Temporal Reasoning Through Automatic Translation of tock-CSP into Timed Automata (An extended report) *

Abdulrazaq Abba
Department of Computer Science
University of York, UK

Abstract

In this work, we consider translating *tock-CSP* into Timed Automata for Uppaal to facilitate using Uppaal in reasoning about temporal specifications of *tock-CSP* models. The process algebra *tock-CSP* provides textual notations for modelling discrete-time behaviours, with the support of tools for automatic verification. Similarly, automatic verification of Timed Automata (TA) with a graphical notation is supported by the Uppaal real-time verification toolbox Uppaal. The two modelling approaches, TA and *tock-CSP*, differ in both modelling and verification approaches, temporal logic and refinement, respectively, as well as their provided facilities for automatic verification. For instance, liveness requirements are difficult to specify with the constructs of *tock-CSP*, but they are easy to specify and verify in Uppaal. To take advantage of temporal logic, we translate *tock-CSP* into TA for Uppaal; we have developed a translation technique and its supporting tool. We provide rules for translating *tock-CSP* into a network of small TAs for capturing the compositional structure of *tock-CSP* that is not available in TA. For validation, we start with an experimental approach based on finite approximations to trace sets. Then, we explore mathematical proof to establish the correctness of the rules for covering infinite traces.

*The authors gratefully acknowledge the financial support of Petroleum Technology Development Fund (PTDF)

Contents

Chapter One

1. Introduction

One of the significant achievements of technology is the development of robotics systems; a category of systems that interprets the environment, takes action, moves (mobile robots) and performs a series of operations while avoiding harmful effects without human intervention. This includes the kind of systems that both interact with and learn from the environment as well as adjust their behaviour according to their current knowledge and interpretation of the environment.

The capabilities of these systems offer great promise in the development of technology, economy and improving life in general. Since robotics systems operate in the physical world and share an environment with human and other living creatures, one of the key issues of these systems is their ability to avoid harmful behaviour. In circumstances where safety is critical, it is therefore imperative that these systems are developed to minimise the possibility of undesirable behaviour. Currently, ensuring the behaviours of these systems are both correct and safe remains an open question of concern [1,2].

In the recent past, various techniques and tools have been developed to support the verification of robotics systems. In this regard, one of the promising approaches is applying rigorous validation and verification techniques to the control software. However, the effective methods, techniques and tools for checking the behaviour of robotics systems is one of the conundrums of software engineering [3]. The motivation behind this work is to discover ways of improving the verification techniques that will support the development of better robotics systems, thus increase confidence in their trustworthiness.

In the aim of addressing this problem, we take a step forward for improving the techniques and tools used for checking the temporal specification of the control software for robotics systems. In this work, we consider temporal reasoning, and we describe a formal translation of tock-Communicating-Sequential-Processes (*tock-CSP*) [4] into Timed Automata (TA) [5] that facilitates using the Timed Computational Tree Logic (TCTL) in the verification of temporal specifications, with the automatic support of Uppaal [5].

This chapter introduces the research conducted in this work. It begins with discussing the main motivation of the work, including the four directions that motivate the work: Software Engineering for Robotics in Section 1.2, Domain Specific Modelling Languages (DSML) in Section 1.3, formal methods in Section 1.3.1 and then Temporal Reasoning in Section 1.3.2; followed by Section 1.4 that describes the contributions of this work. Lastly, the chapter concludes with an outline structure of the rest of the document in Section 1.5.

1.1. Motivation

The current technology is highly dependent on software systems. Nowadays, software systems play a vital role in the operation and control of systems; including robotics. Software systems open a door for considerable achievement in the area of robotics. In recent years, robotics systems have been recognised as one of the most important technologies, as nowadays they are becoming acceptable in almost every sector: industry, school and home [6, 7]. In industry, for example, robotics systems have successfully reduced human labour in various sectors, such as automotive and electronics. In the same vein, it has been reported that robotics systems are significantly increasing productivity, and there are more predictions for this technology's continuous rise in future years [8, 9].

The capabilities and flexibility of robotic and autonomous systems (RAS) make them suitable for handling both difficult and harmful tasks [10]. They are also suitable for use in hazardous environments such as in rescue missions, mining, chemical and explosives detection [11, 12]. Additionally, they also support life and improve welfare, for instance, in assisting aged and disabled people to live independently [13].

This research is important because in this current age of technology robotics systems are moving from a controlled industrial environment to the world for universal commercial use. Thus, the need for safety becomes a crucial point of concern because these systems will be operating and interacting with the physical environment that may involve sharing an environment with humans. Importantly, the environment may include critical equipment, or the behaviour of the robotics system may involve operating critical equipment. Ensuring the safety and correctness of their operation is essential for the continuation of their success.

This can be achieved with rigorous verification and validation (V&V) techniques that check whether a system satisfies its specification correctly, both logically and temporally, before launching it into the full operation for universal purpose. For years, these V&V techniques have been used as an essential part of the Software Engineering (SE) tool kit [14].

The main objective of the research is extending and complementing the existence research for addressing the problem of verifying temporal specifications of robotics applications. Thus, we study the available resources for verifying temporal specifications of robotics applications, where we select *tock-CSP* and UPPAAL. This is because *tock-CSP* captures the generated semantics of the RoboChart models, which will remain consistent despite the evolution of the structure of the RoboChart and its tool RoboTool. This choice will have an additional advantage of enhancing the application of this research to enable other related researches based on *tock-CSP* to use our work.

1.2. Software Engineering for Robotics

In this technological era, advancement in hardware development increases the demand for software that operates the hardware correctly. Likewise, robotics increases the demand for support from SE for developing quality applications for robotics; for

complementing the robotics hardware with efficient software, which operates and controls an autonomous system safely and correctly [15,16].

The development of a software system for operating and managing robotics systems mostly involves various areas of expertise. Although the expertise involved depends on the type of system and the targeted operating environment, typically this includes SE, Artificial Intelligence (AI), Electronics, Mechanics, and Human-Computer Interaction (HCI). Combining these kinds of expertise to develop a sound software system remains a challenging task.

One of the target goals of SE is providing an abstraction that supports smooth collaboration among the various experts involved in the development of robotic software systems. This abstraction provides an effective separation of concern that decomposes a system into modules of various concerns each for a specific expertise [15]. SE techniques, such as Component-Based Software Development (CBSD), provide useful means for breaking down a complex robotics system into smaller units that are developed and managed independently [17,18]. The abstract descriptions of each unit are used to describe and design a complete system.

Also, in some cases, software systems for RAS are developed with partial information of both the system behaviour and its operating environment because robotics systems perform a series of continuous actions with partial feedback and uncertainty about the effect of each action. Additionally, in some cases, these systems are expected to operate in a dynamic environment that evolves and changes at any given time during operation. Importantly, this poses the need for exercising extreme caution when putting the system into operation.

Further, the design and development of robotics systems involve handling a variety of unrestricted requirements due to the nature of their operating environment, which comprises mostly open-ended unrestricted environments. More often, these requirements are subject to change during the system operation. These concerns create many complexities in the operation of the systems. However, these complexities can be handled and managed effectively using well-developed systematic techniques for rigorous verification.

The field of SE has a collection of well-established systematic concepts and approaches for incorporating various software aspects properly at each phase of software development. These concepts have been used successfully to develop software systems for multiple fields, such as nuclear factory, medicine, archaeology, similar to the expectation that robotics are expected to be used in multiple fields. These are also applicable to robotics, specifically in improving the quality of the robotics systems by providing better techniques for developing software systems that improve safety of robotics, while still improving both the performance and the complexity of verifying robotics applications.

In most cases, a robotics system consists of components that operate concurrently while processing a large amount of accumulated data in real-time and reacting to the operating environment within the budgeted (scheduled) time due to real-time constraints. As a result of this, most of the research focuses on improving performance and addressing complexity. Thus, less attention is paid to other important quality

issues such as verification, safety, reliability and maintenance [19].

In this case, suitable systematic techniques of SE for combining each aspect of the software properly at each developmental phase will play a vital role in balancing each aspect that will enable the production of quality software; the kind of software that is safe for use in operating robotics systems. This will improve the quality of robotics systems considerably by making them trustworthy systems. Additionally, DSML will improve further by narrowing down the scope of robotics systems and reducing the complexity of both handling the requirements and development of the systems.

In this work, we consider the definition of robotic in broader sense as described in IEEE Standard 1872-2015 [1]. As described below:

> An agentive device in a broad sense, purposed to act in the physical world in order to accomplish one or more tasks. In some cases, the actions of a robot might be subordinated to actions of other agents, such as software agents (bots) or humans. A robot is composed of suitable mechanical and electronic parts. Robots might form social groups, where they interact to achieve a common goal. A robot (or a group of robots) can form robotic systems together with special environments geared to facilitate their work.

1.3. Domain-Specific Modelling Languages (DSML)

Considering the achievement of software abstraction and Model-Driven Engineering (MDE) in SE, there are several proposals for developing an effective DSML for supporting Robotic Software System Development (RSSD) [20–22]. A DSML that is well equipped with desirable features for developing reliable robotics systems; with comprehensive formal techniques and tools for exhaustive verification and validation techniques [23].

