and `I` is an invariant that is associated with a location. 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 that is triggered by an action A associated with a set of clocks B(C) that are reset on following the edge E to a location L [3, 4].

In UPPAAL, a TA performs an action over an edge, synchronously with the environment and possibly with some other TA in the case of broadcast action. An action triggered a transition from one location to another or possibly returns to the same location. A state of a system comprises of three elements: a list of the locations for all the network of TA, a list of values for both associated clocks and variables. So, a transition changes the state of the whole system, which defines the traces of a model that is used for both reasoning and verification of a system, together with the automatic support of UPPAAL.

Haskell [13] is the functional programming language we use for expressing, implementing and evaluating the translation technique. Both concise and expressive power of Haskell helps us in providing a precise description of the translation technique in the form of translation rules.

3. Translation Technique

This section discusses an overview of the translation technique. Then, we illustrate the technique with translating a simple specification of an Automatic Door System (ADS) in Example 3.1. Also, we present samples of using the translation rules, which summarise the translation technique. Each rule defines a translation of an operator of the `tock-CSP` operators that we consider for this work.

3.1. Overview of the Translation Technique

The translation technique produces a list of small TA, such that the occurrence of each `tock-CSP` event is captured in a small TA with an UPPALL action, which records an occurrence of an event. The action has the same name as the name of the translated event from the input tock-CSP process. Then, the technique composes these small TA into a network of TA that express the behaviour of the original tock-CSP model. The main reason for using small TA is coping with the compositional structure of tock-CSP, which is not available in TA [6]. The small TA provides enough flexibility for composing TA in various ways that capture the behaviour of the original `tock-CSP` process.

Connections between the small TA are developed using additional coordinating actions, which coordinate and link the small TA into a network of TA to establish the flow of the translated `tock-CSP` process. Each coordinating action `a!` (with an exclamation mark) synchronises with the corresponding co-action `a?` (with a question mark) to link two TA, in such a way that the first TA (with `a!`) communicates with the second TA that has the corresponding co-action (`a?`).

Coordinating Action

A coordinating action is an UPPAAL action that is not part of the original tock-CSP process. There are six types of coordinating actions:

- **Flow action** only coordinates a link between two TA for capturing the flow of the behaviour of the original tock-CSP process.
- **Terminating action** records termination information, in addition to coordinating a link between two TA.
- **Synchronisation action** coordinates a link between a TA that participate in a multi-synchronisation action and a TA for controlling the multi-synchronisation.
- **External choice action** coordinates the translation of external choice such that choosing one of the processes composed with external choice blocks the other alternative choices.
- **Interrupt action** initiates an interruption of a process that enables a process to interrupt another process that are composed with interrupt operator.
- **Exception action** coordinates a link between a TA that raises an exception and a control TA for handling the exception.

The name of each coordinated action is unique to establish correct flow. The name of a flow action is in the form `startIDx`, where `x` is either a natural number or the name of the original tock-CSP process. Likewise, the name of the remaining coordinating action follows in the same pattern `keywordIDx` where `keyword` is a designated word for each of the coordinating action: `finish` for terminating action, `ext` for external choice action, `intrp` for interrupting action, and `excp` for exception action. Similarly, the name of a synchronising action is in the form `eventName___sync`, that is an event name appended with the keyword `___sync` to differentiate the synchronisation event from other events.

Termination actions are provided to capture essential termination information from the input tock-CSP in the cases where a TA needs to communicate a successful termination for another TA to proceed. For example, like in the case of sequential composition `P1;P2` where the process `P2` begins after successful termination of the process `P1`.

For each translated tock-CSP specification, we provide an environment TA that has corresponding co-actions for all the translated events of the input tock-CSP process. In addition, the environment TA has two coordinating actions that link the environment TA with the network of the translated TA. First, a flow action that links the environ-

ment with the first TA in the list of the translated TA. Also, this first flow action is the starting action that activates the behaviour of the translated TA. Second, a terminating action that links back the final TA in the list of the translated TA to the environment TA, and also records a successful termination of the whole process.

Environment TA

An environment TA models an explicit environment for UPPAAL models. The environment TA has one state and transitions for each co-action of all the events in the original tock-CSP process, in addition to two transitions for the first starting flow action and the final termination co-action.

Also, for translating multi-synchronisation events, we adopt a centralised approach developed in [20] and implemented using Java in [5]. The approach describes using a separate centralised controller for controlling multi-synchronisation events. Here, we use UPPAAL broadcast channel to communicates synchronisation between the synchronisation controller and the TA that participate in the synchronisation.

Synchronisation TA

A synchronisation TA coordinates a translated multi-synchronisation action. The structure of synchronisation TA has an initial state and a committed state for each multi-synchronisation action. And for each of the committed state, there are two transitions. The first transition connects the initial state to the committed state. This first transition has a guard that is enable only when all the TA that participate in that multi-synchronisation become ready for the synchronisation, which enables the TA to synchronise first with the environment TA on the first transition and then immediately broadcast the synchronisation action in the second transition, which connects the committed state back to the initial state of the synchronisation TA.

When all the participating TA become ready, a synchronisation TA broadcasts the multi-synchronisation action such that all the corresponding participating TA synchronise using their corresponding co-action. The provided guard ensures that the TA synchronises with the required number of TA that participate in a multi-synchronisation action. The guard blocks the broadcast multi-synchronisation action until all the participating TA become ready, which enables the corresponding guard for broadcasting the multi-synchronisation action.

This example illustrates a translation of an automatic door system (ADS) that opens a door, and then after at least one-time unit, the system closes the door in synchronisation with a lighting controller that turn-offs the light after closing the door.

A `tock-CSP` process for modelling ADS is:

```
(open -> tock -> close -> SKIP)[|{close}|](close -> offLight -> SKIP)
```

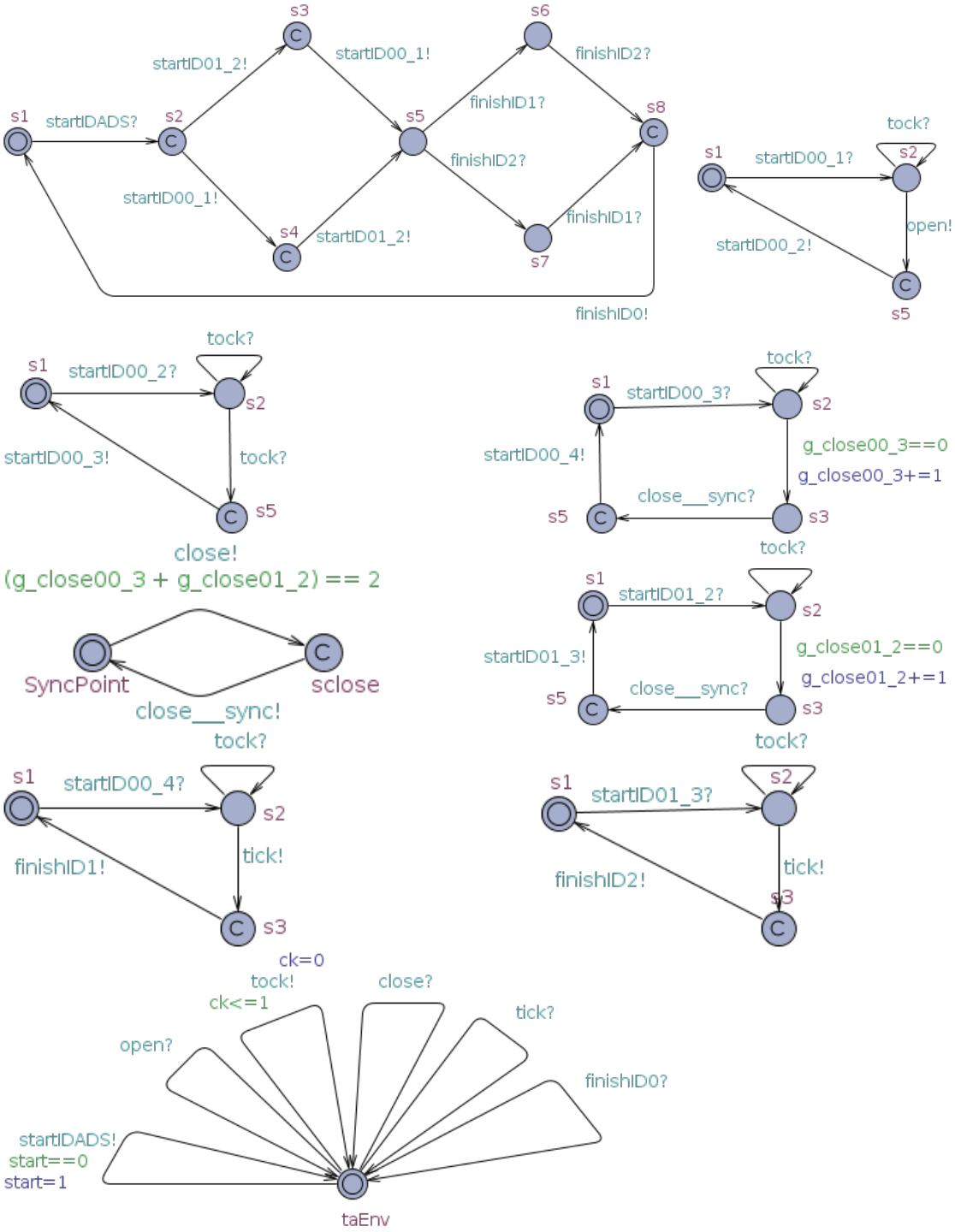


Figure 1: A list of TA for the translation of the process ADS

The list of network TA in Figure 1 expresses the behaviour of the input tock-CSP

process ADS, which opens a door and after atleast one unit time the process closes the door in synchronisation with the lighting controller that also switch-off the light after closing the door. Details of the translated TA are as follows. Starting from the top-left corner, the first TA captures the translation of concurrency that starts both RHS and LHS process concurrently in two possible orders, either RHS process then LHS process or vice versa, and then also waits for their termination actions. The second TA captures the occurrence of the event `open`. The third TA captures the occurrence of the events `tock`. The fourth TA captures the occurrence of the event `close` that needs to synchronise with synchronisation controller that is the fifth TA in the list. The sixth TA is close event of the RHS process that also needs to synchronise. The seventh and eight TA captures the occurrence of the events `tick` for both RHS and LHS process respectively. Finally, the last TA is an environment TA that provides co-actions for all the translated event.

Furthermore, in tock-CSP, a process can be interrupted by another process when the two processes are composed with an interrupt operator (`/\`). This is due to the compositional structure of tock-CSP. However, in the case of TA, an explicit transition is needed for expressing an interrupt, which enables a TA to interrupt another one. So in this translation work, we provide an additional transition for capturing interrupt using an interrupt action which as defined in the coordination action, Definition 3.1.

For example, given a process `P = P1/\P2`, the process `P1` can be interrupted by process `P2`. Thus, in translating each event of process `P1`, we provide additional transitions for the initials of the interrupting process `P2`, which enables the translated behaviour of `P2` to interrupt the translated behaviour of `P1` at any event.

Similarly, in translating external choice, we provide additional transitions that enables the the behaviour of the chosen process to block the behaviour of the other processes. Initially, the translated TA make the initials of the translated processes available such that choosing one of the processes block the other alternative processes with the co-actions of the additional transitions for external choice, as defined in Definition 3.1.

For example, given a process `P = P1[]P2` that composes `P1` and `P2` with an operator for external choice such that `Tp` is a list of TA for the translation of the process `P`. Similarly, `Tp1` and `Tp2` are list of TA for the translation of the processes `P1` and `P2` respectively. The first TA in the list `Tp1` has additional transitions for the initials of `P2` such that choosing `Tp2` blocks `Tp1`. Similarly, the first TA in the list of TA `Tp2` has additional transitions for the initials of `P1` such that choosing `Tp1` blocks the behaviour `Tp2`. Additional details of this example is available in Appendix B.

Also, in `tock-CSP`, an event can be renamed or hidden from the environment. In handling renaming, the translation technique carries along a list of renamed events. Before translating each event, the technique checks if the event is part of renamed events, and then translate the event appropriately with the corresponding new name. In the like manner, if an event is part of the hidden events the techniques carries along a list of hidden events, such that on translating a hidden event the technique uses a special name *itau* in place of the hidden event.

3.2. Translation Rules

Here we describe the translation rules. We provide a sample of the translation rules with a structure of a rule for translating operator of generallised parallel (concurrency). Details of the complete list of the translation rules together with examples are available in Appendix B

We develop the translation technique as a function `transTA`, which defines the translation of each of the BNF's constructs into TA.

Type of Function `transTA`

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

The function `transTA` has 6 parameters. The type of the parameters are `CSPproc`, `ProcName`, `BranchID`, `StartID`, `FinishID` and `UsedNames`. The first parameter of type `CSPproc` is the input CSP process to be translated. The second parameter is a name for the process, of type `ProcName`. While the third and fourth parameter are of type `BranchID` and `StartID`. Fifth parameter of type `FinishID`, is a termination ID, as defined in the coordination action 3.1. 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.

The output of the function is pairs: a list of translated TA, and a list of synchronisation actions. From the BNF of UPPAAL [17], the key elements of the output TA are the definition of TA itself and the definition of its components: `Locations`, `Transitions` and the corresponding definitions of their `Labels`. These are expressed in the following Definition 3.2 as Haskell data-type.

Type of TA and its components

```
data TA = TA Name        [Parameter] [Declaration]
             [Location]  Init        [Transition]

data Location   = Location   ID     Name   Label   LocType
data Transition = Transition Source Target [Label]
data Label      = EmptyLabel
                | Invariant  Expression
                | Guard      Expression
                | Update     [Expression]
                | Sync       Identifier   Direction
```

In Definition 3.2, the type `TA` has a constructor `TA` with six parameters. The first parameter of type `Name` is an identifier for the TA itself. The second parameter of type `Parameter` is a list of arguments for the TA. The third parameter of type `Declaration`

is a list of local definitions inside the TA. The fourth parameter of type `Location` is a list of locations in the TA, which is defined in Definition 3.2. The fifth parameter of type `Init` is an initial location of the TA. The last parameter of type `Transitions` is a list of transitions that connect the locations of the TA, also defined in Definition 3.2.

Also, in Definition 3.2, the type `Location` has a constructor that has four parameters of types `ID, Name, Label and LocType`. `ID` is an identifier for the location. `Name` is a tag that provides visual description of a location. Third parameter of type `Label` is a constraint label for the location as defined in Definition 3.2. Last parameter of type `LocType` is a format of the location, which can be one of these three: `urgent`, `committed`, `none` (which means normal location that is neither urgent nor committed).

Similarly, the type `Transition` has a constructor `Transition` that has three parameters, two parameters of type Location for both source and target location of a transition. The third parameter of type `Label` provides a label for a transition. Both location and transition have label, which can be empty (for a silent transition), invariant, guard, update and synchronisation action or combinations of them, as defined in Definition 3.2.

3.2.1. Translation of Construct for Generalised Parallel

The following Rule B.11 provides a structure of a rule for translating a construct for generalised parallel.