The concept of modelling systems before their implementation remains an interesting concept that is pushing SE forward. DSML adds a step forward in providing effective facilities for defining an abstract system that hides unnecessary details. This concept of DSML is visible in three dimensions: effective separation of concern that reduces complexity, eases prototyping and overall speeds up the development process [22, 24].

Also, the concept of using DSML has been successfully tested for use in the development of software systems in various fields that produced a good result in reducing domain complexities for an appropriate suitable solution to a specific problem. Additionally, the combination of the various areas of expertise involved in robotics makes SE and specifically DSML suitable candidates for use in the development of robotic software systems [22].

In the available literature, many DSMLs have been developed and used in the development of robotics systems. Most of these languages consider providing facilities for improving capabilities of robotics systems, such as motion, recognition and planning. Among the available DSMLs in the literature, only three DSMLs considered

[1] `https://ieeexplore.ieee.org/document/7084073`

using formal techniques for verification. These are GenoM [25], DSML for Adaptive Systems [26], and RoboChart [11].

In the case of GenoM, it was initially developed as a tool for generating models from a combination of models and code, with no basis for using formal techniques. Use of formal techniques was added as an extension for improving its strength in verification. In comparison, DSML for Adaptive Systems was developed as a textual language with a very narrow scope for modelling the adaptation logic only. RoboChart was developed as a graphical language with a good foundation for using formal techniques, since its inception. Thus, RoboChart provides formal notations that are suitable for developing formal models.

In short, complementing a robotics DSML with formal techniques has provided good additional facilities for improving the quality and safety of robotics systems; more specifically, in improving the verification aspect that will reduce considerable defects.

1.3.1. Formal Methods

The use of mathematics for analysis and verification is the basis of formal method (FM), which enables using well-established theorem proving techniques and model checking for formal verification. Using FM has motivated the production of a broad spectrum of mathematical techniques for constructing and analysing models of computer systems with logical reasoning. This leads to a notable contribution to quality improvement, which has been reported as one of the main strengths and contributions of the FM in software development. Significantly, this concept improves quality and productivity, as it is widely recommended that formalisation is used for designing trustworthy systems [27–30].

In designing systems (such as robotics systems), FM emphasises the concept of creating abstract models to specify the required characteristics of systems. The abstract models are subjected to rigorous consistency checks to determine whether they meet the intended requirements. As intended, formal techniques have been successfully tested for Verification and Validation (V&V) in various industrial sectors, such as transport, business and critical systems. They have been used in various types of projects for both hardware and software. For example, FM has been used successfully in designing and verifying microprocessors, internet protocols and scheduling [19, 28, 31]. This success tallies with the expectation that robotics systems are use for various services, ranging from toys for fun to large critical industries like nuclear systems.

Using formal verification serves as a better alternative to testing in ensuring the correctness of a system. This is due to the advantage of formal validation in detecting bugs at the early development phase, at the design level, rather than at the implementation level, which is more costly [19]. This advantage plays a significant role in reducing overall project cost and time. This contribution has been achieved with the use of formal notations such as CSP [32], Z [33], and Alloy [34] that have been used for system modelling and verification using formal techniques.

In addition, the approach of using a formal method is suitable for complementing

and improving the currently used popular approaches for testing and simulation. For a long time, simulation and testing have been used in software development in various fields with a satisfactory result. However, both of these approaches are incomplete because they only check a specifically formulated scenario. It is clear that a collection of formulated scenarios or test cases only count for a fraction of the targeted operating environment [19]. This can be improved significantly by using formal techniques for checking all possible behaviour of the system to establish whether a system satisfies its specifications. Usually, this is achieved using well-established theorem-proving techniques and model checking for formal verification.

1.3.2. Temporal Reasoning

In contrast to the previous sections that discussed techniques and tools, this section focuses on the role of time in organising the flow of activities to define the behaviour of a system. Timing is an essential component of robotics systems; it is used to assemble and schedule a series of activities that accomplish a mission correctly. Therefore, investigating the correctness of system behaviour with respect to the flow of time is essential in the overall system behaviour, especially the robotics system that operates in real-time.

Depending on the nature of a system, analysing system behaviour in terms of a sequence of events could be appropriate for reasoning about untimed safety. However, in the case of timed sensitive systems, especially, the kind of systems that operate in real-time, reasoning about time is essential for its safety and correct operation. The behaviour of timed sensitive systems needs to be specified at the level of timed detail, which takes into account timing, scheduling, delay and deadline [35].

There are various approaches to modelling temporal specification. For instance, in refinement modelling approach such as in CSP notations, the facilities for addressing the timing specification was provided by extending the existing untimed notations. This is achieved by providing additional constructs for capturing time specification, such as *tock-CSP* [4] and Timed CSP [36]. In the same manner, Circus Time [37] is an extension of Circus with the notion of time. Both of these notations use the discrete approximation of time, which is appropriate at the level of computer application. However, there are specifications that can not be specified and verified in the discrete-time model.

Another popular approach of modelling temporal specification is using temporal logics that provide different constructs for verifying temporal specifications. Both modelling approaches of refinement and temporal logic are powerful approaches for model checking systems [38]. The refinement approach models both the system and its specifications with the same notation [4,36]. Temporal logic enables asking whether a system is a model for a logical formula of the specification ($system \models formula$) [39].

In the literature, Lowe has investigated the relationship between the refinement approach (in CSP) and the temporal logic approach [38]. The result of Lowe's work shows that, in expressing temporal logic checks using refinement, it is necessary to use the infinite refusal testing model of CSP. The work highlights that capturing the

expressive power of temporal logic in specifying the availability of an event (liveness specification) is not possible. Also, due to the difficulty of capturing refusal testing, automatic support becomes problematic with the previous version of the CSP supporting tool Failures-Divergence Refinement (FDR) supports refusal testing, but not its recent efficient version [40].

There are three classes of specifications that can not be express with simple refinement check. Lowe's [38] proves that simple refinement checks cannot cope with these three operators: *eventually* ($\diamond p$: p will hold in some subsequent state), *until* ($p \mathcal{U} q$: p holds in every state until q holds) and *negation* ($\neg(\diamond p)$: p will never hold in the subsequent states). All these three operators express behaviour that is captured by infinite traces. In this work, we introduce a translation tool, presented here, which will facilitate using the resources of temporal logic (and UPPAAL for automatic support) in checking *tock-CSP* models, particularly specification that are difficult to specify with refinement model.

In plain English, example of these specifications are:

1. The robot will eventually reach the target goal.

2. The robot remains in a safe state until the hazard disappear.

3. The robot will not cross the barrier until the barrier open.

Time is in continuous form, but the basis of computing happens to be in digital form, which is in discrete form. Real life is a continuous flow of events in continuous form, and correspondingly a real-environment evolves continuously with real-time. Therefore, a specification of a system that behaves and interacts with a real environment needs to be more appropriate in the hybrid form; a combination of continuous and discrete form. Precisely, in the case of reasoning, it has been reported that, the number of reachable states in modelling a system using a continuous-time model is always greater than or equal to the number of reachable states in modelling the same system using a discrete-time model [35, 41, 42]. This shows that in using discrete-time models alone, it is impossible to reason about the unreachable states. A discrete model of time has a fixed time interval (time unit). If the interval is too coarse, some reachable states will be missed. Thus, it is not possible to draw a complete conclusion about the behaviour of a system without considering the unreachable states.

On the one hand, it can be argued that finding a good enough granularity can provide an equivalent correct result, which is similar to modelling the system with a continuous-time model. On the other hand, there is no known simple way of finding enough granularity that is good enough to provide a similar correct result that is equivalent to the continuous-time model. Within the studied literature, there is no well-defined process of finding an optimal granularity that is comparable to a continuous-time model. It has been mentioned that it is likely that the process of finding the optimal granularity is as complex as solving the whole problem using the continuous-time model. In some cases, the procedure could be worse than using the continuous-time model. This is because the finer the granularity, the more it reduces

the efficiency of the verification process. In contrast, it is more likely to increase the complexity of the verification techniques [3,35].

In the literature, there is an interesting concern about handling temporal specifications. For instance, all the timed models of the relevant DSMLs found in the literature have considered using discrete-time approximation for modelling and verification of temporal specification. In conclusion, most of the modelling notations and constructs used a form of discrete time in verifying temporal specifications. Considering the safety concern for real-time systems and the uncertainties associated with discrete approximation, as highlighted above, there is a need for a better alternative approach for improvement.

1.4. Contributions

The goal of the proposed work is to facilitate verification of temporal specifications of robotics systems using UPPAAL, which provides facilities for supporting reasoning with temporal logic, especially for the verification of safety properties. These provided facilities can complement the limitations of refinement approach in expressing temporal specifications. Previous research [38] has established that reasoning with simple refinement checks has a limited scope in expressing temporal specifications; for instance, expressing liveness specifications is beyond the scope of refinement checks [38].

In the literature, many techniques and tools have been developed for improving robotics applications. There is a clear indication of less concern regarding the verification of temporal specifications. This work focuses on improving the verification tools by translating *tock-CSP* into TA that will facilitate reasoning with temporal logic in the verification of temporal specifications.