Rule 3.1. Translation of Generalised Parallel

```
1  transTA (GenPar P1 P2 es) procName bid sid fid usedNames =
2      ([(TA idTA [] [] locs [] (Init loc1) trans )] ++ ta1 ++ ta2,
3      (es ++ sync1 ++ sync2) )
4      where
5        idTA   = "taGenPar" ++ bid ++ show sid
6        loc1   = Location  "id1" "s1" EmptyLabel None
7        ...
8        -- Definitions of the remaining seven locations follow in
9        -- the same pattern
10
11       tran1  = Transition loc1 loc2 [lab1]
12       ...
13       -- Definitions of the remaining nine transitions follow in
14       -- the same pattern.
15
16       lab1 = Sync (VariableID (startEvent procName bid sid) []) Ques
17       ...
18       -- Definitions of the remaining five labels follow in
19       -- the same pattern.
20
21       (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
22                excps) = usedNames
23       syncEv'    = es ++ syncEv               -- Update synch names
24       usedNames' = (syncEv', syncPoint, hide, rename, exChs, intrr,
25                     iniIntrr, excps)
26       (ta1, sync1, syncMap1) =
27                transTA P1 [] (bid ++ "0") (sid+1) (fid+1) usedNames'
28       (ta2, sync2, syncMap2) =
29                transTA P2 [] (bid ++ "1") (sid+2) (fid+2) usedNames'
```

Details of Rule B.11 is as follows. Line 1 defines the function `transTA` for translating the operator generallised parallel. Lines 2-3 defines the output of the function `transTA`, that is a pair containing two elements. First is a definition of the translated TA concatenated with `ta1` and `ta2`, list of TA for the translation of the two concurrent processes `P1` and `P2`, respectively (defined subsequently in Line 26 and 28). Second, list of synchronisation action for constructing synchronisation controller. Line 5 defines an identifier for the output TA. Line 6 defines the first location of the TA. Omitted definition of the remaining seven locations follow in the same pattern. Line 11 defines the first transition of the TA. Similarly, omitted definitions of the remaining nine transitions follow in the same pattern. Line 16 defines a label for the first transition. Definitions of the remaining five labels follow in the same pattern. There are five labels because four pairs of the transitions use similar label. Line 21 extracts the synchronisation name from the used names `usedName`. Line 23 updates the synchronisation names with additional synchronisation actions `es` for the translated operator. Line 24 updates the used names `usedNames'` for subsequent translations. Line 26–29 defines

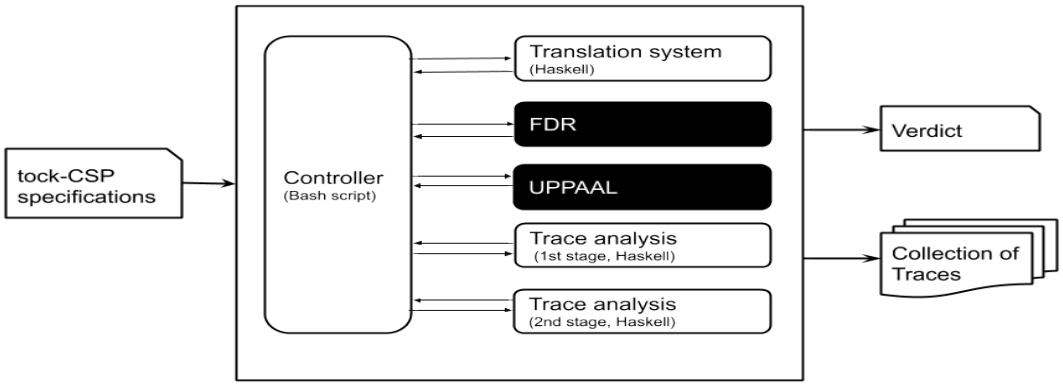


Figure 2: Structure of the trace analysis system

two recursive calls for the function `transTA` that translate the concurrent processes `P1` and `P2`.

A sample output TA of this translation rule can be seen in Example 3.1, that is the first TA in the list of the translated TA (top left).

4. Evaluation

We evaluate the translation technique in two steps. First, we implement the translation techniques into a software tool using Haskell, as a function `transform` that has the following type in Definition 4. The function takes tock-CSP process and produces a list of TA.

Function `transform`

```
transform :: NamedProc -> [TA]
```

Second, in evaluating correctness, we develop an evaluation tool, which uses the translation tool to translate tock-CSP process and then uses both FDR and UPPAAL as black boxes for generating finite set of traces for a given length. The structure of the evaluation tool is illustrated in Figure 2. Therefore, correctness of the translation technique is supported by the trace sets being the same, when the coordinating actions are deleted.

FDR was developed to produce only one trace (counterexample) at a time. So, based on the testing technique developed in [19], we developed a technique that repeatedly invokes FDR for generating traces until we get all the required traces set of the input tock-CSP. This trace generation technique enables us to generate finite traces set of tock-CSP specification with FDR.

Also, based on the testing technique developed in [18], we developed another technique for generating finite discrete traces set of TA with UPPAAL.

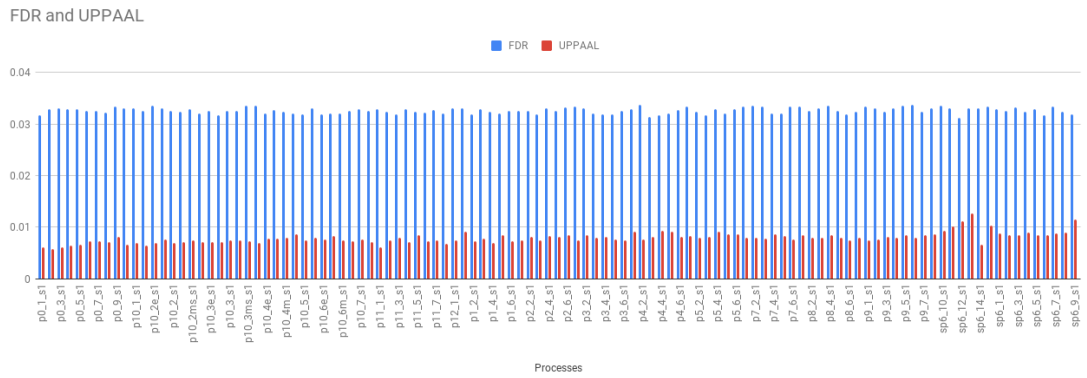


Figure 3: Performance analysis for comparing FDR and UPPAAL

So, based on this trace generation components, we develop the trace analysis in two stages. In stage 1, we generate traces of the input tock-CSP and its corresponding translation using both FDR and UPPAAL, respectively. Then, we compare the traces, and if they did not match, we move to the second stage. FDR trace generation technique distinguishes and generate traces with different permutation. In contrast, generating traces with UPPAAL does not distinguishes traces with different permutations. So, in stage 2, we use UPPAAL to check if the traces of the input tock-CSP process are acceptable traces of the translated TA.

This evaluation tool enables us to evaluate the translation technique by translating a list of formulated processes that pair all the considered constructors in the presented BNF. And then, we compare the performance of verifying deadlock freedom using both FDR and UPPAAL on the formulated processes. Details of the processes are available in a provided repository of the work [1]. Each verification was repeated 10 times, and the average timing in the repository [1]) was used to construct a graph for comparing the performance in Figure 3.

The longer (blue) bars show the average timing of FDR, while the shorter (red) bars show the average time of UPPAAL. From the graph, the average performance of FDR is above 0.03 unit time while the average performance timing for UPPAAL is below 0.01. This result shows that using UPPAAL is at least three times faster than using FDR for verification. This shows a significant performance increase of UPPAAL over FDR.

5. Related Work

Several types of researches explore using CSP in the verification of temporal specifications, especially in verifying real-time systems. Timed-CSP [23] is another popular extension of CSP that provides additional notations for capturing temporal specifications. Timed-CSP records the progress of time with a series of positive real numbers, which facilitates reasoning and verification of real-time systems.

However, the Timed-CSP does not enable specifying strict progress. Thus, traces of Timed-CSP becomes infinite, which is problematic for automatic analysis and verification [22]. So far, there is no tool support for verifying Timed-CSP models.

As a result of that, many researchers explore model transformations for supporting Timed-CSP with an automatic verification tool. Timed-CSP has been translated into tock-CSP to enable using FDR for automatic verification [21]. The work provides a link for utilising the facilities of FDR in verifying Timed-CSP model.

Also, Timed-CSP has been translated into UPPAAL to facilitates using the facilities of UPPAAL in verifying Timed-CSP. The work was initiated in [6] and then subsequently improved in [10].

Additionally, Timed-CSP has been translated into Constraint Logic Programming (CLP) that facilitates reasoning in verifying Timed-CSP, that is supported by constraint solver CLP(R) [7].

However, there is less attention in applying the same transformation techniques in improving tock-CSP with a better approach for verifying real-time system. An attempt for transforming TA into tock-CSP was proposed in [16]. In this work, we consider the opposite direction for translating tock-CSP into TA for UPPAAL to be able to utilise the facilities of UPPAAL in verifying tock-CSP models.

Apart from CSP, model transformations have been used for improving other modelling notations. Circus has been translated into `CSP||B` to enable using ProB for automatic verification [24]. Additionally, B has been translated into TLA+ for automatic validation with TLC [12]. Also, the reverse of translating TLA+ to B has been investigated for automatic validation of TLA+ with ProB [11]. Such that both B and TLA+ benefits from the resources of both TLC and ProB. Model transformation is an established field. A recent survey provides rich collections of model transformations techniques and their supporting tools [15].

6. Conclusion

In this work, we presented a translation of tock-CSP into TA for UPPAAL. The translation work facilitates using UPPAAL for the verification of tock-CSP. We find a significant increase in performance for using UPPAAL in the verification of tock-CSP specifications.

Additionally, this work would provide an easier alternative of using TCTL in specifying liveness requirements that are difficult to specify and verify in tock-CSP. Such possibility will be explored in the future work.

So far, we used trace analysis in justifying the correctness of the work. In the future, we are planning to apply mathematical reasoning to formally prove the translation technique. Secondly, we are planning to explore using the translation technique in translating extensive case studies, from tock-CSP to UPPAAL.

References

[1] A repository for the translation of tock-csp to timed automata for uppaal. Available at: https://github.com/ahagmj/tockCSP2TA [Accessed 20 Jan. 2020].

[2] James Baxter, Pedro Ribeiro, and Ana Cavalcanti. Reasoning in tock-csp with fdr.

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

[4] Patricia Bouyer. An introduction to timed automata. 2011.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Maneesh Khattri. Translating timed automata to tock-csp. In *Proceedings of the 10th IASTED International Conference on Software Engineering, SE 2011*, 2011.

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

[18] 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.

[19] 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.

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

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

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

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

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

Appendix A BNF

This section describes the characterisation of tock-CSP using BNF grammar. The following BNF in Figure 4 defines a valid syntax for constructing a tock-CSP process that we consider within the scope of this work.

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

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

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

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

Figure 4: BNF of tock-CSP for the translation technique

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 [1].

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.

[1] 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.

$e \rightarrow P$ 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.

$P \Box Q$ 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 $P \Box Q$ 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*.

$P \overset{d}{\triangleright} Q$ 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*.

$P \triangle Q$ 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.

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

Appendix B 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.

Example B.1. Here, we use an example to illustrate a definition of TA shown in Figure 5, which is defined in Listing 1 that is express using syntax of Haskell.

Figure 5: A sample output TA with two location and one transition.

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

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

Line 1 defines a TA using Definition **??** with 6 arguments. First, `"ta1"` is the name of the TA. Second and third are empty lists for both parameters of the TA 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 definition location from Definition **??**. 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 from Definition **??**. 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 **??** as an UPPAAL action with identifier ”start” that has direction Excl which specifies the acation as a sender.

In Listing 1, specifically Line 1, we use a simple name ”ta1” for the TA. However, in the coming translation rules, We formulate an approach for generating uniforms names for the small TA. We consider the structure of the AST of the tock-CSP process, which is in a form of a binary tree. So we assign an identifier to both the branch and the depth of the tree, using binary number and positive integer respectively. Then, at in the translation rules, we concatenate the two identifiers to generate a name for each small TA. Examples are provided in the translation rules, that illustrate how we formulate the names of the small TA.

Furthermore, in tock-CSP, a process can be interrupted by another process when the two processes are composed using interrupt operator (/\). However, in the case of TA, an explicit transition is needed for expressing an interrupt, which can enable a TA to interrupt another one. So in this translation work, we provide an additional transition

for capturing interrupt where there is possibility for interruption. This provision of interupt transitions begins in the translation of constant process `SKIP` (Section B.0.3) and the processes that follows. Details of interrupt together with example is provided in Section B.0.13.

Similarly, in translating external choice, we provide additional transitions for blocking non selected processes among the two processes composed with external choice. Thus, in the list of translated TA for a process that has external choice, the translation makes all the initials of the translated process available such that choosing one action blocks the other initials of the other process. Additional details with will be provided in Section B.0.9.

The above Example B.1 illustrates a simple form of the output TA produced by the translation function `transTA`. However, in the translation rule we will have TA with more than two states 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. 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.

B.0.1 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 B.1. Translation of STOP

```
1  transTA STOP 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         tran1 = Transition loc1     loc2    [lab1]  []
13         tran2 = Transition loc2     loc2    [lab2]  []
14         trans = [tran1, tran2]
15
16         lab1 = Sync (VariableID
17                         (startEvent processName (bid ++ sid)) [])
18                     Ques
19         lab2 = Sync (VariableID "tock" []) Ques
```

Rule B.1 expresses the translation of the construct STOP, which produces an output TA depicted in Figure 6. The figure illustrates the structure of the output TA that has 2 locations and 2 transitions as define in Lines 7–9 and lines 11–13 respectively.

Figure 6: A structure of TA for the translation of STOP.

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 element 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 12–14.

Finally, Line 16 – 18 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 B.1. `Lab2` is label for the second transition, which is defined as an UPPAAL action "tock" with `Ques` that indicates a receiver.

Definition B.1. Function `startEvent`

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

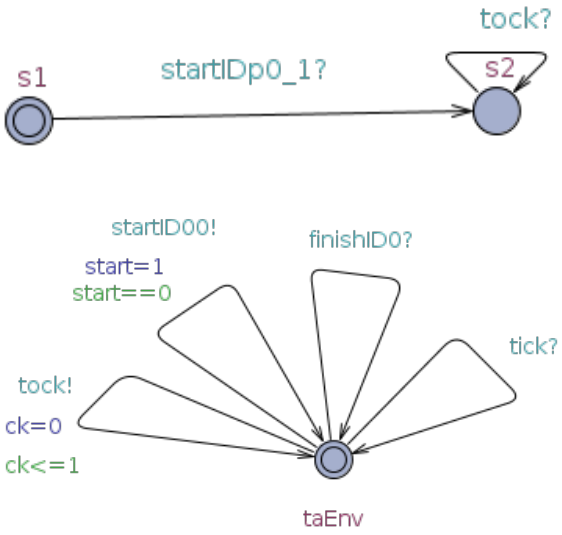
In developing the translation rules, we use a function `startEvent` Definition B.1 for generating a name for the first starting flow action of the first TA in each list of the translated TA. If the input process has an identifier, we used the identifier in the translated TA. Otherwise, the function `startEvent` 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`.

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 B.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 Ex-

ample B.2.

Example B.2. An example of translating a process STOP produces a list of TA that contains two TA in Figure **??** and **??**, as illustrated below.

```
1 transTA STOP "p0_1" 1 0 ([], [], [], [], [], [], [], ([],[])
    ) =
2           [
```




```
    ]
```

Example B.2 demonstrates a translation of the process STOP using Rule B.1, which produces a list of translated TA that has contains two TA, a small TA and its corresponding environment TA as shown in Figure **??** and **??** respectively. 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`.

B.0.2 Translation of Stopu (Urgent Deadlock)

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 B.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 B.2 expresses the translation of the constant process `Stopu` which produces an output TA that is depicted in the following Figure 7, 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 define in Lines 7–9 and Line 12 respectively.

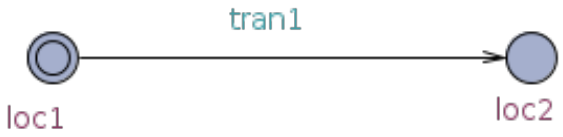


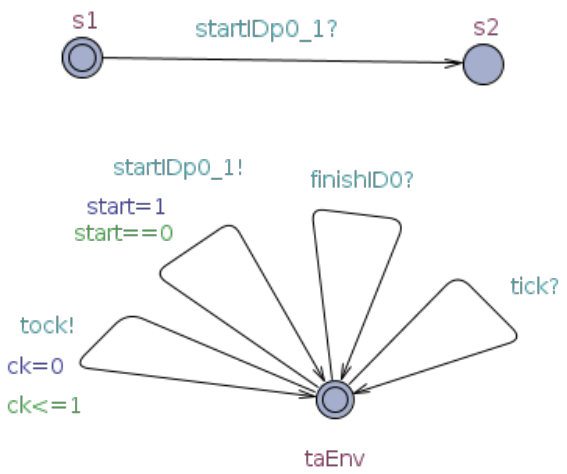
Figure 7: A structure of TA for the translation of `Stopu`

This description of Rule B.2 resembles the previous description of Rule B.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 states `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 B.3. An example for translating an urgent process Stopu.

```
1 -- transTA :: CSPproc -> procName -> BranchID  -> StartID
2 --                    -> FinishID -> UsedNames ->
3 --                       ([TA], [Event], [SyncPoint])
4 transTA Stopu "p0_1" "0" 1 0
5         ([], [], [], [], [], [], [], ([],[])) =
6             (outPutTA, [], [])
7 where
8     outPutTA = [
```




```
9           ]
```

The above Example B.5 illustrates a translation of the constant process `Stopu` according to Rule B.2. Also, this example resembles the previous Example B.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.