The concept of using automata for modelling system behaviour has been extended with the notion of a clock, which is capable of capturing time using an extension of automata called Timed-automata. System behaviour is modelled as a network of timed automata, which enables automatic verification of real-time systems with a well-known tool called UPPAAL model checker [5]; an integrated tool for modelling and verification of real-time systems. The tool has been developed with a promising formal approach for verifying system properties, including temporal specifications. Techniques have been developed for translating RoboChart models into *tock-CSP* that facilitates verifying system specifications using a refinement approach with the automatic support of a verification tool called FDR [40]. FDR is an automatic verification tool for CSP that also supports *tock-CSP*.

These issues raise a fundamental question in using FM for verifying robotics applications. The most fundamental question that emerged from early exploratory work was how to improve formal techniques of verifying temporal specifications that are compatible with the existing resources of developing robotics, especially the ones that have a formal basis, such as RoboChart.

The methodology we consider for this work is model transformation. To answer the research question, we plan to improve the resources used for checking the temporal specification of the control software for robotics applications. The aim of this research

work is to investigate suitable ways of using UPPAAL to verify *tock-CSP* specifications using the resources of temporal logic provided in UPPAAL.

To address the research question, we setup the following objectives:

1. The first goal is to study the available resources for verifying temporal specifications of robotics applications.

2. In the second goal, our work will add a step forward by developing techniques for automatically translating *tock-CSP* models into UPPAAL models; a kind of translation that maintains the existing semantics of *tock-CSP* in generating a correct and suitable TA model as input to the UPPAAL model checker that will facilitate verifying temporal specifications using TCTL. Therefore, existing works based on *tock-CSP*, for instance, RoboChart, will benefit from the facilities of two well-known verification tools FDR and UPPAAL. At the time of conducting this work, RoboChart is still evolving but the technique will be applicable to RoboChart because its semantics is in *tock-CSP*, and other related work around *tock-CSP*. Also, these techniques can serve as a link between the two popular model-checking tools FDR and UPPAAL, and a means of combining the two tools for system verification, with the best of the available resources, like improving performance in the verification stage.

3. The third goal is to automate the translation technique with a tool that will improve the usability of the work by making it easy to use. As a result, a significant step forward will be made in combining the facilities of both modelling approaches; as well as improving our understanding of the complex relationship between *tock-CSP* and TA, and also their modelling approaches for refinement and temporal logic, respectively.

4. The fourth goal is to sketch a mathematical proof that will provide additional justification for the correctness of the translation technique.

1.5. Structure of this Report

The rest of the document is structured as follows. Next, in Chapter 2, we present the technique we have developed for translating *tock-CSP* models into suitable TA for UPPAAL model checker. We begin with presenting a BNF for the *tock-CSP* that describes the constructs of the *tock-CSP* we consider in this work. We describe a technique for translating the constructs of *tock-CSP* into TA. Examples are provided for more detailed illustrations both in Chapter 3 and Appendices. However, the main contribution of this chapter is providing a mechanism for translating *tock-CSP* into TA, whereby we can use temporal logic in the verification of the translated behaviour of *tock-CSP*.

In Chapter 3, we describe the approach we consider in evaluating the translation technique. We describe the automation of the translation technique, as well as another technique we developed for evaluating the translation technique. We use two

categories of test cases: a large collection of formulated processes that capture interesting cases and a list of systems from the literature. We illustrate our plan of using mathematical proof to improve the justification of the translation technique.

Finally, in Chapter 4, we provide a general summary and conclusion of the work. The chapter begins by summarising the contents, then highlights recommended directions of future work for expanding the research.

Chapter Three

2. Translation Technique

This chapter describes the technical part of this research. We discuss the technique we developed for translating *tock-CSP* models into Uppaal models. First, we begin with characterising *tock-CSP* as a language for capturing the semantics of RoboChart models. Second, we describe our strategy for developing the translation technique. Third, we present the translation rules for translating *tock-CSP* into Timed Automata (TA). Finally, we discuss a justification of the translation rules using trace semantics, and then conclude the chapter with a final consideration of the translation technique.

In Section 2.1, we characterise *tock-CSP* as a language for modelling temporal specifications. This section presents a BNF for the language, together with well-formedness conditions for the BNF. As part of this work, we implement an Abstract Syntax Tree (AST) of the BNF using Haskell, which we use for both descriptions of the translation rules and also an implementation of the translation technique. Listing 2.1 presents the relevant part of the Haskell implementation of the AST.

In this work, we consider translating *tock-CSP*, which captures the semantics of the RoboChart. This is because RoboChart is at the development stage during this work and it will continue to evolve that may lead to changing its structure by the time we complete this work. However, the semantics of the structure will always confirm to the constructs of *tock-CSP* as used in this translation work. As such, our work will apply to the RoboChart despite its continues development. Besides, this decision expands the application of our work to cover other relevant work around *tock-CSP*, such that any work based on *tock-CSP* will benefit from our work.

In Section 2.2, we describe the strategy we follow in developing the translation technique. We describe our approach of using small sizes TA to capture the semantics of *tock-CSP*. To ease understanding of the approach, we provide an example that demonstrates the translation strategy in translating *tock-CSP* processes into Uppaal models.

In Section 2.3, we describe the translation rules we develop for translating *tock-CSP* models into suitable Timed Automata for Uppaal. For each construct of the BNF, we provide a rule for translating the construct into TA. For concise presentations of the rules, we used Haskell code in precisely presenting the translation rules. Also, we provide examples that illustrate using each rule in translating *tock-CSP* processes. Additional examples are also provided in Appendix C.

2.1. Characterisation of tock-CSP

This section describes the characterisation of *tock-CSP* using BNF grammar. The following BNF in Figure 1 defines a valid syntax for constructing a *tock-CSP* process that we consider within the scope of this work, then, follow with Haskell implementation of the BNF in Definition 2.1.

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

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

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

Figure 1: BNF of *tock-CSP* for the translation technique

The BNF is implemented into AST using Haskell in the following Definition 2.1.

Definition 2.1. Data definition of `CSPproc`

```
data CSPproc = STOP
             | Stopu
             | SKIP
             | Skipu
             | WAIT        Int
             | Waitu       Int
             | Prefix      Event    CSPproc
             | IntChoice   CSPproc  CSPproc
             | ExtChoice   CSPproc  CSPproc
             | Seq         CSPproc  CSPproc
             | Interleave  CSPproc  CSPproc
             | GenPar      CSPproc  CSPproc [Event]
             | Timeout     CSPproc  CSPproc Int
             | Interrupt   CSPproc  CSPproc
             | Exception   CSPproc  CSPproc [Event]
             | Hiding      CSPproc  [Event]
             | Rename      CSPproc  [(Event, Event)]
             | Proc        NamedProc
             | 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 set of events. Lastly, the parameter d represents a *CSP* expression that evaluates to a positive integer [2].

STOP specifies a process at a stable state in which only the event tock is allowed to happen. This means that the process *STOP* 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

[2]Additional details is available in `https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/syntax.html#csp-expressions`

essence, only two events are possible at that state, *tock* for time or *tick* for termination.

$Skipu$ 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 in external choice. That is $tock \notin (initials(P) \cup initials(Q))$. 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 follows 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 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*.

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

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

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

Well-formedness The expression d is restricted to 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, the processes P and Q are not allowed to begin 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 like the second process Q. If the second process Q did not perform an event, the entire process behaves as the first process P up to its completion.

Well-formedness There is a 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, because the time event *tock* causes non-determinism between the interrupt and the process of time.

$P\ \Theta_{\{Event\}}Q$ The operator exception describes a process that begins until the right-hand-side process performs an exceptional event to start the left-hand-side process. So, the process P begins until it performs an event from the exceptional events *Event* that triggers the process Q.

$P \setminus A$ Hiding specifies the behaviour of a process P which hides all the events in set A. The hidden events A becomes a 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 Here, the restriction is that the event *tock* should not 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 to recording the progress of time.

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

2.2. Strategy for the Translation Technique

The translation technique produces a list of small TAs [3], 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 [43]. The small TA provides enough flexibility for composing TA in various ways that capture the behaviour of the original *tock-CSP* process.

The connections between the small TAs 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

[3] In this work, we use TA that has few states and transitions connected together into a network of UPPAAL models

mark) to link two TAs, in such a way that the first TA (with `a!`) communicates with the second TA that has the corresponding co-action (`a?`), in the form of synchronous communication.

Definition 2.2. 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 TAs 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 participates 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 other processes that are composed with an 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 actions: `finish` for terminating action, `ext` for an external choice action, `intrp` for an interrupting action, and `excp` for an exception action. Similarly, the name of a synchronising action is in the form `eventName___sync`, 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, as 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 environment 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.

Definition 2.3. 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.

Example 2.1. A simple example that illustrates a translation of an Automatic Door System (ADS1), which opens a door, and then after at least one-time unit [4], the system closes the door. A *tock-CSP* process for modelling a simple version of ADS is:

```
ADS1 = open -> tock -> close -> SKIP
```

Translation of the process ADS1 produces the following list of TA in Figures 2 – 6.

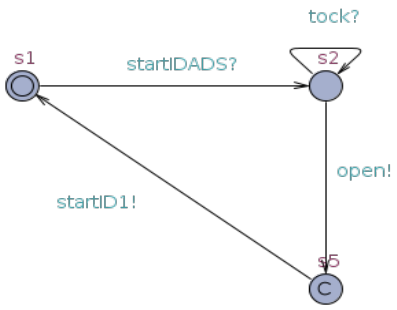


Figure 2: TA1 for the translation of the event `open` in the process ADS1.

[4]The models of *tock-CSP* never blocks time, as such each event `tock` models at least one or more unit time. In translating the event `tock`, we use a recursive transition to capture the progress of time for at least one time unit.

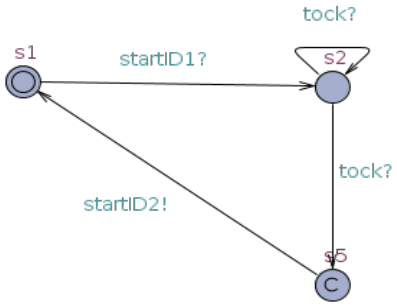


Figure 3: TA2 for the translation of the event `tock` in the process ADS1.

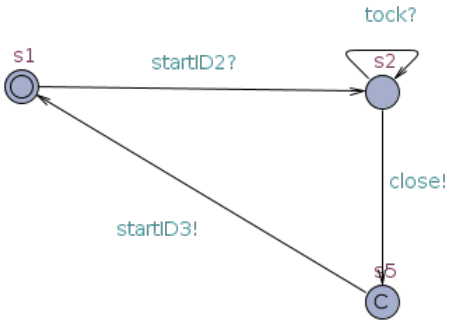


Figure 4: TA3 for the translation of the event `close` in the process ADS1.

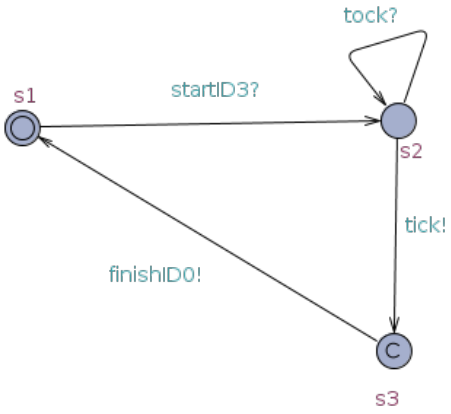


Figure 5: TA4 for the translation of the termination event `tick` in the process ADS1.

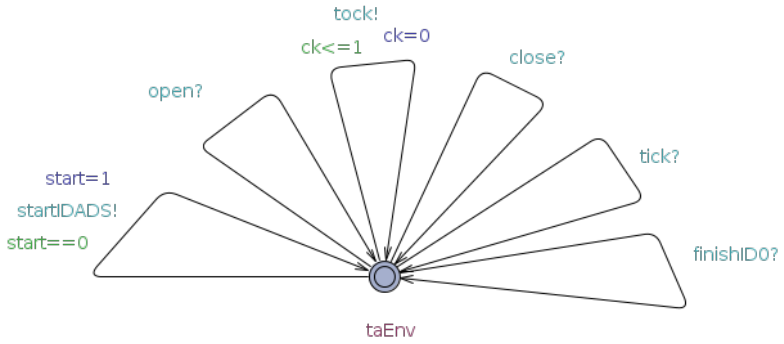


Figure 6: TAenv is an explicit environment of the UPPAAL model ADS1

In Example 2.1, we illustrate a translation of *tock-CSP* into a network of TA in Figure 2 – 6. The *tock-CSP* process ADS01 models the behaviour of an automatic door that opens a door and then at least after one unit time the system closes the door. Later on, we will extend the example to include synchronisation.

Translating Multi-synchronisation

In translating multi-synchronisation events, we adopt a centralised approach developed in [44] and implemented using Java in [45]. The approach describes using a separate centralised controller for controlling multi-synchronisation events. Here, we use UPPAAL broadcast channel to communicate synchronisation between the synchronisation TA and the TAs that participate in the synchronisation. In the following Example 2.2 we illustrate the strategy of translating synchronisation. Then, Definition 2.4 provides a definition of the synchronisation TA.

Definition 2.4. Synchronisation TA

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

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

ticipating TAs become ready, which enables the corresponding guard for broadcasting the multi-synchronisation action.

Example 2.2. An extended version of the Automatic Door System (ADS2) opens a door, and after at least one time unit, closes the door in synchronisation with a lighting controller, which turns off the light. In *tock-CSP*, its description is as follows.