B.0.3 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.

Rule B.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
```

Rule B.3 describes a translation of process `SKIP` into a single output TA, which is depicted in the following Figure 8. The figure is annotated with the names used in the translation rule. The structure of the output TA has 3 states: `loc1`, `loc2` and `loc3` (define in Lines 6 – 9) and 4 transitions `tran1`, `tran2` and `tran3` (define in Lines 11 – 16).

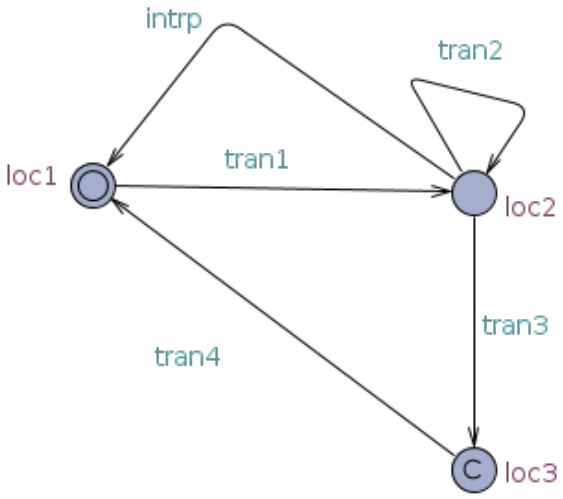


Figure 8: A structure of the TA for the translation of the process SKIP.

The behaviour of the TA begins on `tran1` for a flow action that is define using the function `startEvent` (Definition B.1). Then, the TA follows on of the 3 transitions. On transition `tran2`, the output TA performs the event `tock` to record the progress of time, and remains in the same state `loc2`. On transition `tran3`, the output TA performs the event `tick` and then immediately follows the subsequent transition `tran4` to performs 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 B.0.13. Here, we highlight a definition of the function `transIntrpt` in Definition B.2 due to its first appearance in this translation rule.

Definition B.2. Function `transIntrpt`

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

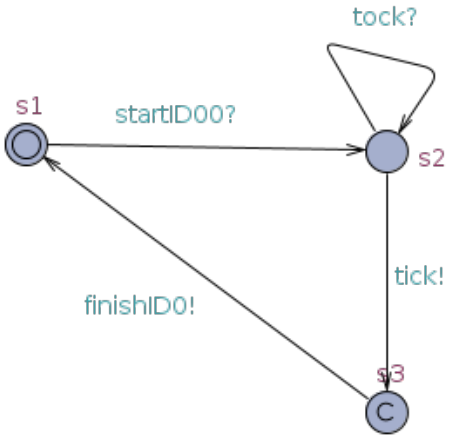
The function `transIntrpt` generates interrupting transitions using the initials of an interrupting process. The function `transIntrpt` has 3 parameters. The types of the parameters are list of event (initials of an interrupting process) and two locations that connect the interrupting transition. The 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 27). Previously, we mentioned that we will explain the names in the point where we start using the names. In this rule (Line 27), we start using the name `intrpts` from `usedNames`. We used the name `intrpts` to collect the initials of interrupting processes for constructing interrupting transitions. This completes the description of Rule B.3. This is illustrated

in the following Example B.4 for translating constant process SKIP.

Example B.4. An example for translating constant process SKIP.

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




```
        ]
```

Example B.4 illustrates using Rule B.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 illustrates translation of a process that has empty name.

Details of the output TA is 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 state s2 either the TA performs the event tock and returns to the same state; 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 empty list for the interrupting transitions. Section B.0.13 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.

B.0.4 Translation of Skipu (Urgent termination)

This section describes the translation of another constant process `Skipu`, which specifies an urgent termination that did 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.

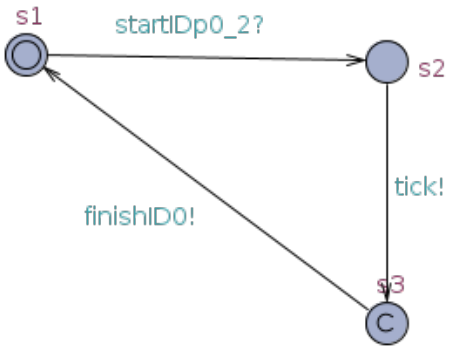
Rule B.4. Translation of Skipu

```
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
```

Rule B.4 resembles the previous Rule B.3, except that on state `s2` the output TA does not perform time `tock`. This means that the TA terminates immediately. The following example demonstrates using the rule in translating a process.

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

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




```
1                ]
```

Similarly Example B.5 resembles Example B.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 state `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 B.5, which illustrates a translation of the constant process `Skipu` for urgent termination.

B.0.5 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 is the largest translation rule because of the number of cases that need to be check in translating each prefix event. Initially, the rule checks if an event is part of either hidden event or renamed events and then translates the event accordingly. On top of this, the rule checks if the event is part of the three operators: synchronisation, external choice, interrupt or any possible combinations of these three operators. This generates additional eight cases that need to be check in translating an event. So, here we present the main part of the translation rule while the remaining details of the translation is provided in Appendix **??**.

Rule B.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.) 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.) Translation of Prefix

```
1
2     -- reset the guards in case of recursive process
3     resetG  = [Update  [(AssgExp  (ExpID (l ++ "_intrpt_guard"))
4                    ASSIGNMENT FalseExp)| (ID l) <- initIntrpts]]
5     t3c5    = Transition  loc3c    loc5    lab4e            []
6     t3c4ci  = Transition  loc3c    loc4c   l23ci            []
7     t3c4cgi = Transition  loc3c    loc4c   altIntrpt        []
8     t4c5    = Transition  loc4c    loc5   (lab2i ++ lab2d)  []
9     t4c5e   = Transition  loc4c    loc5    lab4e            []
10    t34c    = Transition  loc3     loc4c   lab4             []
11    t3c4c   = Transition  loc3c    loc4c   lab4e            []
12    t4c5i   = Transition  loc4c    loc5    l23ci            []
13    t4c5gi  = Transition  loc4c    loc5    altIntrpt        []
14    t4c6    = Transition  loc4c    loc6   (lab2i ++ lab2d)  []
15    t65     = Transition  loc6     loc5    l23ci            []
16    t65gi   = Transition  loc6     loc5    altIntrpt        []
17    t12     = Transition  loc1     loc2    lab1             []
18    t12G    = Transition  loc1     loc2   (lab1 ++ resetG)  []
19    t23     = Transition  loc2     loc3    lab2i            []
20    t2c3    = Transition  loc2c    loc3    lab2i            []
21    t23c    = Transition  loc2     loc3c  (lab2i ++ lab2d)  []
22    t23ech  = Transition  loc2     loc3   (lab2i ++ lab2d)  []
23
24    t25     = if   elem e hides
25              then Transition    loc2  loc5
26                ([(Sync (VariableID "itau" []) Excl)]) [] -- hiding
27              else Transition    loc2  loc5   lab2i    []
28    t25r    =  Transition  loc2  loc5  ([(Sync (VariableID
29                   (show new_e) []) Excl)] ++ labpath) [] -- renaming
30    new_e   =  head [newname | (oldname, newname) <- renames,
31                                          oldname == e]
32    t51     =  Transition  loc5  loc1  [lab3]          [] -- startTA
33    t33     =  Transition  loc3  loc3  [labTock]       [] -- tock
34    t44     =  Transition  loc4  loc4  [labTock]       [] -- tock
35    t35     =  Transition  loc3  loc5   lab4           []
36    t22     =  Transition  loc2  loc2  [labTock]       [] -- tock
37    t21     = [(Transition loc2  loc1  [(Sync  (VariableID
38                ((show ch) ++ "_exch")  []) Ques)] [])|ch <- exChs']
39    t23e    = [(Transition loc2  loc3  [(Guard (ExpID ((show ch) ++
40                                "_exch_ready")) )] [])|ch <- exChs']
41    t34     =  Transition  loc3  loc4   lab6  []
42    t45     =  Transition  loc4  loc5   lab4  []
43
44    lab1    = [Sync (VariableID (startEvent procName bid sid) []) Ques]
45
```

(Cont.) 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.) 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'

```

As discussed in Section A, 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 8 possible cases, which combines the 3 operators, synchronisation, external choice and interrupt. Each case has different behaviour for the prefix event. So the translation rule handle each case separately.

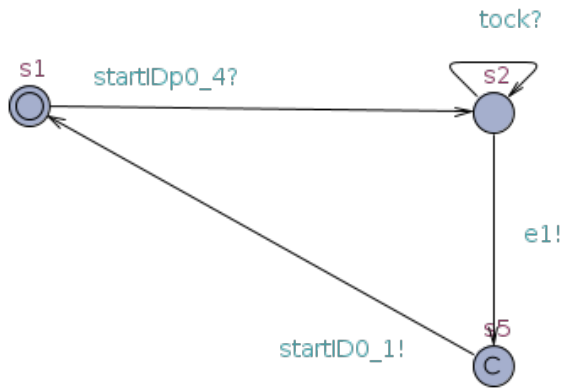
Cases 1 to 4 are cases that did not involve synchronisations. Case 1 is the simple case where a prefix event is not part of any of one the 3 operators. Case 2 is for an event that is part of interrupt, which means that the event is one of the initials of an interrupting process. Case 3 is for an event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

Cases 5 to 8 are cases that involve synchronisations. Case 5 is for the 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 is for the translation of an event that is part of all the 3 operators: synchronisation, external choice and interrupt.

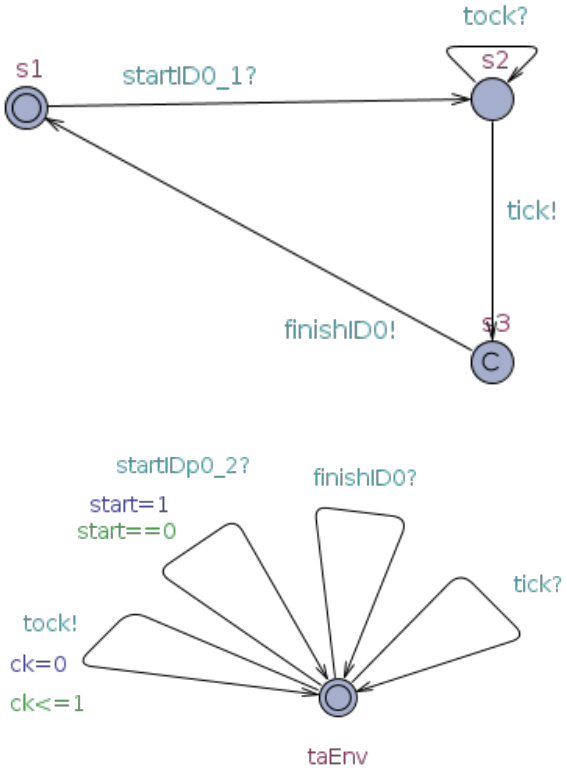
For each of these 8 cases, Rule B.0.5 defines a separate TA for translating the behaviour of a prefix event. Definition of all the states and transitions of these 8 possible TA generate a long list of definitions. Here, we present a high-level definition of the rule for the main part, which omits the details description of states and transitions of all the possible output TA for this translation rule. Details of the rule with all the completes definitions is available in Appendix **??**. The following Example B.6 illustrates using the translation rule B.0.5 in translating a process.

Example B.6. An example that demonstrates using Rule B.0.5 in translating a process `e1->SKIP`

```
1 transTA e1->SKIP "p04" "0" 0 0 usedNames =
2     [
```




```
3     ] ++ transTA(SKIP)
4   = [
```



]

Example B.6 illustrates using Rule B.0.5 in translating a process `c1->SKIP`, which is translated into a list containing two TA: TA0 and TA1 as illustrated in Figure **??** and **??** respectively. Figure **??** is a translation of the prefix event `c1` using Rule B.0.5. While Figure **??** is a translation of the subsequent process `SKIP` using Rule B.3.

The details behaviour of the output TA is as follows. Initially, `TA0` synchronises on the coordination action `startIDp04`. Then, on state `s2` either `TA0` performs the event `tock` and remains in the same state `s2` or `TA0` performs the prefix event `e1` that leads to immediately performing the subsequent coordinating event `startID01` to activate `TA1`, which synchronises on the coordination action `startID01`. Then, either `TA1` performs the time event `tock` and remains in the same state; or `TA1` performs the event `tick` which leads to performing the termination event `finishID` for a successful termination. These two TA describe the translation of process `c1->SKIP`.

B.0.6 Translation of WAIT n

This section describes the translation of process `WAIT n`, which is 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 B.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
```

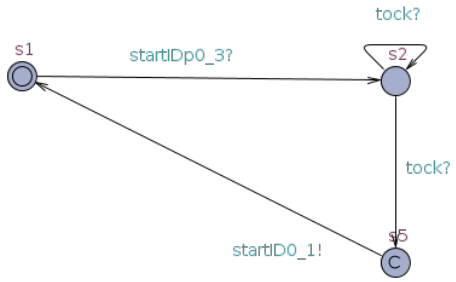
Rule B.6 describes a translation of delay process `WAIT(n)`, which is translated in terms of the translation of two constructs: `Prefix` and `SKIP`, previously discussed in Rule B.3 and Rule B.0.5 respectively. In the syntax of tock-CSP, this is specify as :

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

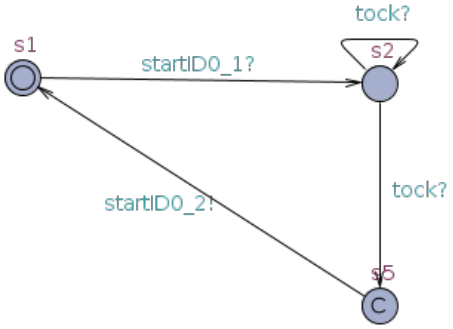
The process `WAIT(n)` is translated into a list of TA, which performs the event `tock` n times until the value `n` becomes `0` and then the TA behaves as `SKIP`. The base case is translated according to Rule B.4. While the remaining cases are translated according to Rule B.0.5. The following example illustrates using the rule in translating a process.

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

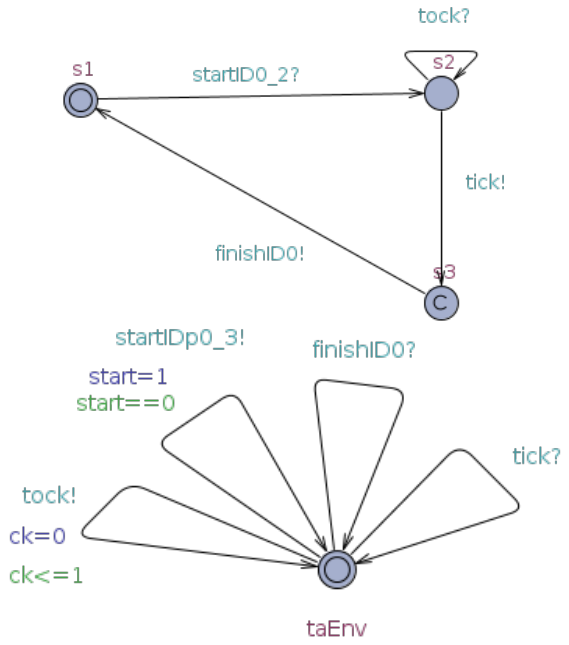
```

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




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




```
9           ] ++
10          transTA (WAIT 0) "" "0" 2 0
11                  ([], [], [], [], [], [], [], ([],[])) =
```




```
12          ]
```

Example B.7 illustrates using Rule B.6 in translating the process `WAIT(2)`. Initially, the example defines the function `transTA` for the construct `(WAIT n)` and its required arguments: process `WAIT(2)`, "p0_3" for process name, "0" for `branchID`, 0 for `startID` , 0 for `finishID` and `usedNames` for the collection of names used in the translation. The translation produces a list of TA containing 3 TA: `TA0, TA1 and TA2`.

The details behaviour of the output TA is as follow. First, `TA0` synchronises on the flow action `startIDp0_3`, which connects the environment with the first TA in the list of the translated TA. On state `s2`, `TA0` performs the time event `tock` at least ones and then performs the subsequent flow action, `startID01` which connects `TA0` and `TA1`. In this case, `TA1` is similar to the previous `TA0`. `TA1` performs the second action `tock`

and then performs another coordinating 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 state; or `TA2` performs the action `tick` and then immediately performs a terminating action `finishID0`, which indicates a successful termination. These 3 TA describe the translation of the process `WAIT(2)`.