```
1        ADS2 = Controller [|{close}|] Lighting
2
3  Controller = open -> tock -> close -> Controller
4
5    Lighting = close -> offLight -> Lighting
```

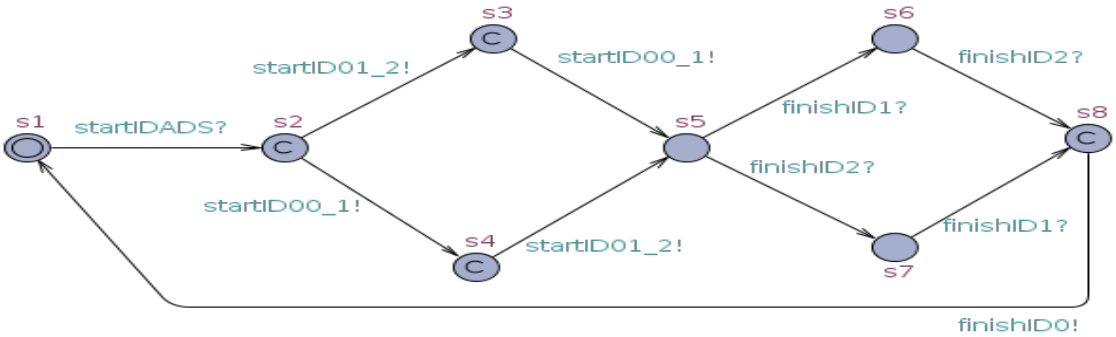
For example, a translation of the process ADS, from Example 2.2, produces the network of small TAs in Figure 7. Details of the translated TA are as follows. Starting from the top-left corner, the first TA captures concurrency by starting the two concurrent automata corresponding to the processes `Controller` and `Lighting` in two possible orders, either `Controller` then `Lighting` or vice versa, depending on the choice of the operating environment. Afterwards, it also waits on state `s5` for their termination actions in the two possible orders, either `Controller` then `Lighting` or vice versa, depending on the process that terminates first. Then, after the termination of the second process the whole system terminates with the action `finishID0`, if both process terminate.

The second (TA02), third (TA03) and fourth (TA04) TAs capture the translation of the process `Controller`. TA02 captures the occurrence of the event `open`. TA03 captures the occurrence of the event `tock?` (with a question mark) for synchronising with the environment in recording the progress of time. TA04 captures the occurrence of the event `close`, which synchronises with the synchronising controller, the fifth TA (TA05).

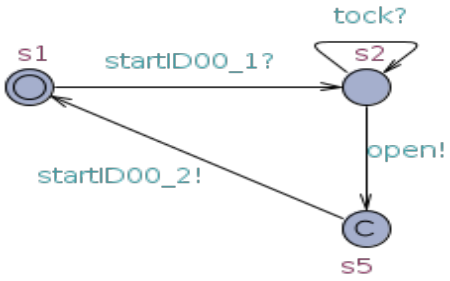
The sixth (TA06) and seventh (TA07) capture the translation of the process `Lighting`. TA06 captures the translation of `close`, which also needs to synchronise with the synchronisation controller (TA05). TA07 captures the event `offLight`. Finally, the last TA (TA08) is an environment TA that has co-actions for all the translated events. Also, the environment TA serves the purpose of 'closing' the overall system as required for the model checker. In the environment TA, we use the variable *start* to construct a guard *start* == 0 that blocks the environment from restarting the system. This concludes the description of the list of TA for ADS.

Translating Interrupt

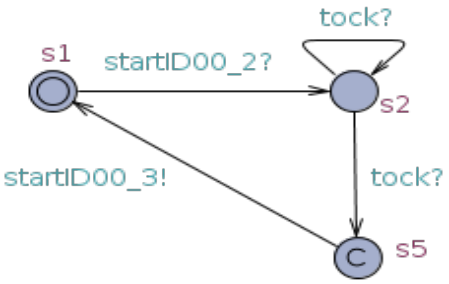
In *tock-CSP*, a process can be interrupted by another process when the two processes are composed with an interrupt operator (`/\`). This is due to the compositional structure of *tock-CSP*. However, in the case of TA, an explicit transition is needed for expressing an interrupt, which enables a TA to interrupt another one. So in this translation work, we provide an additional transition for capturing an interrupt event using an interrupt action, as defined in the coordination actions (Definition 2.2).



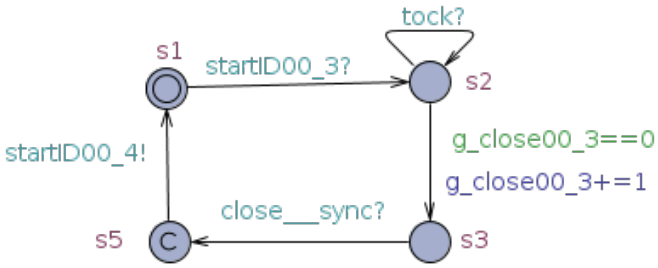
(a) TA01



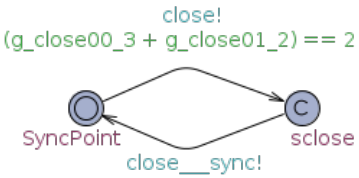
(b) TA02



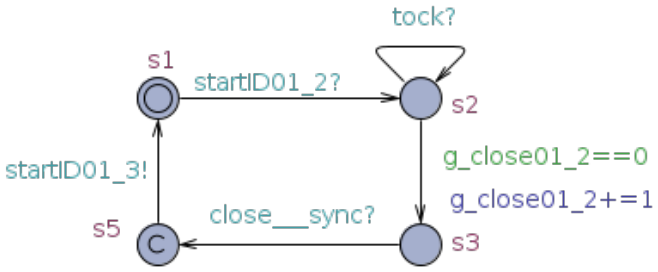
(c) TA03



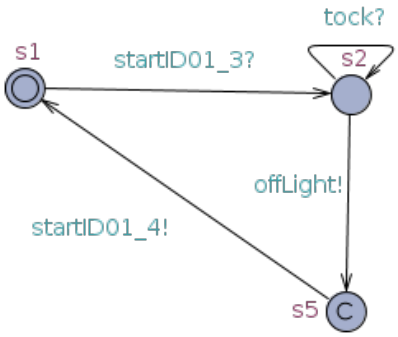
(d) TA04



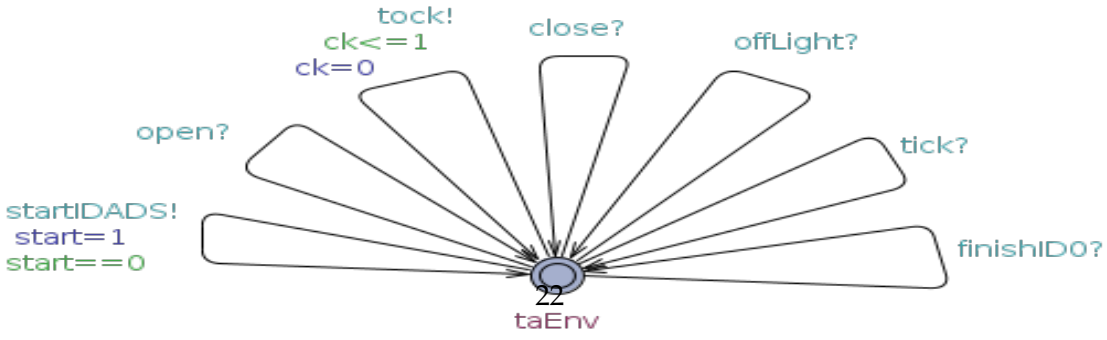
(e) TA05



(f) TA06

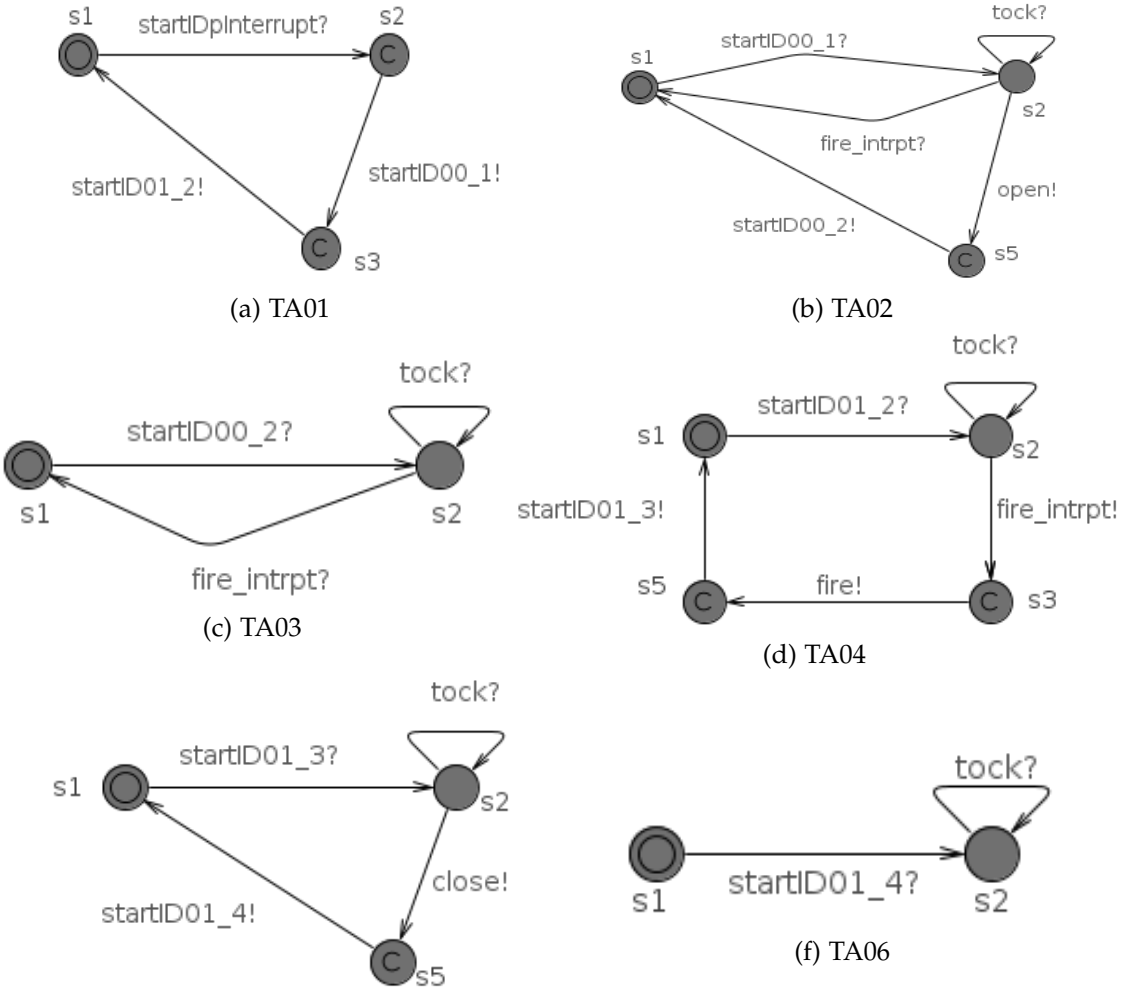


(g) TA07



(h) TA08

Figure 7: A list of networked TA for the translation of the process ADS2.



(a) TA01

(b) TA02

(c) TA03

(d) TA04

(e) TA05

(f) TA06

Figure 8: A list of TA for the translated behaviour of the process `Pi`.

For instance, given a process `Pi = 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.

An example of translating interrupt is provided in Figure 8, which illustrates a translation of the process `Pi = (open->STOP)/\(fire->close->STOP)`. For the process `Pi`, the RHS process `(fire -> close -> STOP)` can interrupt the behaviour of `(open->SKIP)` at any stable state. In the translated behaviour of the LHS process, we provide additional interrupting actions (`fire_intrpt`) that enable the translated behaviour of the RHS process to interrupt the LHS process. The interrupting actions are provided only for the initials of the RHS process (`fire`).

In Figure 8, starting from the top-left, TA01 is a translation of the operator interrupt. The second and third, TA03 and TA04 capture the translation of the LHS process `open->STOP`. TA04, TA05 and TA06 are translation of the RHS process `fire->close->STOP`. The environment TA is also omitted because it is similar to the last TA in Figure 7.

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

The third TA03 synchronises on the flow action `startID00_2` and moves to location `s2`, where it either performs the action `tock?` to record the progress of time or is interrupted with the co-action `fire_intrpt?`, and returns to its initial location `s1`. This completes the behaviour of the process `open->STOP`.

The fourth TA04 is a translation of the event `fire`. The TA begins with synchronising on the flow action `startID01_2`, which progresses by interrupting the LHS process using the interrupting flow action `fire_intrpt`, then `fire`, and proceeds to `startID01_3` for starting TA05. which synchronises on the flow action and moves to location `s2`, where it either performs the action `tock?` for the progress of time and remains in the same location or performs the action `close`, and proceeds to location `s5` then performs the flow action `startID01_4` for starting TA06, which captures the translation of the process `STOP` (deadlock).

Translating External Choice

Similarly, in translating external choice, we provide additional transitions that enable the behaviour of the chosen process to block the behaviour of the other processes. Initially, the translated behaviour makes 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 provided additional transitions of translating external choice, as defined in Definition 2.2.

For instance, consider a process `P = P1[]P2` that composes `P1` and `P2` with operator external choice such that the translated behaviour of the process `P` is denoted by `Tp` (a list of TAs for the translation of the process `P`). In similar manner, `Tp1` and `Tp2` are lists of TA for the translation of the processes `P1` and `P2`, respectively. Then, the first TA in the list `Tp1` has additional transitions for the initials of `P2` such that choosing the behaviour `Tp2` block the alternative behaviour `Tp1`. Similarly, the first TA in the list of TA `Tp2` has additional transitions for the initials of `P1` such that choosing the behaviour `Tp1` blocks the behaviour `Tp2`. Additional examples will follow later in this chapter and Appendix C.

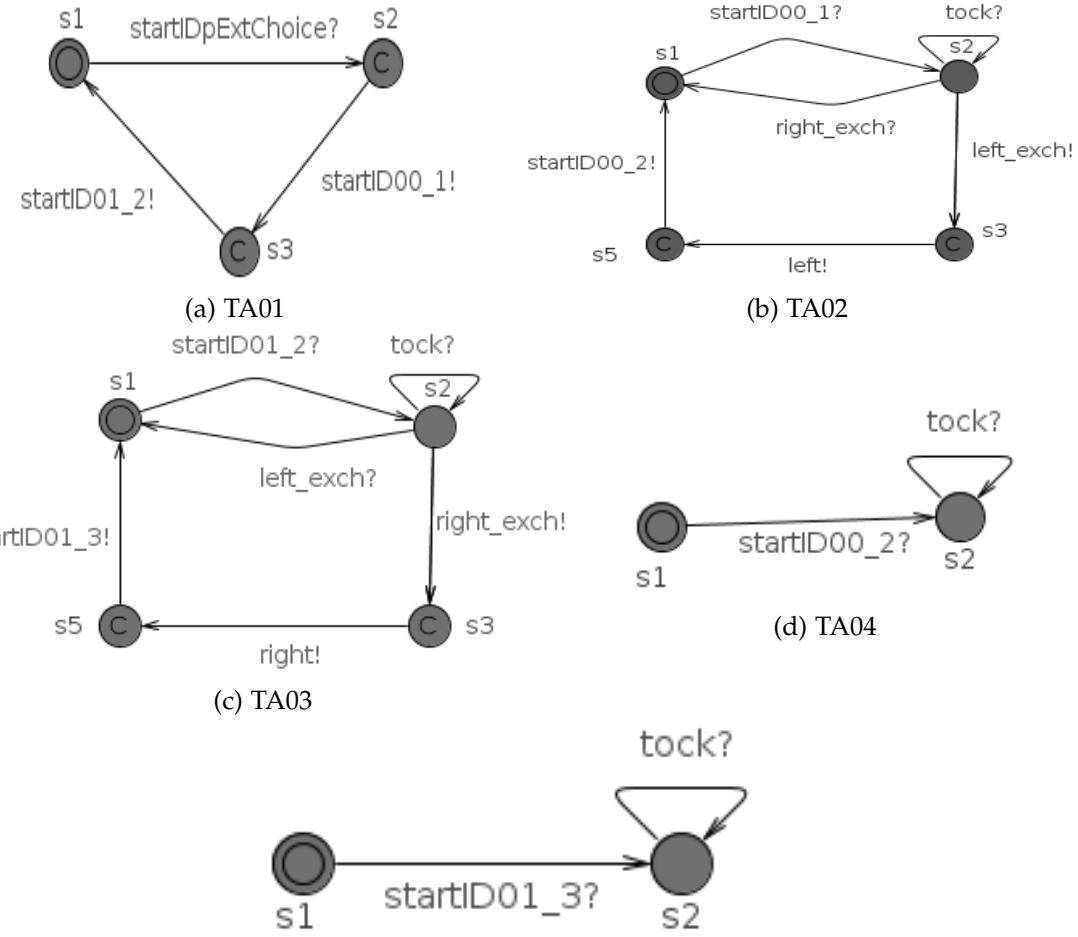


Figure 9: A list of TA for the translated behaviour of the process Pe.

An example of translating external choice is provided in Figure 9 for the process `Pe = (left->STOP)[](right->STOP)`, which composes the two processes `(left->STOP)` and `(right->STOP)` with the operator of external choice. In Figure 9, starting from the top-left, the first TA01 is a translation of the operator external choice. TA02 and TA04 are translations of the LHS process `left->SKIP`. Then, TA03 and TA05 capture the translation of the RHS process `(right->SKIP)`. From the beginning, TA01 has three transitions, each labelled with a flow action. The TA begins with the first flow action `startIDpExtChoice?`, then starts the two TAs, for the translation of both the left and right process, using the flow actions `startID00_1!` and `startId01_2!`, respectively.

Second, TA02 is a translation the event `left`. Initially, the TA synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has 3 possible transitions labelled with the actions: `left_exch?`, `right_exch!` and `tock?`. With

the co-action `tock?`, the TA records the progress of time and remains on the same location `s2`. With the co-action `right_exch?`, the TA performs an external choice co-action for blocking the TA of the LHS process when the environment chooses the right process, and the TA returns to its initial location `s1`. Lastly, the TA performs the action `left_exch!` when the environment chooses the LHS process, and the TA progress to location `s3` to perform the chosen action `left` that leads to location `s5` for performing the flow action `startID00_2`, which activates TA03 for the subsequent process `STOP`. This describes the behaviour of the LHS process `left->STOP`.

Fourth, TA04 is a translation of `right`, similar to the previous translation of `left` in the second TA. Fifth, TA05 is a translation of the process `STOP`. The omitted environment TA is similar to the last TA in Figure 7.

Translating Hiding and Renaming

Also, in *tock-CSP*, an event can be renamed or hidden from the environment. In handling renaming, the translation technique carries a list of renamed events. Before translating each event, the technique checks if the event is part of the renamed events, and then translates the event appropriately with the corresponding new name. In the same manner, if an event is part of the hidden events, the technique stores a list of hidden events, such that on translating a hidden event the technique uses a special name *itau* [5] in place of the hidden event.

Later in this section, we provide a function `transform()`, which composes the three functions (`transTA()`, `envTA()` and `syncTA()`) into a system. Details description of these three functions will come later in this section. The function `transform()` prepares the required arguments of the three functions and collects the output list of TA from these three functions, and then the function `transform()` assembles the list of TA into an XML file that is suitable for Uppaal toolbox.

This completes the description of the strategy we follow in developing the translation technique. The following section describes the details of the translation rules, and provides examples that demonstrate using each rule in translating a *tock-CSP* process.

2.3. Translation Rules

This section discusses the details of the translation rules. The section describes the translation rules in a functional style. Then, the section proceeds with presenting each translation rule separately. In addition, we provide examples that illustrate using each of the translation rules in translating a *tock-CSP* process.

Example 2.3. Here, we use a simple example of TA to illustrate the definition of TA provided in Figure 10, which is defined in Listing 1 using the syntax of Haskell for expressing the translation work. The TA has two locations A and B with an inner circle in the initial location A.

[5] A special event similar to the event tau in FDR to represent hidden event.

Figure 10: A sample output TA with two locations and one transition.

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

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

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

Line 3 highlights a definition location from Definition 2.5. 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 specifies 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 an empty label that specifies no constraint in the location, and specifies a type for the location to be of type None.

Line 8 is a comment that highlights a definition of a transition from Definition 2.6. Then, Line 9 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 11 using the definition of label from Definition 2.6 as an UPPAAL action with identifier ”start” that has direction Excl which specifies the action as a sender.

The above Example 2.3 illustrates a simple form of the output TA produced by the translation function `transTA`. However, in the translation rule, we will have a TA that has more than two locations and multiple transitions. The upcoming translation rules define the function `transTA`. Each rule defines a translation of one of the constructors

[6] A TA has its own local parameters for expressing its behaviour.

of the BNF previously presented in Section 2.1. In the next section, we discuss details of each of the translation rules together with an example for illustrating using the rule in translating a process.

Definition 2.5. Location

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

Definition 2.5 defines a location with a constructor that has 4 parameters, of types `ID, Name, Label and LocType`. First parameter of type `ID` is an identifier for the location. Second parameter of type `Name` is a tag for the location. Third parameter of type `Label` is a constraint label for the location, defined below in Definition 2.7. Last parameter of type `LocType` is a format of the location, which can be one of these three: `urgent`, `committed`, `None` (which means neither urgent nor committed, just normal location with no constraint).

Definition 2.6. Transition

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

Also, Definition 2.6 defines a data type for Transition, which has a constructor with 3 parameters, of type `Source`, `Target` and `[Label]`. First parameter of type `Source`, is a starting location for the transition. The second parameter of type `Target` is a destination location for the transition. The third parameter of type `[Label]` is a list of labels for the transition.

Definition 2.7. Label