B.0.7 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 B.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)) ] )  ]
```

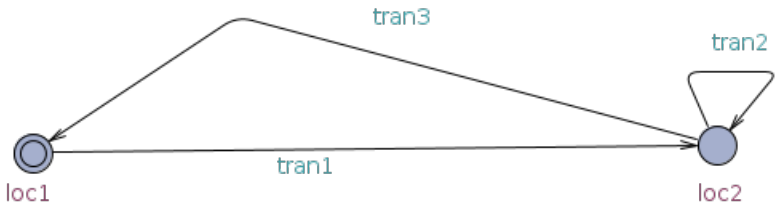


Figure 9: A structure of a TA for a translation of strict delay.

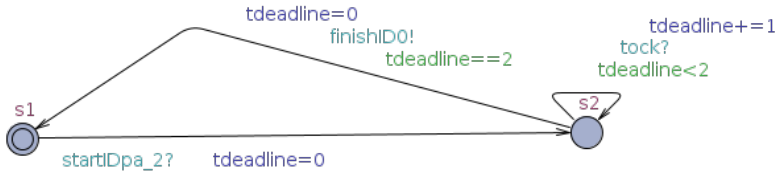
Rule B.7 describes a translation of strict delay. In Figure 9, we annotates the structure of the output TA with the names used in the translation rule: tran1, tran2, tran3, loc1 and loc2.

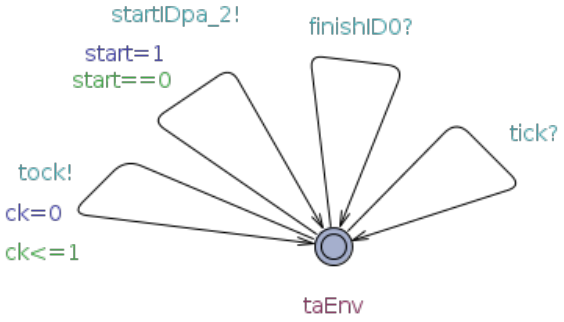
Also, this rule resembles the previous Rule B.6. Like the previous rules, Line 1 defines the function `transTA` for the construct `Waitu` and its required arguments process name, branch ID, start ID, finish ID and the names used in the translation rule. Line 2 describes the output tuple, which contains 3 element: a list of TA, a list of synchronisation actions and a list of identifiers for the synchronisation action. Line 4 defines an identifier for the output TA. Lines 6–8 define two locations of the output TA, `loc1` and `loc2`. The 2 locations are connected with 3 transitions, `tran1`, `trans2` , `trans3`, define in Lines 10 – 13. Line 14 defines an interrupt transitions, which is provided for the case of translating a process that involves interrupt. Line 16 extracts the initial of an interrupting process. Details of translating an interrupting process will be provided in Section **??**. Then, Lines 18–20 define label for the transitions. Line 22 defines a label for initialising the guard of the delay. Line 23 defines a label for updating the initial guard. Line 27 defines a label for checking the guard. Finally, Line 30 defines a label for resetting the guard.

The behaviour of the output TA begins on transition `tran1`, where the TA synchronises on a flow action (define in Line 18) and also resets the deadline timer with the expression `t_reset` (define in line 30). On state `loc2` (Line 6), either `TA` follows transition `tran2` or `tran3`. On transition `tran2` (Line 10), the TA checks the delay guard `dlguard` (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 coordinating event 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 B.8 illustrates using the rule in translating a process.

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

```
1 transTA (Waitu 2) "p0_3" "0" 0 0
2         ([], [], [], [], [], [], [], ([],[]))
3         ⇝ [
```





]

Example B.8 illustrates using Rule Rule B.7 in translating a process `Waitu 2`. The example translates the process `Waitu 2` into a TA shown in Figure **??**. In the beginning, the example applies the function `transTA` on the required parameters, `Waitu 2` for the process, `p0_3` for process name, "0" for `branchID`, 0 for `startID`, 0 for `finishID`, and a tuple of empty elements for the parameter `usedNames`. 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 Figure **??**. 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.

B.0.8 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 B.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

```

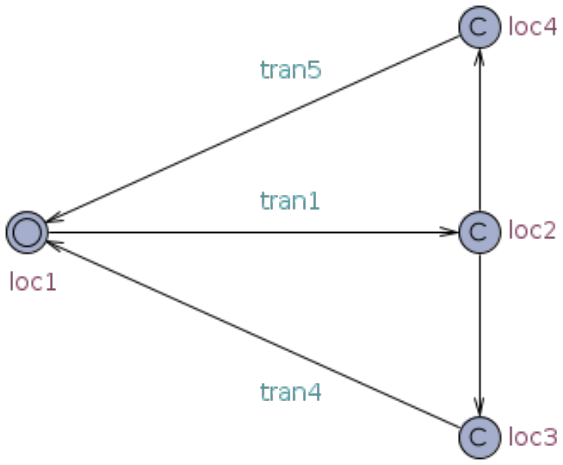


Figure 10: A structure of a TA for translating Internal choice.

Internal choice is a binary operator that combines two processes `P1` and `P2`. Rule B.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 compose with internal choice operator.

In Figure 10, we annotate the structure of the output TA with the names used in the translation rule. The output TA begins on transition `tran1` with performing a flow action. 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 `Tp1` and on transition `tran5` the TA activates `Tp2`.

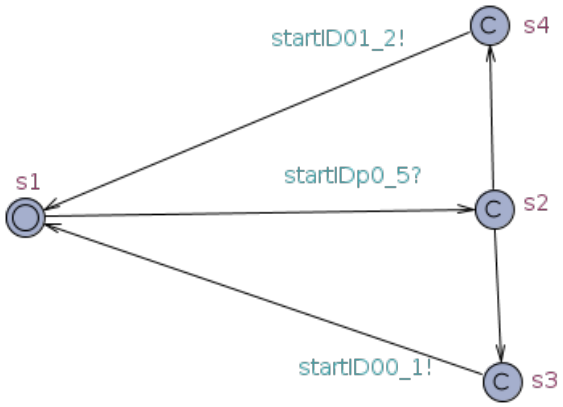
Details of Rule B.8 is as follow. Line 1 defines the function `transTA` for the construct `IntChoice` and the 5 required parameters for this rule. Line 2 describes the output tuple that contains 3 elements, a list of translated TA, a list of synchronisation actions and a list of identifiers for identifying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

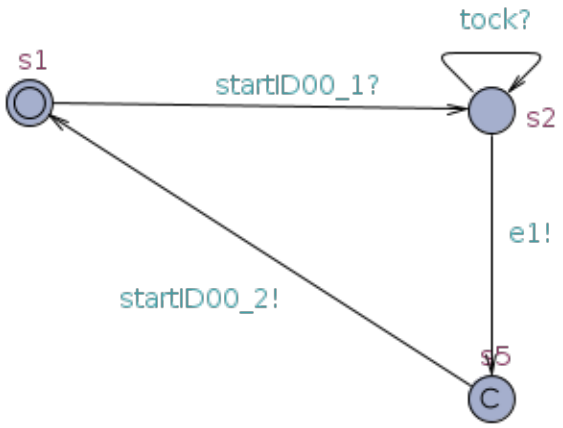
The behaviour of the output TA begins with a flow action (define in Line 20). Then, on location `loc2`, the TA follows one of the two silent transitions, that is either `tran2` or `tran3`. Transition `tran2` leads to transition `tran4`, where the TA performs another flow action (Line 22) that activates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

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

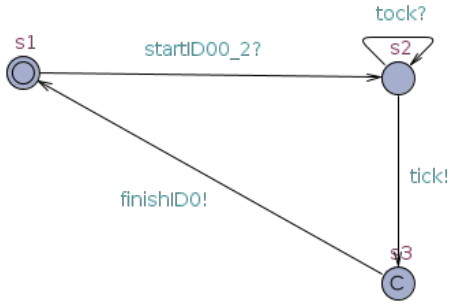
```
1 transTA((e1->SKIP)|-|(e2->SKIP)) =
2     [
```



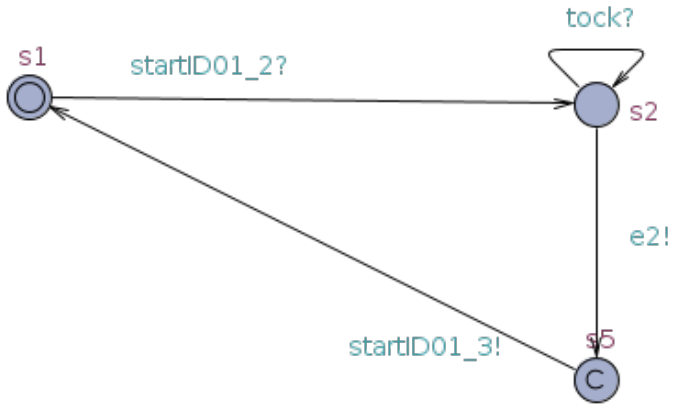

```
    ] ++ ta1 ++ ta2

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



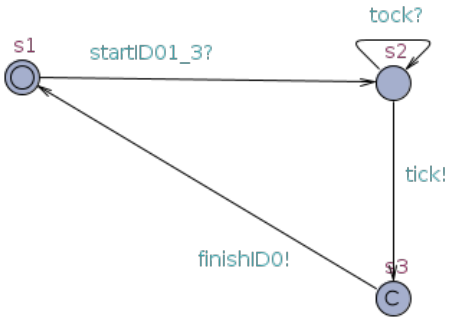

```
     ] ++ transTA(SKIP)
        = [
```

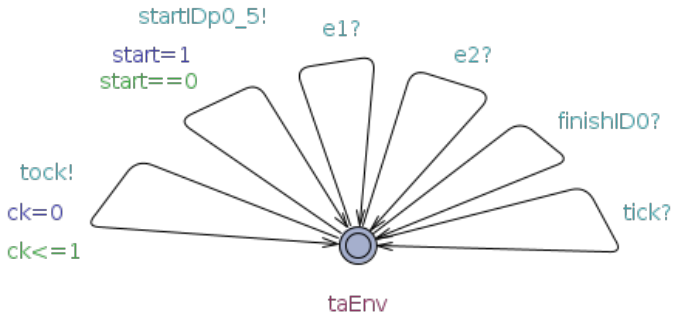


```
    ]

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




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






```
]
```

Example B.9 translates the process `((e1 -> SKIP)|-|(e2 -> SKIP))` into a list containing 5 TA: TA0, TA1, TA2, TA3 and TA4. TA0 is a translation of the operator internal choice. TA1 and TA2 are translation of the LHS process `(c1 -> SKIP)`. Where TA1 is a translation of the prefix event `e1` using Rule B.0.5. And TA2 is a translation of the subsequent process `SKIP` using Rule B.3. TA3 and TA4 are translation of the RHS process `(e2 -> SKIP)`. TA3 is a translation of the prefix event `e2` using Rule B.0.5. While, TA4 is a translation the subsequent process `SKIP` using Rule B.3. This completes the list of output TA for an example that demonstrates a translation of an operator for internal choice with translating the process `((e1 -> SKIP)|-|(e2 -> SKIP))` using Rule B.8.

B.0.9 Translation of External Choice

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

Rule B.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

      exChs'  = exChs ++ (initials p2)
      exChs'' = exChs ++ (initials p1)

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

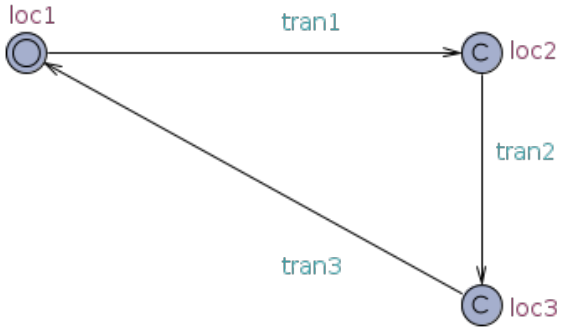


Figure 11: A structure of the control TA for the translation of external choice.

Rule B.9 defines a translation of external choice. The operator of external choice is another binary operator that combines two processes `P1` and `P2`. Rule B.9 translates the operator external choice into a TA that coordinates the list of TA for the translation of processes `P1` and `P2` that produces list of TA `Tp1` and `Tp2` respectively.

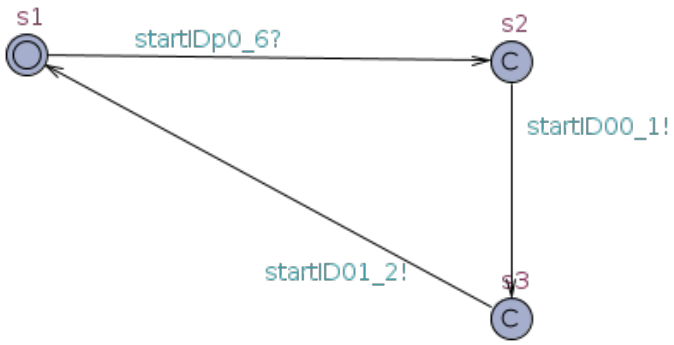
In Figure 11, we annotate the structure of the output TA with the names used in the translation rule. The output TA has 3 transitions and 3 locations define in Lines 7–10 and 12–15 respectively. Lines 17–21 define the corresponding labels of the transitions. Lines 27 extracts the initials of the external choice and updates them in Lines 30–37. Lines 39–42 define the subsequent translation of processes `P1` and `P2` that produces list of TA `Tp1` and `Tp2` respectively.

The behaviour of the output TA begins with a flow action (define in Line 12) on transition `tran1`. Then, on transition `tran2` (define on Line 13) and `tran3` (define on Line 14) the TA performs two additional coordinating actions that activate two TA: `Tp1` and `Tp2` define in lines 35–36 and 37–38 respectively. Thus, the output TA activates both `Tp1` and `Tp2` simultaneously, which makes the behaviour of both `Tp1` and `Tp2` available to the environment; in a mutually exclusive form, such that choosing one of the translated process blocks the other alternative choice. That is, choosing `Tp1` blocks `Tp2`, likewise choosing `Tp2` blocks `Tp1`.

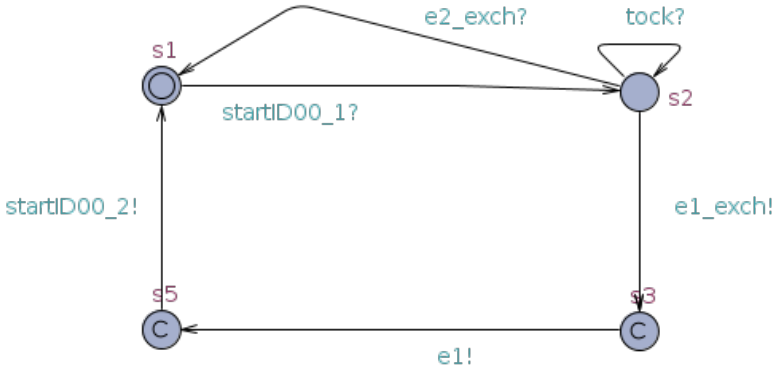
This provision of mutual exclusive behaviours is achieved with providing an additional transition immediately after the flow action. The additional transition blocks the other TA when the environment chooses the other alternative. For instance, for each initial event `ex` from the initials of the process `P1`, we construct an action by concatenating the name of the initial event with an additional key word `"exch"`; to form a name for the action in the form of `ex_exch!`. The new constructed action triggers a transition in the behaviour `Tp1`, and its co-action `ex_exch?` blocks the other alternative behaviour `Tp2`. Thus, choosing the translated behaviour of `Tp1` blocks the other alternative translated behaviour `Tp2`. Similarly, we replicate similar concept with the initials of process `P2` in choosing the translated behaviour `Tp2`. The following Example B.10 illustrates using the rule in translating a process that composes processes with the operator of external choice.

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

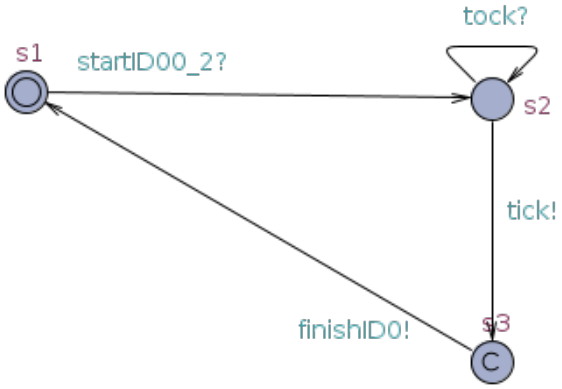
```
transTA((e1->SKIP)[](e2->SKIP))
  = [
```



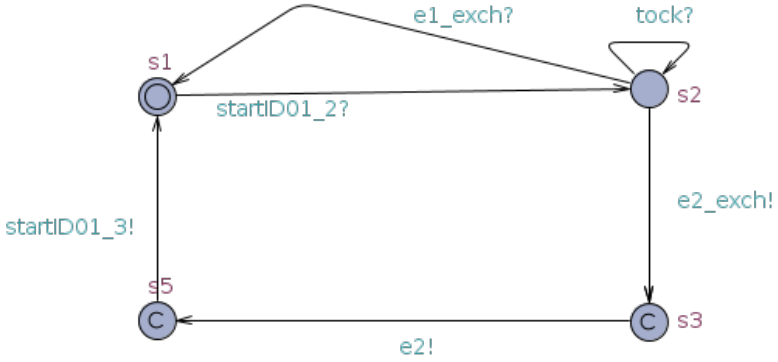

```
        ] ++ ta1 ++ ta2

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



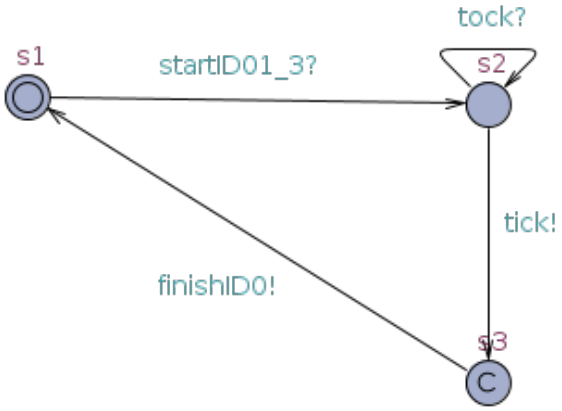

```
      ] ++ transTA(SKIP)
        = [
```

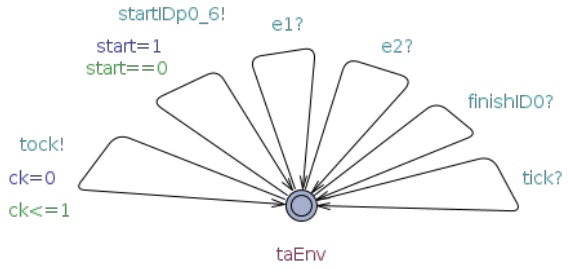



```
            ]

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




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





]

Example B.10 illustrates using Rule B.9 in translating a process `((e1->SKIP)[](e2->SKIP))` into a list of TA that contains 5 TA: TA0, TA1, TA2, TA3 and TA4. TA0 is a translation of the operator external choice. TA0 has 3 transitions, each label with a flow action, `startIDp0_6` `startID00_1` and `startId01_2`. Initially, the behaviour of TA0 synchronises on the first flow action `startIDp0_6` and then immediately performs the two subsequent flow action `startID00_1` and `startId01_2` that activate both the translations of the LHS and RHS processes, `(e1->SKIP)` and `(e2->SKIP)` respectively.

TA1 and TA2 are translation of the LHS process `e1->SKIP`. TA1 is a translation the event `e1`. The behaviour of TA1 synchronises on the flow action `startID00_1` and moves to state `s2` where it 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!` blocks the other alternative behaviour `Tp2` with the co-action `e1_exch?`, and then immediately proceeds the chosen action `e1` which leads to the subsequent flow action `startID00_2` that activates the subsequent TA `TA2`. While, TA2 is a translation of the subsequent process `SKIP` for the LHS process `e1->SKIP` which is translated with Rule B.3.

TA3 and TA4 are translation of the RHS process `(e2->SKIP)`. T3 is a translation the event `e2` using Rule B.0.5, similar way to the previous translation of TA `TA1`. While TA4 is a translation of the remaining process `SKIP` using Rule B.3. This completes the description of the translation of process `(e1->SKIP)[](e2->SKIP)` into a list of TA.

B.0.10 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 B.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 sequential composition into a control TA that coordinates the translation of the

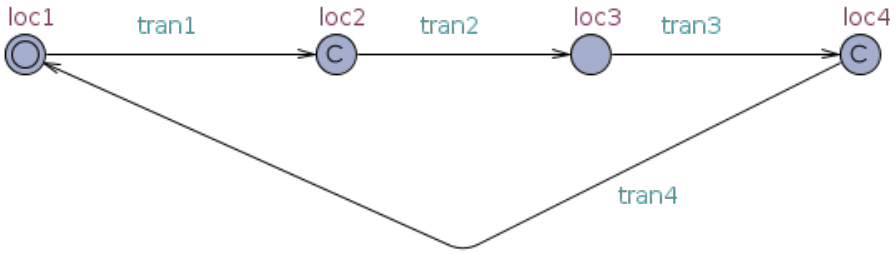


Figure 12: A structure of the control TA for the translation of the operator for sequential composition.

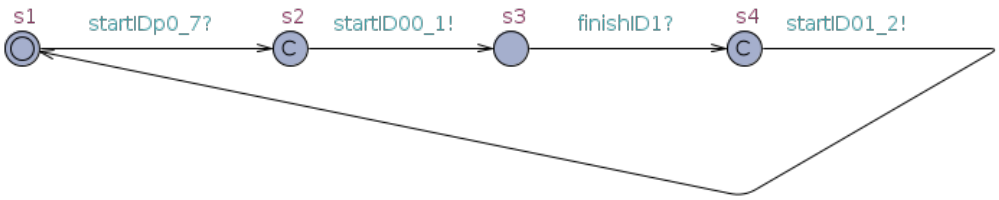
two processes `P1` and `P2` into list of TA: `Tp1` and `Tp2`.

In the following Figure 12, we annotate the structure of the output TA with names used in the translation Rule B.10, which has 4 locations and 4 transitions that are define in Lines 7–11 and Lines 13–17 respectively.

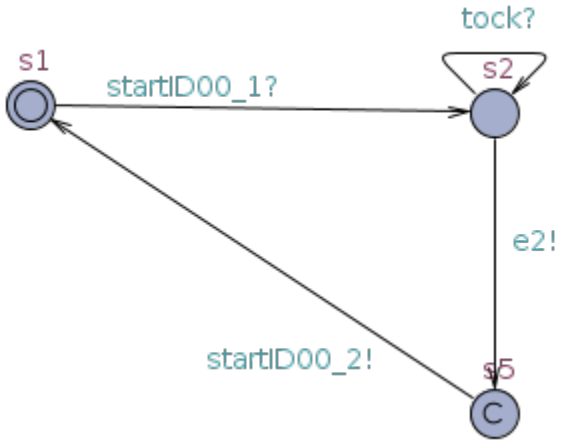
In Rule **??**, the behaviour of the output TA begins with synchronising on a flow action (define in Line 13) and then immediately performs another flow action (define in Line 14) to activate the translation of the LHS process `Tp1`. After that, the control TA waits on state `loc3` until it synchronise on a terminating action, which signals a termination of `Tp1`, and then immediately activates `Tp2` which proceeds up to its termination point. The following Example B.11 illustrates using this rule in translating a process.

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

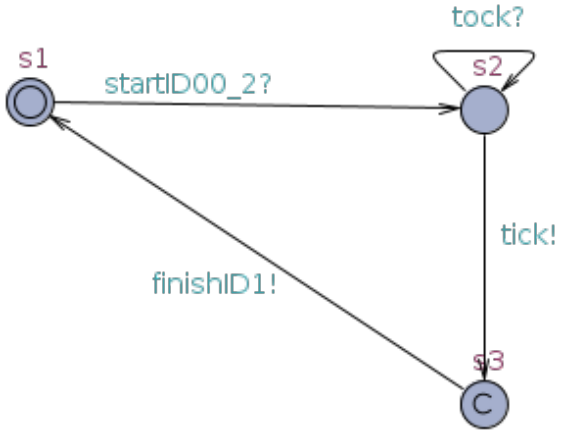
```
1 transTA((e2->SKIP);(e1->SKIP)) = [
```



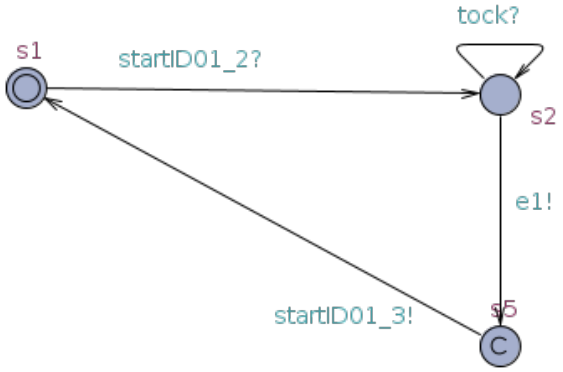

```
1     ] ++ ta1 ++ ta2
2
3 Where
4 ta1 = transTA(e2->SKIP)
5     = [
```



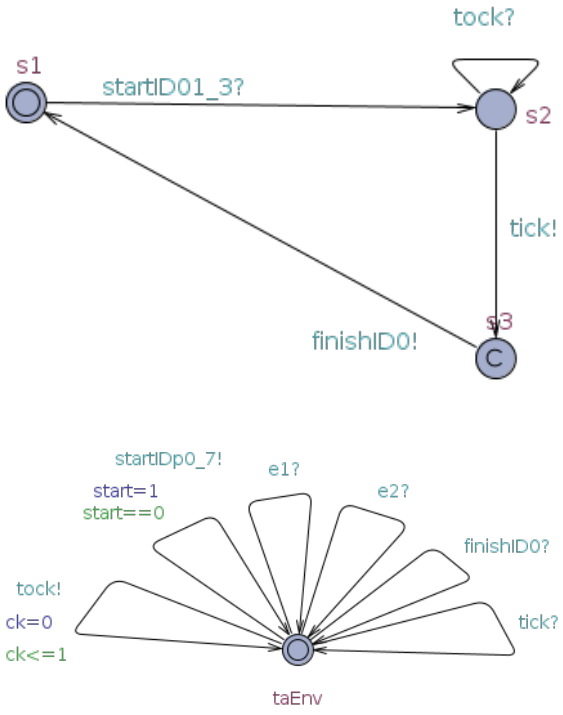

```
  ] ++ transTA(SKIP)
    = [
```




```
        ]

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




```
1       ] ++ = transTA(SKIP)
2            = [
```




```
1            ]
```

Example B.11 illustrates using Rule B.10 in translating the process `((e2-> SKIP);(e1-> SKIP))` into a list of TA that contains 5 TA: TA0, TA1, TA2, TA3 and TA4. TA0 is a translation of the operator sequential composition using Rule B.10. TA1 and TA2 are translation of the LHS process `(e2-> SKIP)`, while TA3 and TA4 are translation of the RHS process.

Details behaviour of the list of the translated TA is as follows. First, TA0 synchronises on the flow action `startIDp0_7?` and then immediately performs another flow action `startID00_1` to activate the TA `TA1`, and then waits on state `s3` until the TA synchronises on the termination action `finishID1?`; then immediately the TA performs the subsequent flow action `startID01_2` which activates `TA3` for the translation of the RHS process. In the like manner, `TA3` synchronises on the coordinating action and then performs the action `e1`, which follows with the subsequent flow action `startID01_3` that activates `TA4`. TA4 synchronises on the flow action and then performs the action `tick` follows with a termination action `finishID0` that synchronises with the co-action from the environment TA. This completes description of translating the process `((e2-> SKIP);(e1-> SKIP))` into a list of TA.

?????????????????????? STOP Revisions ??????????????

B.0.11 Translation of Generallised Parallel

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

Rule B.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 that composes two processes `P1` and `P2`. The two processes run in parallel and synchronise on the synchronisation events. The construct `GenPar` is translated into two TA: a control TA and synchronisation TA. The synchronisation TA (Definition B.3) coordinates the synchronisation of the translated processes `Tp1` and `Tp2`. The synchronisation TA is generated separately together with the environment TA after translating all the processes. While, the control TA coordinates the behaviour of the translated processes `Tp1` and `Tp2`.

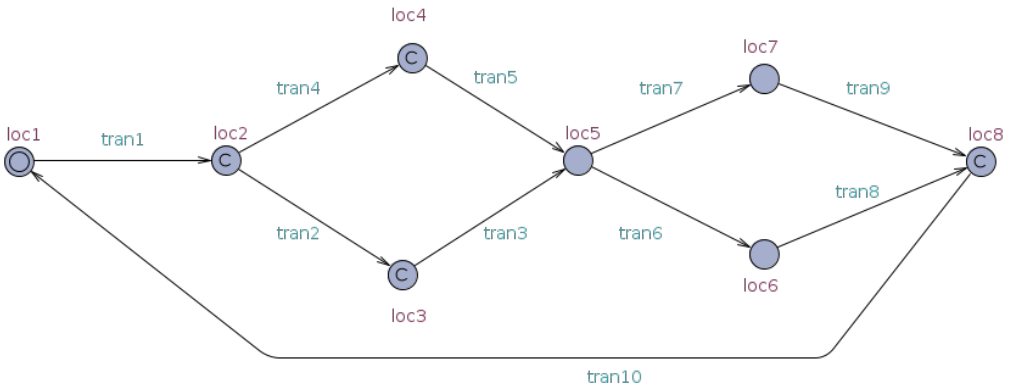


Figure 13: A structure of the control TA for the translation of operator Generallised Parallel

In Figure 13, 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 Line 38. Then, Lines 39–40 re-packet the used names into an updated tuples `usedNames'`.

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

In handling multi-synchronisation, we adopt a centralised approach developed in [20] and implemented using Java in [5]. The approach describes using a separate controller in handling multi-synchronisation. In this work, we implement the approach in a functional style with Haskell. Definition B.3 describes a function for generating a separate controller TA `syncTA` that coordinates multiple synchronisations.

In implementing the multi-synchronisation, each 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 its synchronisation controller TA `syncTA` and then waits for a synchronisation response.

For synchronisation controller, when all the required clients sent the synchronisation request, which indicates that all the synchronisation client are ready, then immediately `syncTA` communicates the corresponding multi-synchronisation action to the environment and then also immediately responds to all the awaiting client TA.

On receiving the synchronisation response, all the client TA synchronise and proceed, which enable all the multi-synchronisation processes to proceed. This is illustrated in the following Example B.12, which demonstrates using the rule in translating a process that composes processes with generalised parallel operators `GenPar`.

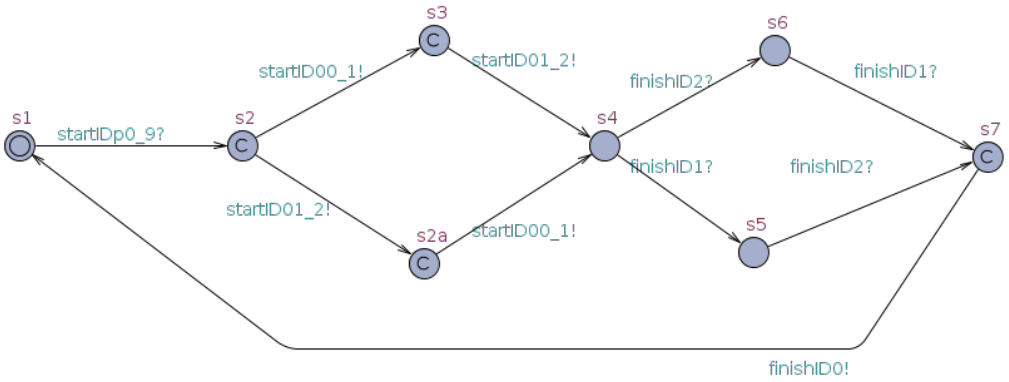
Definition B.3. Synchronisation TA