```
1 data Label = EmptyLabel
2            | Invariant     Expression
3            | Guard         Expression
4            | Update       [Expression]
5            | Sync          Identifier   Direction
```

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

Definition 2.8. Function `transTA`

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

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

2.3.1. 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 2.1. 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         intrp = transIntrpt intrptsInits loc1 loc2
15         trans = [tran1, tran2] ++ intrp
16
17         lab1 = Sync (VariableID
18                        (startEvent processName (bid ++ sid)) [])
19                     Ques
20         lab2 = Sync (VariableID "tock" []) Ques
21
22         -- Get initial events for possible interrupting process
23         (_, _, _, _, _, intrptsInits, _, _) = usedNames
```

Rule 2.1 expresses the translation of the construct STOP, which produces an output TA depicted in Figure 11. The figure illustrates the structure of the output TA that has two locations and two transitions as defined in Lines 7–9 and Lines 11–13, respectively.

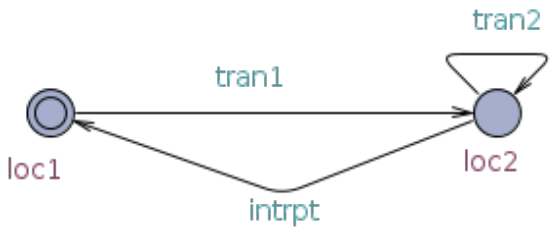


Figure 11: A structure of TA for the translation of STOP.