```
syncTA :: [Event] -> [SyncPoint] -> Template
syncTA     events     syncMaps   =
    Template "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)

```

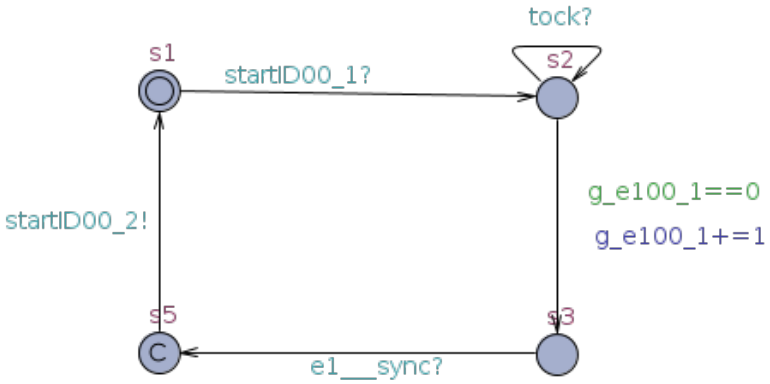
Example B.12. An example that illustrates using both Rule B.11 and the definition of synchronisation controller (Definition B.3) in translating a process `(e1->SKIP)[|{e1`

}|](e1->SKIP) into a list of TA as follows.

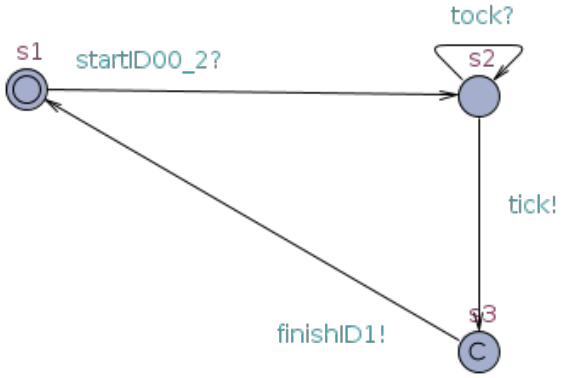
```
transTA((e1->SKIP)[|{e1}|](e1->SKIP)) = [
```



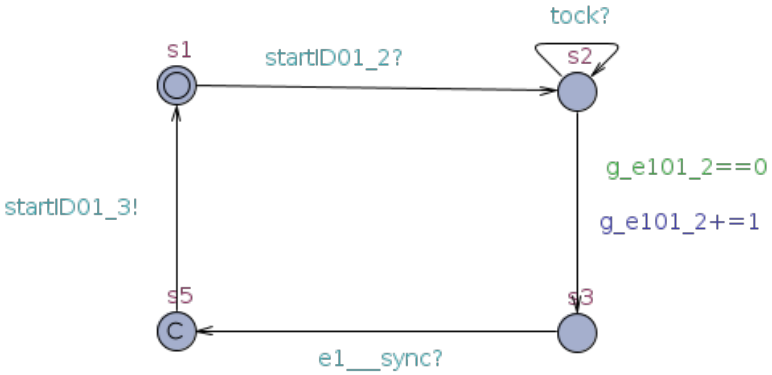

```
      ] ++ ta1 ++ ta2

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



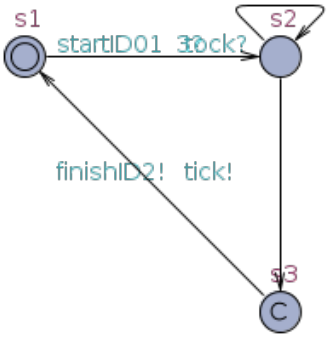

```
  ] ++ transTA(SKIP)
    = [
```

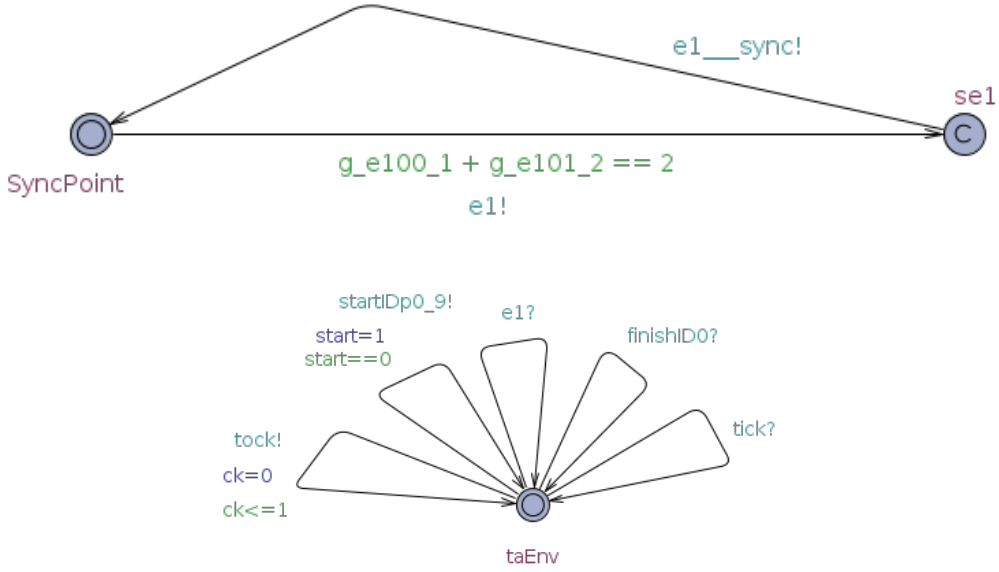



```
            ]

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




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





]

B.0.12 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.

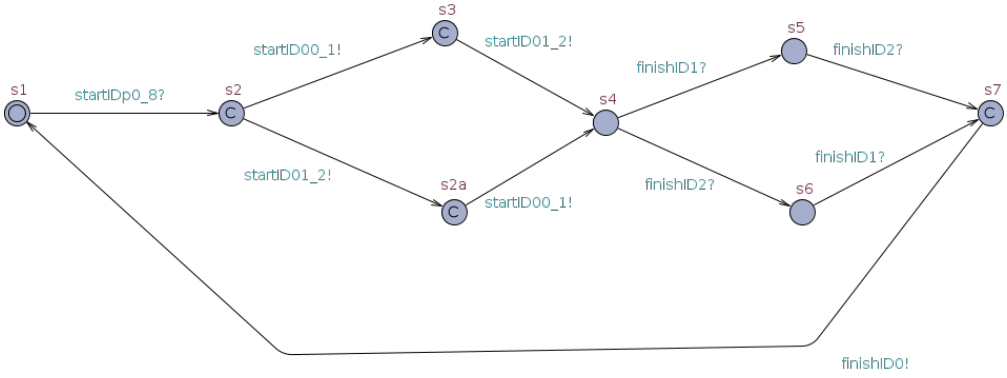
Rule B.12. Translation of Interleaving

```
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
```

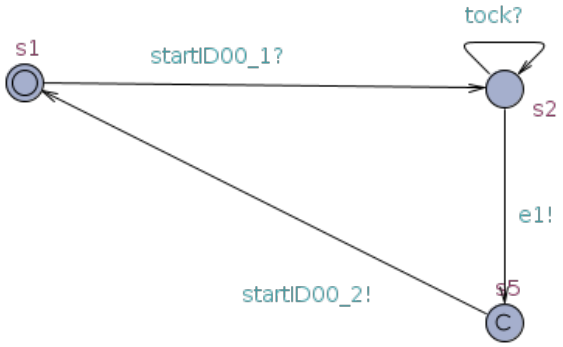
The operator interleaving is translated in terms of the constructor for generalised parallel with empty synchronisation events. In tock-CSP, this is expressed as `(P1 ||| P2) = (P1 |[{}]| P2)`. Line 1 defines the function `transTA` for the construct `Interleave` and the required parameters. While line 2 defines the output 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 B.13. An example of translating a tock-CSP process that composes processes with the operator interleaving using Rule B.12.

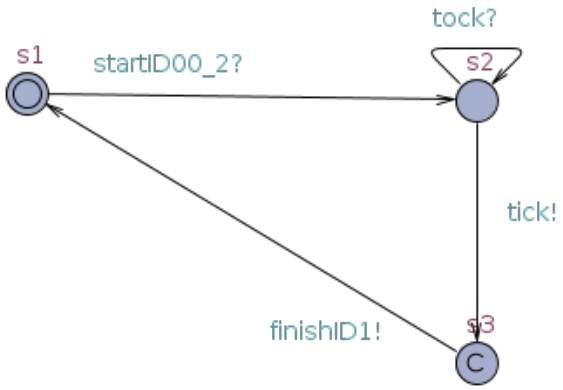
```
1    transTA((e1->SKIP)|||(e2->SKIP)) = [
```



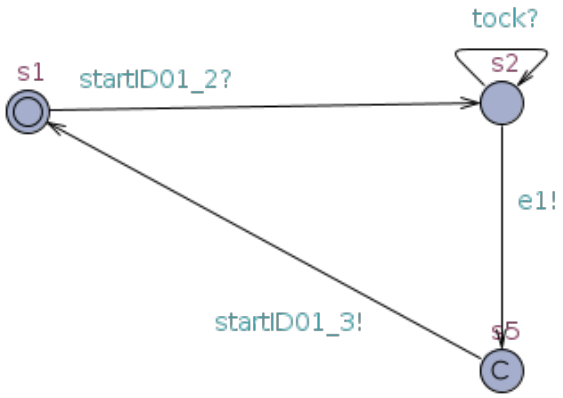

```
        ] ++ ta1 ++ ta2

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



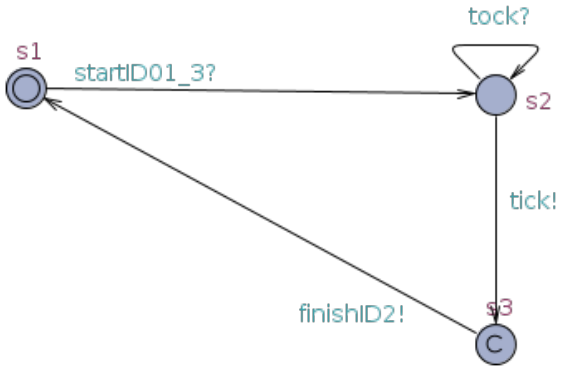

```
  ] ++ transTA(SKIP)
     = [
```

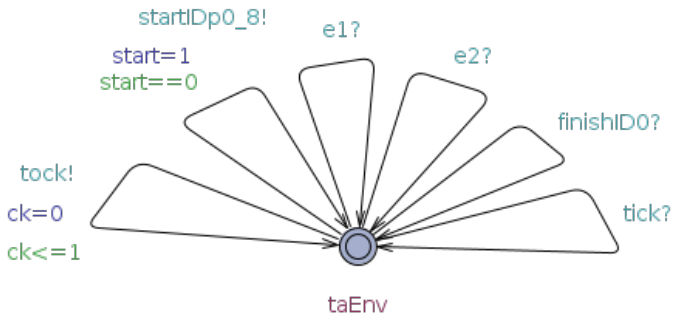


```
            ]

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




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






```
]
```

Example B.13 illustrates using Rule B.12 in translating a process `(e1->SKIP)|||(e2->SKIP))` into a list of 5 TA, TA0, TA1, TA2, TA3, and TA4. Where, TA0 is a translation of the operator Interleaving. TA1 and TA2 are translation of the LHS process `(e1->SKIP)`.TA3 and TA4 are translation of RHS process `!e2->STOP)`.

The behaviour of the TA is as follows. First, TA0 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 TA0 waits on state `s3` until it synchronises on a termination, either `finishID1` or `finishID2`. `finishID1` is termination action of the translated LHS process `(c1->SKIP)`. After that, TA0 waits on state `s4` for the termination of the translated TA for the RHS process `(e2->SKIP)` through the termination action `finishID2`. Alternatively, first, TA0 synchronises on the termination action `finishID2` for the termination of the RHS process and then waits for the second termination action `finishID1` for the termination of the LHS process.

TA1 is a translation of the prefix event `e1` using Rule B.0.5. While TA2 is a translation of the subsequent process `SKIP` using Rule B.3. Also, TA3 is a translation of the prefix event `e2` using Rule B.0.5. While TA4 is a translation of the subsequent process `SKIP` using Rule B.3. This completes the description of using Rule B.12 in translating the process `(e1->SKIP)|||(e2->SKIP)`.

B.0.13 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 B.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''

```

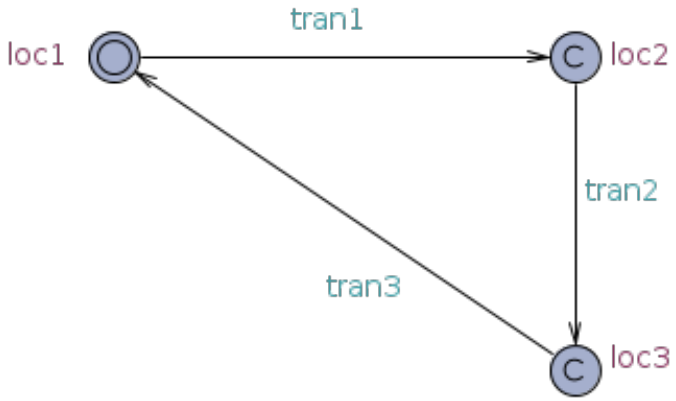


Figure 14: A TA for the structure of the translation of the operator interrupt.

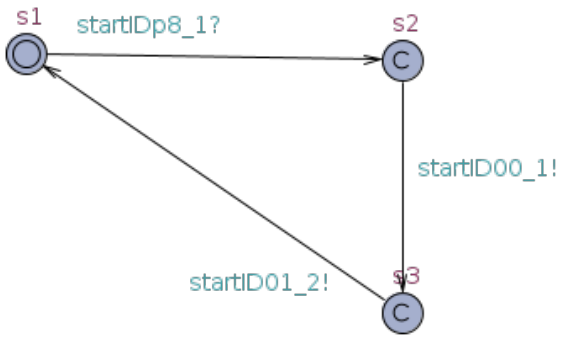
The operator Interrupt is another binary operator that composes two processes `P1` and `P2` in such a way that, process `P1` begins its behaviour but the process is interrupted by process `P2`, whenever process `P2` performs an event. The operator Interrupt is translated into a TA that coordinates the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

In Figure **??**, 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 the label of the transitions. Line 24 extracts the list of used names for interrupt. Lines 28–29 updates the used names with 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 `p1` and `p2` respectively.

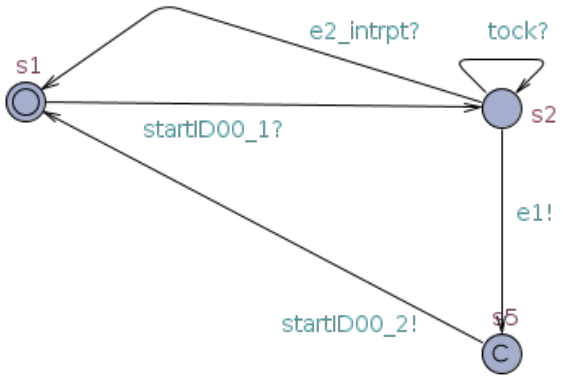
The behaviour of the control TA begins with a flow action on transition `tran1`. And then immediately activates the translation of the processes `Tp1` and `Tp2`. For the translation of the interrupted process `p1`, each TA in the list `Tp1` has an additional interrupting transition in each stable state, that is formed with the co-action of the initials of the interrupting process `p2`. The interrupting transition responds to the initials of `Tp2`, which enables `Tp2` to interrupt `Tp1` at any stable state. Definition of the additional interrupt transitions is in Rule B.0.5 for translating the construct `Prefix`, which comes in the 4 cases that involve translating interrupt. The following example B.14 demonstrates using the rule in translating a process.