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

Also, in the output tuples, the first element (non-empty element) is a definition of the output TA for the translation of the constant process STOP, which has six parameters. First, `idTA` is an identifier for the TA, which is defined subsequently in Line 4, as concatenation of the keyword `"taSTOP__"` with the 2nd and 3rd arguments of the function `transTA`, that is `bid` and `sid` respectively. Additionally, still in Line 2, in the definition of the output TA, the 2nd, 3rd and 5th parameters are empty for the output TA. While the 4th parameter `locs` is a list of locations for the output TA defined in Lines 7–9. The 6th parameter `(Init loc1)` specifies `loc1` as the initial location of the output TA. Lastly, `trans` describes a list of transitions that connect the two locations as defined in Line 12–13. Lines 14 defines interrupt transitions in the case of translating a process that is composed with an operator interrupt.

Finally, Lines 17 – 20 define the labels of the two transitions in the output TA. The first label `lab1` defines a label for the first transition as a first flow action, which we generate its name using the function `startEvent`, as defined in the following Definition 2.9. The second label `Lab2` for the second transition is defined as an UPPAAL action "tock" with `Ques`, which indicates a receiving action.

Definition 2.9. Function `startEvent`

```
1 startEvent :: String      -> String -> Int      -> String
2 startEvent    processName    bid       sid      =
3                              if    notNull      processName
4                              then "startID" ++  processName
5                              else "startID" ++  bid ++ show sid
```

The function `startEvent` generates the name of the first flow action. If the input process has an identifier, we use the identifier in the translated TA as the name of the first flow action, else the function generates a new name for the first flow action. 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`.

Declaration of the function `transIntrpt`

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

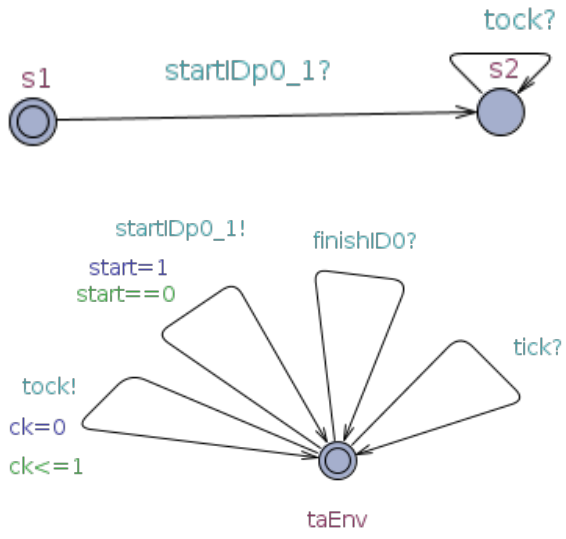
Line 14 defines interrupt transitions using the initials of an interrupting process defined in the function `transIntrpt`, as highlighted in the translation strategy in Section 2.2. The function `transIntrpt` has three parameters. The types of the parameters are the list of events (initials of an interrupting process) and two locations that connect the transition for interrupt. The first argument `intrpts` is the initials of an interrupting process and generates a transition for each of the initials.

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

The behaviour of the output TA begins with the first flow action (line 17), which is constructed using a function `startEvent`, previously defined in Definition 2.9. After that, the TA performs the action `tock` (line 18), repeatedly, which allows time to progress. An illustration of using this translation rule is provided in the following Example 2.4.

Example 2.4. An example of translating a process STOP produces a list of TA that contains two TAs, as illustrated below.

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




```
]
```

Example 2.4 demonstrates a translation of the process STOP using Rule 2.1, which produces a list of translated TA that contains two TA, a small TA and its corresponding environment TA. The behaviour of the output TA begins with the environment TA that performs its first flow action `startIDp0_1!` with the cooperation of the small

TA using its corresponding co-action `startIDp0_1?`. 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`.

2.3.2. Translation of Stopu (Timelock)

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

Rule 2.2 expresses the translation of the constant process `Stopu` that produces an output TA that is depicted in the following Figure 12, which is annotated with the names used in the translation rule. The figure illustrates the structure of the output TA, which has two locations and one transition as defined in Lines 7–9 and Line 12 respectively.

This description of Rule 2.2 resembles the previous description of Rule 2.1 (translation of STOP), except that the output TA of this rule does not perform the event `tock` that allows time to progress in the previous rule. The structure of this output TA has two locations, `loc1` and `loc2` as defined 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 synchronising on the first coordinating start event and then deadlock immediately. This is illustrated in the following example for translating the constant process `Stopu`.

Rule 2.2. 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
```

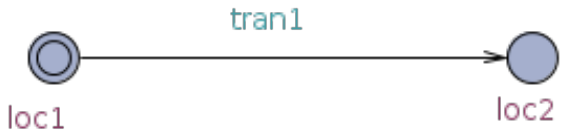
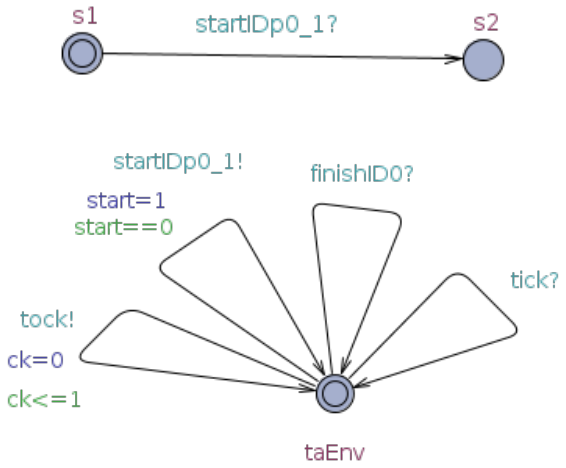


Figure 12: A structure of TA for the translation of `Stopu`

Example 2.5. An example for translating an urgent process Stopu.

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




```
8         ]
```

The above Example 2.7 illustrates a translation of the constant process `Stopu` according to Rule 2.2. Also, this example resembles the previous Example 2.4, 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.

2.3.3. Translation of SKIP

This section describes the translation of process SKIP. The section begins with presenting a rule for translating the process SKIP. Then, follows with an example that

illustrates using the rule in translating a process. However, the behaviour of SKIP can be interrupted just before termination, so we have to consider a possible interrupt in translating interrupt.

Rule 2.3 describes a translation of process `SKIP` into a single output TA, which is depicted in the following Figure 13. The figure is annotated with the names used in the translation rule. The structure of the output TA has 3 locations: `loc1`, `loc2` and `loc3` (define in Lines 6 – 9) and 4 transitions `tran1`, `tran2` and `tran3` (define in Lines 11 – 16).

The behaviour of the TA begins on `tran1` for a flow action that is defined using the function `startEvent` (Definition 2.9). Then, the TA follows one of the 3 transitions: `tran2, tran3 or intrp`. On transition `tran2`, the output TA performs the event `tock` to record the progress of time, and remains in the same location `loc2`. On transition `tran3`, the output TA performs the event `tick` and then immediately follows the subsequent transition `tran4` to perform the termination event `finishID0!`. Finally, on transition `intrp`, the TA is interrupted by another process.

Line 15 defines an interrupt transition, which is provided for the case of translating a process that involves interrupt. Details of translating interrupt will be provided in Section 2.3.13. Here, we highlight a declaration of the function `transIntrpt` below, due to its first appearance in this translation rule.

Rule 2.3. 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           loc1 = Location "id1" "s1" EmptyLabel None
6           loc2 = Location "id2" "s2" EmptyLabel None
7           loc3 = Location "id3" "s3" EmptyLabel CommittedLoc
8           locs = [loc1, loc2, loc3]
9
10          tran1 = Transition loc1 loc2 [lab1]  []
11          tran2 = Transition loc2 loc2 [lab2]  []
12          tran3 = Transition loc2 loc3 [lab3]  []
13          tran4 = Transition loc3 loc1 [lab4]  []
14          intrp = transIntrpt intrptsInits loc1 loc2
15          trans = [tran1, tran2, tran3, tran4] ++ intrpt
16
17          lab1 = Sync (VariableID (startEvent processName
18                       (bid ++ sid)) []) Ques
19          lab2 = Sync (VariableID "tock" []) Ques
20          lab3 = Sync (VariableID "tick" []) Excl
21          lab4 = Sync (VariableID  finishLab []) Excl
22
23          finishLab = ("finishID" ++ show fid)
24
25          -- Gets initial events of an interrupting process
26          (_, _, _, _, _, intrptsInits, _, _) = usedNames
```

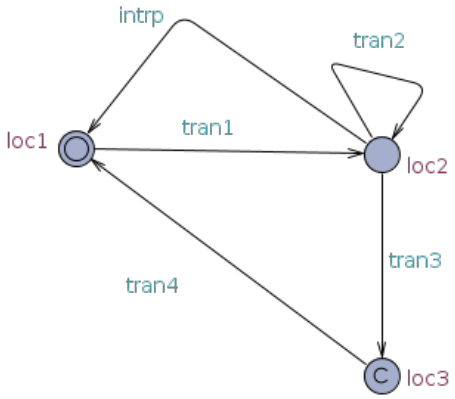


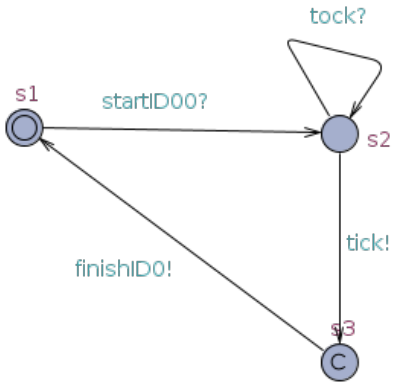
Figure 13: A structure of a TA for the translation of the process SKIP.