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

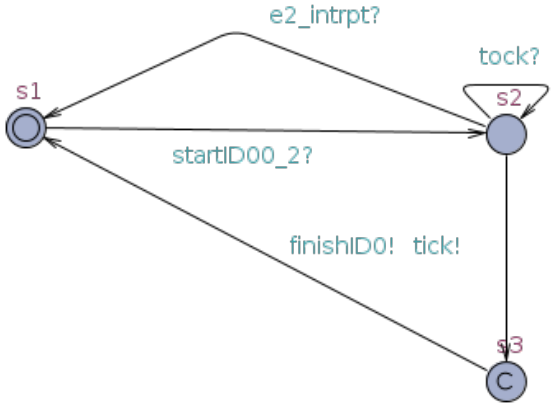
```
transTA((e1-> SKIP)/\(e2-> SKIP)) = [
```



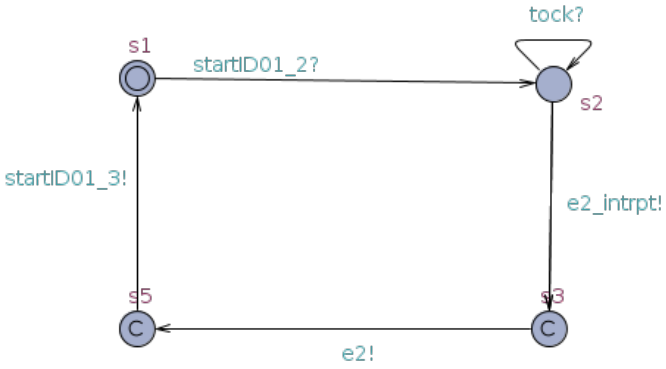

```
      ] ++ ta1 ++ ta2

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



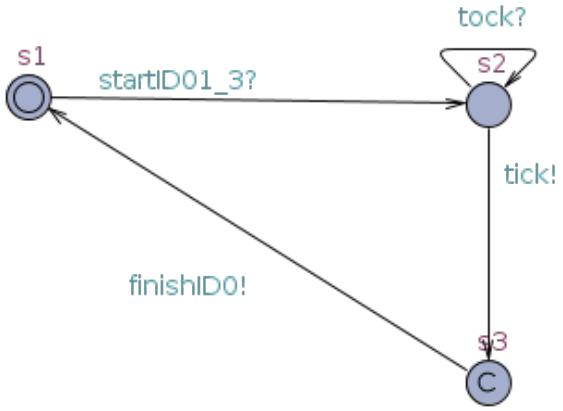

```
  ] ++ transTA(SKIP)
     = [
```




```
            ]

ta2 = transTA(c2->SKIP)
    = [
```




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



]

Example B.14 demonstrates a translation of the process ((e1->SKIP)/\(e2->SKIP)) into a list containing 5 TA: TA0, TA1, TA2, TA3 and TA4. TA0 is a translation of the operator interrupt. TA1 and TA2 are translation of the LHS process (e1->SKIP). TA1 is a translation of the event e1 using Rule **??**. And TA2 is a translation of the subsequent process SKIP using Rule B.4. While TA3 and TA4 are translation of the RHS process (e2->SKIP). TA3 is a translation of the prefix event e2. And TA4 is a translation of the subsequent process SKIP.

The behaviour of the translated TA begins with TA0, which starts with a flow action startIDp8_1, and then immediately performs two subsequent flow actions startID00_1 and startID01_2 that activate TA1 and TA3 respectively. TA1 initiates the translation of the behaviour of the LHS process (e1->SKIP). TA3 can be interrupted with the co-action e2_intrpt, which is the formulated co-action for interrupt (Definition B.2.

If TA3 interrupts the translation of the LHS process with performing its first action e2_intrpt, then the TA will immediately performs the action e2 and then performs the subsequent flow action startID01_3, which activates TA4. TA4 synchronises with the flow action startID01_3 and then on state s2 either TA4 performs the action tock or tick. The event tock records the progress of time. While the event tick leads to the final termination action finishID0. This completes the behaviours of the translated process ((e1->SKIP)/\(e2->SKIP)) without interruption.

Alternative, if TA1 is not interrupted on state s2, the TA proceeds to perform another flow action startID00_2!, which activates TA2. TA2 synchronises with its co-action startID00_2!?. Then on state s2, either the TA is interrupted with an interrupting action e2_intrpt or proceeds to perform the action tick! and then immediately perform the final termination action finishID0 that signals the termination of the process.finishID0, which indicates the termination of the LHS process (e1->SKIP), without interruption, and also the termination of the whole process ((e1->SKIP)/\(e2->SKIP)). This completes the description of the translated process ((e1->SKIP)/\(e2->SKIP)) into a list of TA that contains 5 TA.

B.0.14 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 B.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
     (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
              excps) = usedNames

     -- Updating the list of names used for exception
     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'
```

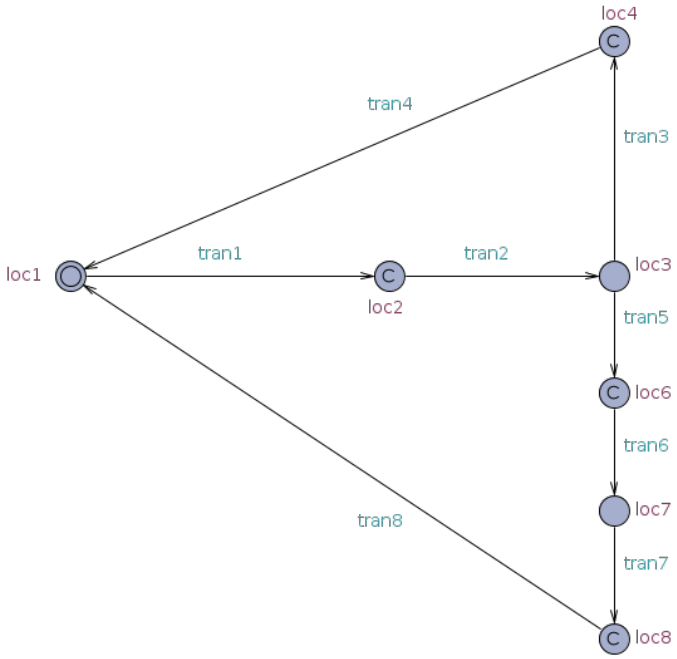


Figure 15: A structure of the control TA for the translation of the operator Exception.

The operator Exception is another binary operator that combines two processes `p1` and `p2`. Initially the process `p1` begins until either `p1` terminates or performs an exception event from the list `es` which terminates the LHS `p1` and initiates the RHS process `p2`. In the like manner, the operator `Exception` is translated into a TA that coordinates the translation of the two processes `Tp1` and `Tp2`. The translated TA initiates the translated list of TA for the LHS process `p1`, that is `Tp1`, until either `Tp1` terminates or performs an exception action that raises a special exception action `excp` (Definition **??**), which leads to activating `Tp2`.

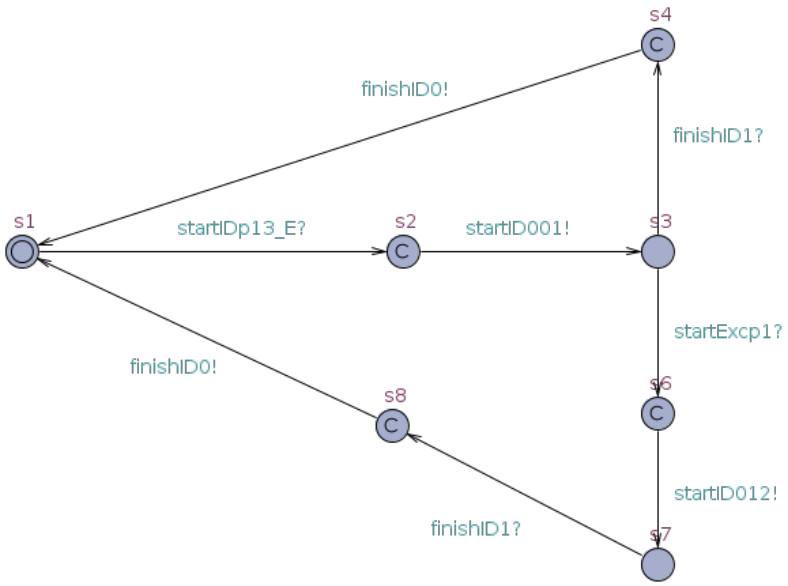
In Figure 15, we annotate the structure of the translated TA for the operator with the names used in the translation Rule B.14. The translated TA has 8 locations and 8 transitions defined in Lines 6–13 and Lines 14–22 respectively. Lines 23–31 define the labels of the transitions. Lines 32–33 extracts the used names from the tuples `usedNames`. Line 36 an updated the list of used names for the exception `excps'`. And then Lines 37–38 defined an updated used names `usedNames'`. Finally, the remaining lines, Lines 41–44 define the subsequent translation of the remaining processes.

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

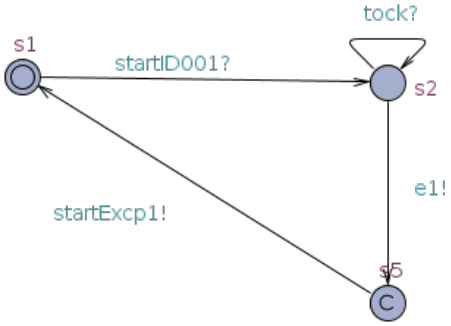
Alternatively, if `Tp1` performs an exception action before termination, the translated TA synchronises with its co-action on transition `tran5`, and then immediately initiates the translation of the RHS process `Tp2` on transition `tran5`, and then waits on state `loc7` until the translated TA synchronises on a termination action from `Tp2` on transition `tran7`. Then, on transition `tran8` the control TA performs another termination action for terminating the whole process. The following Example B.15 illustrates using the Rule B.14 in translating a process.

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

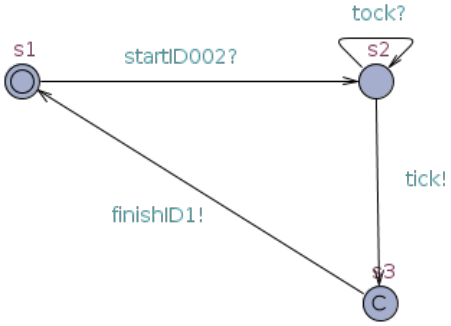
```
1 transTA((e1->SKIP)[|{e1}|>(e2->SKIP)) = [
```



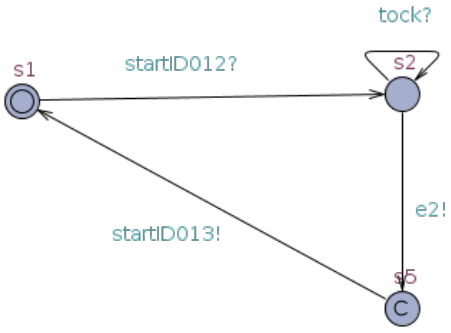

```
1       ] ++ ta1 ++ ta2
2
3 Where
4 ta1 = transTA(c1->SKIP)
5     = [
```



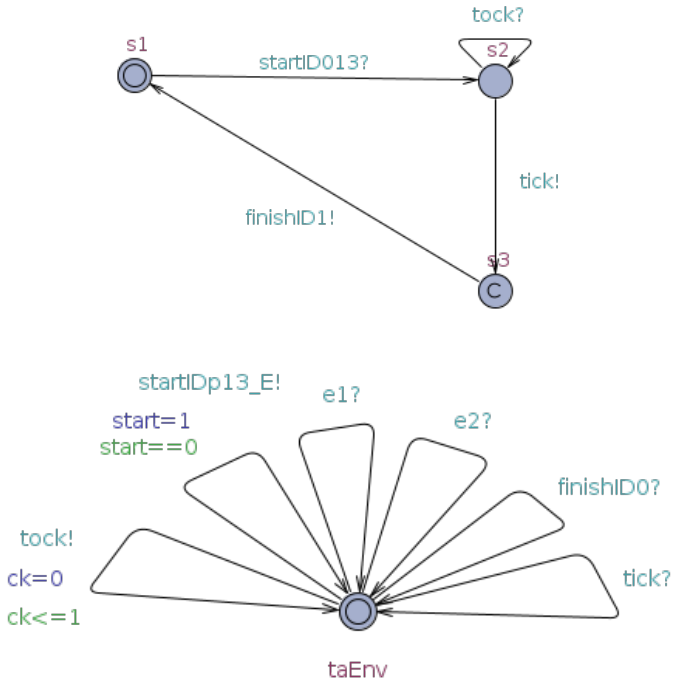

```
  ] ++ transTA(SKIP)
     = [
```




```
        ]

ta2 = transTA(c2->SKIP)
    = [
```




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



]

Example B.15 translates process `(e1->SKIP)[|{e1}|>(e2->SKIP)` into a list containing 5 TA; TA0, TA1, TA2, TA3 and TA4. TA0 translates the operator interrupt using Rule B.14. TA2 and TA3 translate the LHS process `(e1->SKIP)`. TA2 translates the prefix event `e1`. While TA3 translates the subsequent process `SKIP` TA4 and TA5 translate the RHS process `(e2->SKIP)`. TA4 translates the prefix event `e2` using Rule B.0.5. TA5 translates the subsequent process `SKIP` using Rule B.3.

The behaviour of TA0 begins with a coordinating start event `startID13_E` and then immediately performs the subsequent coordination event `startID001` which activates TA1. After that, TA0 waits on state `s3` until it receives either a termination action `finishID1` or an exception action `startExcp1`. If TA0 receives `finishID1` which indicates a successful termination of the LHS process `(e1->SKIP)`. And then TA0 performs a subsequent termination action `finishID0` to signal a termination of both processes. Alternatively, if TA0 receives an exception action `startExcp1` TA0 immediately performs a coordination action `startID012` which activates the RHS process `(e2->SKIP)`. After that, TA0 waits on state `s7` until it receives a termination action `finishID1` and then immediately performs the subsequent termination action `finishID0` to signal the termination of both processes. This completes the behaviour of TA0 for the translation of operator Exception in the process `(e1->SKIP)[|{e1}|>(e2->SKIP)`.

B.0.15 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 ex-

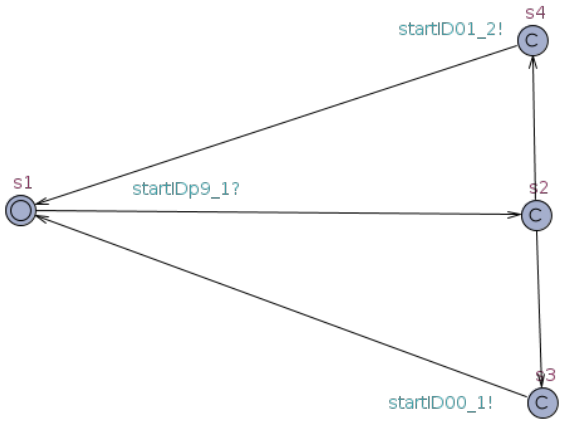
ample that illustrates using the rule in translating a process.

Rule B.15. Translation of Timeout

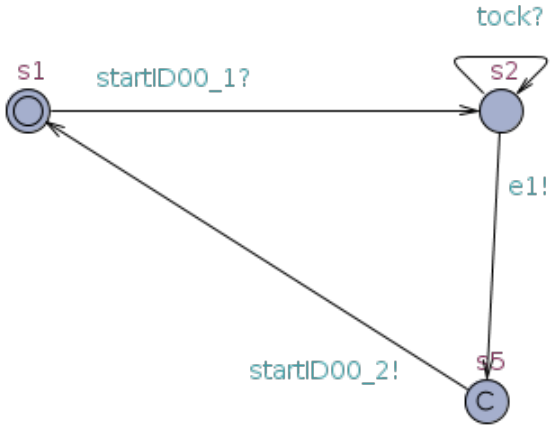
```
transTA (Timeout p1 p2 d) procName bid sid fid usedNames =
        transTA (IntChoice p1 (Seq (Wait d) p2 )) procName bid sid
                                                  fid usedNames
```

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

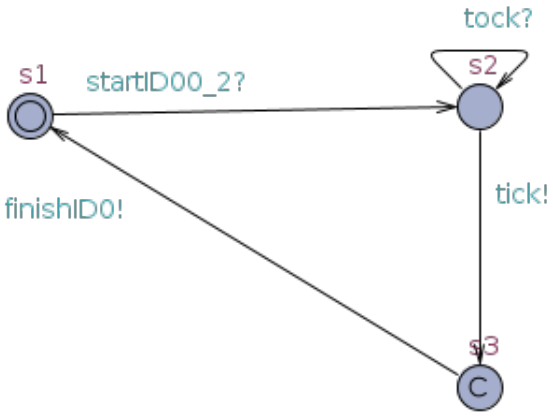
```
transTA ((e1->SKIP)[2>(e2->SKIP))=
transTA ((e1->SKIP)|~|(WAIT(2);(e2->SKIP)))= [
```




```
1       ] ++ ta1 ++ ta2
2
3  Where
4  ta1 = transTA(c1->SKIP)
5      = [
```

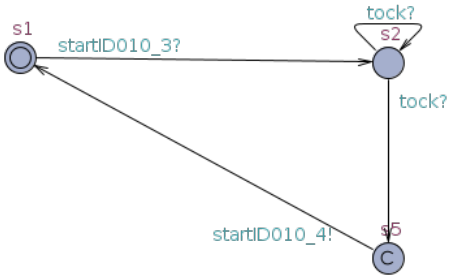



```
  ] ++ transTA(SKIP)
       = [
```

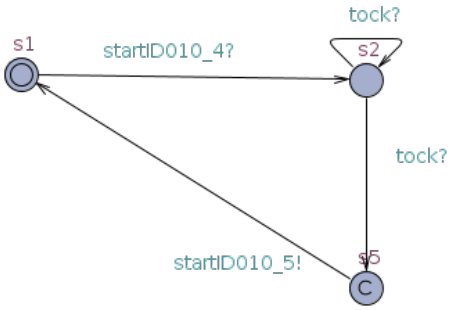



```
          ]

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

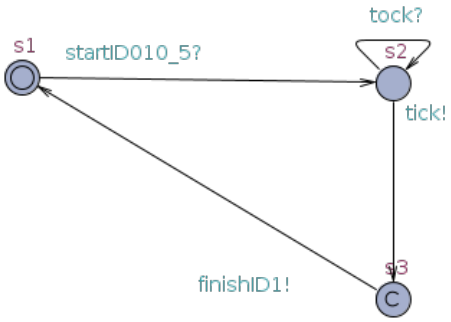
```
       ] = ta21 ++ ta22
ta21 = transTA(WAIT 2)
     = [
```



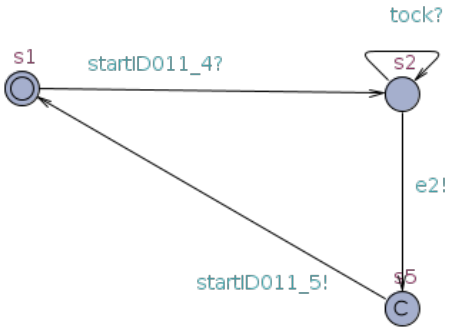

```
  ] ++ transTA(WAIT 1)
     = [
```



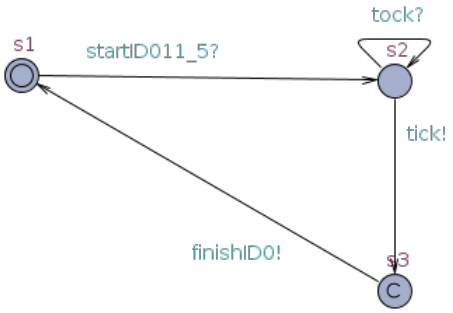

```
  ] ++ transTA(SKIP)
     = [
```

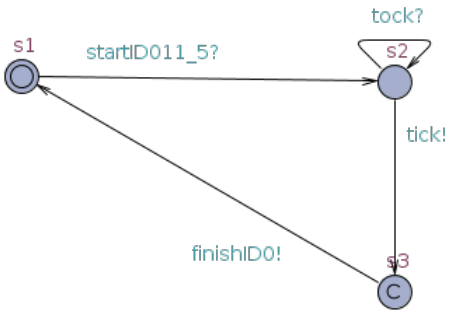



```
        ]

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




```
1   ] ++ transTA(SKIP)
2       = [
```






```
    ]
```

B.0.16 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 B.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] ++ dl_reset)   []
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
28     dlupdate = [(Update [(AssgExp (ExpID "tdeadline")
29                                   AddAssg (Val 1) ) ] ) ]
30
31     dlguard1 = [(Guard (BinaryExp (ExpID "tdeadline") Lth (Val n)))]
32     dlguard2 = [(Guard (BinaryExp (ExpID "tdeadline") Lte (Val n)))]
33     dlguard3 = [(Guard (BinaryExp (ExpID "tdeadline") Gth (Val n)))]
34
35     (_, _, _, _, _, intrpts, _, _) = usedNames
36
```

Rule B.16 translates the construct `Edeadline` into a TA. The TA has 3 locations and 4 transitions defined in lines 6–9 and lines 11–15 respectively. The following Figure 16 maps the structure of the output TA with the definitions of locations and transitions in this Rule B.16.

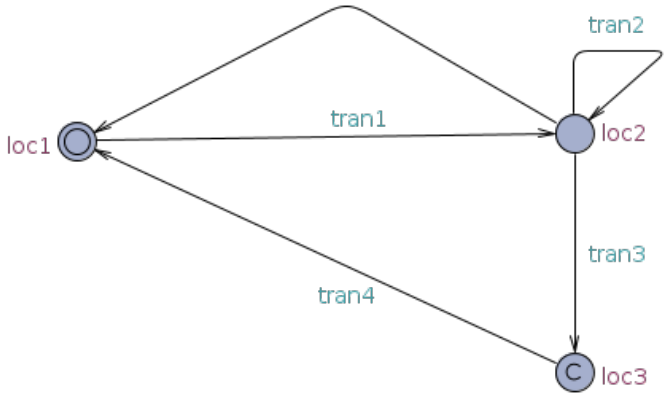
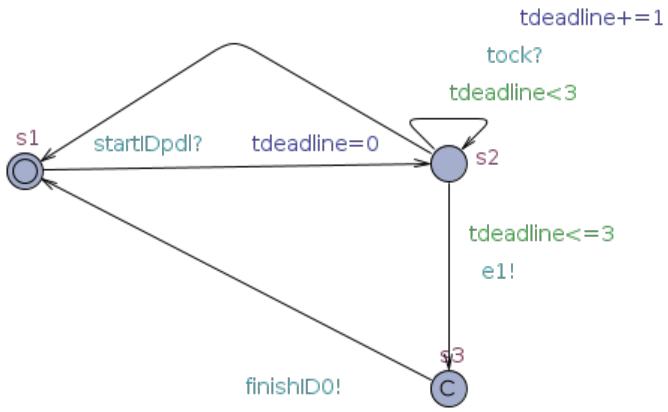


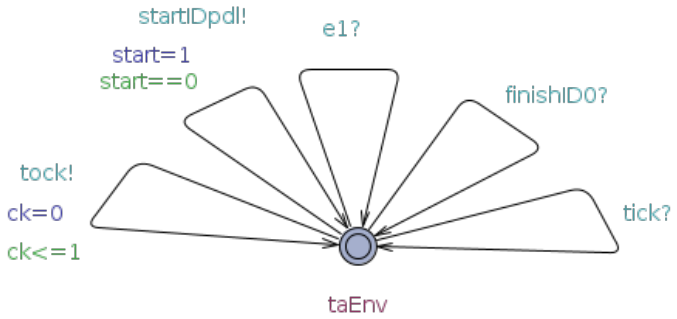
Figure 16: A structure of the control TA for the translation of the process Edeadline.

The process `Edeadline(e, d)` takes an event `e` and its deadline `d`. This is translated into TA that translate the event `e` into a TA with a deadline assign to the event. Initially, the TA begins with a coordination start event. On location `s2` either the TA performs the event `e` within the deadline `d` and then moves to location `s3`. Alternatively, after the deadline the TA blocked the event `e` and follows a silent transition to the initial location `s1`. The following example illustrates using this Rule B.16 in translating a process.

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

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





The behaviour of the TA begin with responding to a coordinating start event `startIDpd` and then reset deadline `tdeadline` to zero. After that, on location `s2` either the TA performs the event `e1` within a deadline of 3 time units. Alternatively, TA performs time event `tock` to record the progress of time. After the deadline `tdeadline` the guard `tdeadline<=3` blocks the event `e1` and the TA follows a silent transition to the initial location `s0`. This completes the behaviour of the TA for the translation of the process `Edeadline(e1, 3)`.

B.0.17 Translation of Hiding

This section describes the translation operator Hiding. 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.

Rule B.17. Translation of Hiding

```
1 transTA (Hiding p es ) procName bid sid fid usedNames =
2         transTA p      procName bid sid fid usedNames'
3         where
4            (syncs, syncPoints, hides, renames, exChs, intrrs,
5                    iniIntrrs, excps) = usedNames
6
7            -- Updates the parameter for hiding
8            usedNames' = (syncs, syncPoints, (es ++ hides), renames,
9                          exChs, intrrs, iniIntrrs, excps)
```

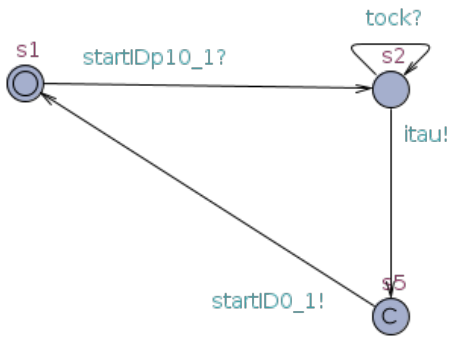
Rule B.17 updates the used name hiding, which is used in the subsequent translations. Rule B.0.5 checks the name `hides` in translating each prefix event. If an event is part of the list of hiding events in the name hides, Rule B.0.5 translates the event into a special name `tau`. If an event is not part of `hides`, Rule B.0.5 translates the event and retains its name in the translated TA. The following Example B.18 illustrates using this rule in translating a process.

Line 1 defines the function `transTA` for the construct `Hiding`. Line 2 describes the

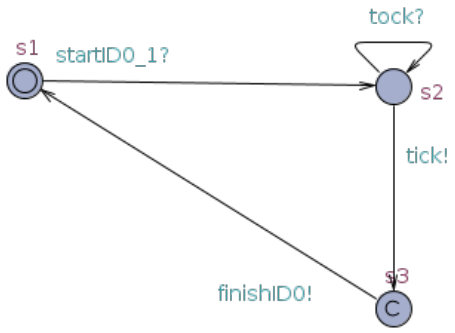
output in terms of the function `transTA` with an updated name `usedNames'`, which contains an updated name `hides`. Lines 4–5 extract the name `hides`. Lines 8–9 updates the name `usedNames`. This completes the definition of Rule B.17

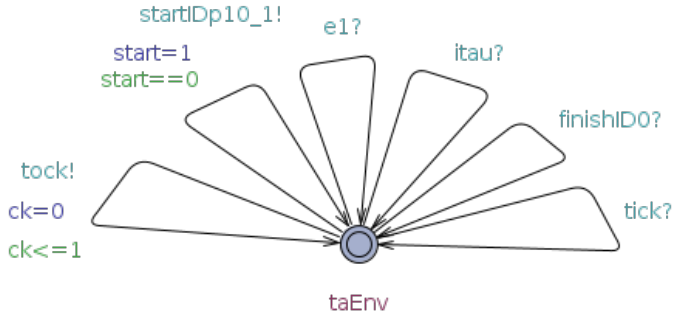
Example B.18. An example for translating a process using Rule B.17. This example translates the process `((e1->SKIP)\{e1})` into a list of TA as follows.

```
1  transTA((e1->SKIP)\{e1}   "p10_1" _ 0 0 usedNames ) =
2         transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
3      where
4      (syncs, syncMap, hides, rename, chs, intrpt, initIntrpt)
           = usedNames
5
6      usedNames' = (syncs, syncMap, [e1]++hides, rename,
7                    chs, intrpt, initIntrpt)
8
9      transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
10         = [
```




```
1      ] ++ transTA(SKIP)
2            = [
```





]

{{

Example B.18 illustrates a translation of process using B.17. The example translates the process `((e1->SKIP)\{e1})` into TA0 and TA1. TA0 translates the hiding event e1 into a special name `tau`. While TA1 translates the process `SKIP`. The behaviour of the TA begins with a coordinating start event `startIDp10_1` and then performs the event `itau` and then immediately performs the subsequent coordination event `startID0_1!`. TA1 responds and performs the event `tick` and then immediately performs termination action `finishID0`. This completes the translation of the process `((e1->SKIP)\{e1})`.

In translating the operator Hiding, the process p1 is translated according to the appropriate translation rules. But on encountering a hidden event, we replace each hidden event with a special event `tau`.

B.0.18 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 B.18. Translation of Renaming

```
1  transTA (Rename p pes) procName bid sid fid usedNames
2   =  transTA    p       procName bid sid fid usedNames'
3          where
4              (syncs, syncPoints, hides, renames, exChs, intrrs,
5                      iniIntrrs, excps) = usedNames
6
7             -- Updates the name renames in the tuple usedNames
8             --  for subsequent translations.
9             usedNames' = (syncs, syncPoints, hides, (renames ++ pes),
10                                 exChs, intrrs, iniIntrrs, excps)
```

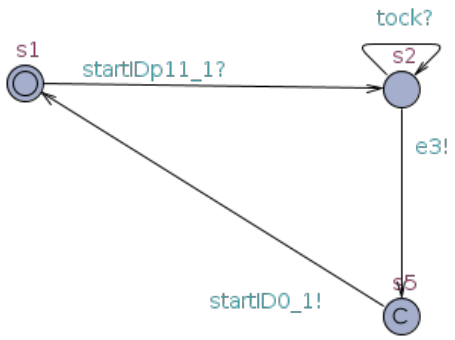
Translation of operator Renaming follows similar patterns with the previous Rule B.17, except that, this parameter is a list of pairs, an event and its new name. In this rule, we replace each rename event with its new name. Unlike the previous Rule B.17, which replaces each hiding event with a special name `tau`. Here we introduce another used name `rename` from the name `usedNames`.

Example B.19. An example for translating a process that illustrates translating the operator Renaming

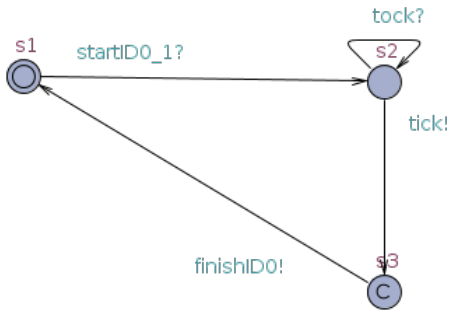
```
1 transTA(((e1-> SKIP)[[e1<-e3]]) "p10_1" _ 0 0 usedNames)
2             = transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
3 where
4     (syncs, syncMap, hides, rename, chs, intrpt, initIntrpt)
5              = usedNames
6
7     usedNames' = (syncs, syncMap, hides, rename ++ [e1, e3],
8                     chs, intrpt, initIntrpt)
```

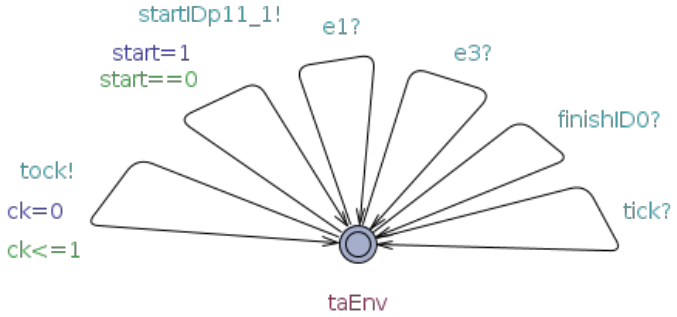
Then, we expand the translation of the prefix (line 1 above) as follows:

```
1 transTA((e1->SKIP) "p10_1" _ 0 0 usedNames')
2     = [
```




```
1     ] ++ = transTA(SKIP)
2           = [
```





]

B.0.19 Definition of Environment

This section describes the formulation of an explicit environment for the translated UPPAAL model. The following Definition B.4 defined a Haskell function for an explicit environment of the translated UPPAAL system.

Definition B.4. Function for an Explicit Environment of UPPAAL

```
env :: String -> [Event] -> Template
env    pid       es      =
  Template "Env" [] [] [loc] [] (Init loc) trans
  where
   loc = Location  "taEnv" "taEnv" EmptyLabel None
   tll = Transition loc     loc
   trans =
     [(tll [(Sync    (VariableID id []) Ques)] [])|(ID id) <- es] ++
     [ tll [(Sync    (VariableID "startID0_0"  []) Excl),
            (Guard   (BinaryExp (ExpID "start"   ) Equal (Val 0))),
            (Update [(AssgExp   (ExpID "start"   ) ASSIGNMENT
                                                     (Val 1))])] [],
     tll [(Sync    (VariableID ("startID" ++ pid) []) Excl),
          (Guard   (BinaryExp  (ExpID "start"   ) Equal (Val 0))),
          (Update [(AssgExp    (ExpID "start"   ) ASSIGNMENT (Val 1))
                   ]
           )
         ] [],
     tll [(Sync  (VariableID "finishID0" []) Ques)] [],
     tll [(Sync  (VariableID "tick"      []) Ques)] [],
     tll [(Sync  (VariableID "itau"      []) Ques)] [],
     tll [(Sync  (VariableID "tock"      []) Excl),
          (Guard (BinaryExp (ExpID "ck") Lte (Val 1))),
          (Update [AssgExp  (ExpID "ck") ASSIGNMENT (Val 0)])
          ] []
    ]

```

The function environment provides a TA that has the following transitions. One separate transition for each of the first coordinating start event. Another transition for the final termination event. Another transition for the timed event `tock`. And then a separate transition for each alphabet of the input process.

This section completes the description of the translation rules. In the next section, we will provide a justification of the translation rules using trace analysis.