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

using the names. In this rule (Line 27), we start using the name `intrpts` from the names `usedNames`. We used the name `intrpts` to collect the initials of interrupting processes for constructing interrupting transitions. This completes the description of Rule 2.3. The following Example 2.6 illustrate using the rule for translating constant process `SKIP`.

Example 2.6. An example for translating a constant process `SKIP`.

```
1 transTA SKIP "" "0" 0 0
2          ([], [], [], [], [], [], [], ([],[])) =
3          [
```




```
4          ]
```

Example 2.6 illustrates using Rule 2.6 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 an empty name to illustrate the translation of a process that has an empty name.

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

In this example, there is no interrupting process, so the function `transIntrpt` produces an empty list for the interrupting transitions. Section 2.3.13 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`.

2.3.4. Translation of Skipu (Urgent termination)

This section describes the translation of another constant process `Skipu`, which specifies an urgent termination that does not allow 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 2.4. Translation of Skipu

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

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

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

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

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

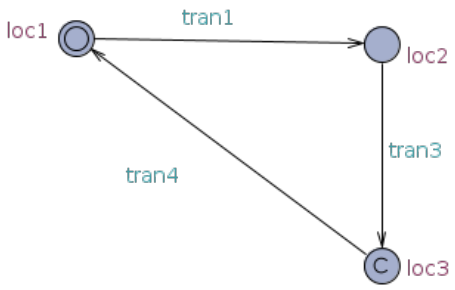


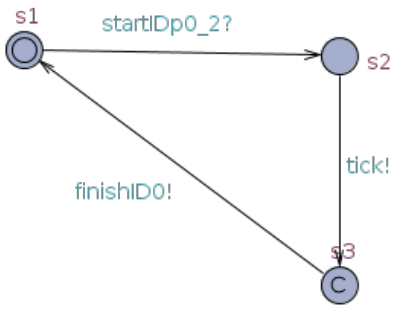
Figure 14: A structure of TA for the translation of urgent termination.

Rule 2.4 resembles the previous Rule 2.3, except that on the location `s2` the output

TA does not perform the event `tock`. This means that the TA terminates immediately, as illustrated in Figure 14. The following example demonstrates using the rule in translating a process.

Example 2.7. An example for translating a process for an immediate termination.

```
1 transTA Skipu "" "0" 0 0
2          ([], [], [], [], [], [], [], ([],[]))
3          = [
```




```
4          ]
```

Similarly, Example 2.7 resembles Example 2.6, except that the output TA terminates immediately. Initially, the TA synchronises on the coordinating start action `startIDp0_2`, then the TA performes the event `tick` and proceeds immediately to perform a termination action `finishID0`, which indicates successful termination. There is no interrupting process in this example, so the interrupting transitions are empty. This completes the description of Example 2.7, which illustrates a translation of the constant process `Skipu` for urgent termination.

2.3.5. 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 the operator prefix happens to be the largest translation rule because in translating each event, we need to check if the event is part of events that are hidden, renamed, synchronisation, initial of external choice or initial of another interrupting process. In each of these cases, the TA has different behaviour that has to be translated according to the specification of the process.

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

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

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

Thus, first the translation rule includes a function for checking both hidden and renamed events. And then we formulate eight cases for capturing the possible behaviour of an event that is used as part of synchronisation, external choice or interrupt. Each case captures a possible behaviour of an event for a process that is part of synchronisation, external choice or interrupt.

Rule 2.5. Translation of Prefix (1 of 5))

```
1  transTA (Prefix e1 p) procName bid sid fid usedNames =
2          (([(TA idTA [] [] locs1 [] (Init loc1) trans1)] ++ ta1),
3                  sync1, syncMapUpdate)
4    where
5      idTA = "taPrefix" ++ bid ++ show sid
6      (syncs, syncMaps, hides, renames, exChs, intrpts,
          initIntrpts,
7              excps) = usedNames
8
9      -- Checking hiding or renaming
10     e = checkHidingAndRenaming e1 hides renames
11
12     {- 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 -}
13     (locs1, trans1)
14         |((not synch)&&(not exChoice)&&(not interupt)) = case1
15         |((not synch)&&(not exChoice)&&(    interupt)) = case2
16         |((not synch)&&(    exChoice)&&(not interupt)) = case3
17         |((not synch)&&(    exChoice)&&(    interupt)) = case4
18         |((    synch)&&(not exChoice)&&(not interupt)) = case5
19         |((    synch)&&(not exChoice)&&(    interupt)) = case6
20         |((    synch)&&(    exChoice)&&(not interupt)) = case7
21         |((    synch)&&(    exChoice)&&(    interupt)) = case8
```

Translation of Prefix (2 of 5)

```
23      case1 = ([loc1, loc2, loc5],
24               [t12, t25,  t51] ++ addTran ++ transIntrpt')
25      case2 = ([loc1, loc2, loc3c, loc5],
26               if not $ null intrpts
27               then [t12G, t23ci, t3c5, t51] ++ addTran
28               else [t12,  t23ci, t23cgi, t3c5, t51] ++ addTran
29                    ++ transIntrpt')
30      {- if a process can interrupt and also be interrupted, then
           it can only be interrupted after initiating its interrupt
         -}
31      case3 = ([loc1, loc2,  loc3c,  loc5],
32               [t12,  t23c,  t3c5,    t51] ++ t23e ++ addTran
33               ++ transIntrpt')
34      case4 = ([loc1, loc2,  loc3c,  loc4c,   loc5],
35               [t12G, t23c,  t3c4ci, t3c4cgi, t4c5e,  t51] ++
36               addTran ++ transIntrpt')
37      case5 = ([loc1, loc2, loc3, loc5],
38               [t12,  t23,  t35,  t51] ++ addTran ++ transIntrpt')
39      case6 = ([loc1, loc2,  loc3,   loc4c,   loc5],
40               [t12G, t23,   t34c,   t4c5i,   t4c5gi, t51]  ++
41               addTran ++ transIntrpt')
42      case7 = ([loc1, loc2,  loc3,   loc4,    loc5],
43               [t12,  t23ech, t34,   t45,     t51] ++
44               addTran ++ transIntrpt')
45      case8 = ([loc1, loc2,  loc3,   loc4,    loc5,   loc6],
46               [t12G, t23,   t34c,   t4c6,    t65, t65gi, t51] ++
47               addTran ++ transIntrpt')
48
49      --     =  Location  ID    Name  Label      LocType
50      loc1   =  Location "id1"  "s1"  EmptyLabel None
51      loc2   =  Location "id2"  "s2"  EmptyLabel None
52      loc2c  =  Location "id2"  "s2"  EmptyLabel CommittedLoc
53      loc3   =  Location "id3"  "s3"  EmptyLabel None
54      loc3c  =  Location "id3"  "s3"  EmptyLabel CommittedLoc
55      loc4   =  Location "id4"  "s4"  EmptyLabel None
56      loc4c  =  Location "id4"  "s4"  EmptyLabel CommittedLoc
57      loc5   =  Location "id5"  "s5"  EmptyLabel CommittedLoc
58      loc6   =  Location "id6"  "s6"  EmptyLabel CommittedLoc
59
60      transIntrpt' = (transIntrpt intrpts loc1 loc2)
61
62      -- Additional transitions for tock, external choice
63      addTran | ((not  $ elem e syncs)&&(null exChs))      = [t22]
64              | ((elem e syncs)       &&(null exChs))      = [t22]
65              | ((not  $ elem e syncs)&&(not $ null exChs))= [t22]
                    ++ t21
66              | otherwise = [t22, t33, t44] ++ t21
```

Translation of Prefix (3 of 5)

```
68     t23ci   = Transition  loc2     loc3c   l23ci            []
69     t23cgi  = Transition  loc2     loc3c   altIntrpt        []
70     l23ci   = [(Sync (VariableID   ((show e) ++ "_intrpt")  [])
          Excl), (Update  [(AssgExp  (ExpID ((show e) ++ "
          _intrpt_guard")) ASSIGNMENT TrueExp )])]
71
72     {- 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 -}
73     altIntrpt = [(Guard
74                   (ExpID (intercalate "␣||␣"  [l ++
75                  "_intrpt_guard"| (ID l) <- initIntrpts])))]
76     -- reset the guards in case of recursive process
77     resetG  = [Update  [(AssgExp  (ExpID (l ++ "_intrpt_guard"))
78                     ASSIGNMENT FalseExp)| (ID l) <- initIntrpts
                        ]]
79     t3c5    = Transition  loc3c    loc5    lab4e            []
80     t3c4ci  = Transition  loc3c    loc4c   l23ci            []
81     t3c4cgi = Transition  loc3c    loc4c   altIntrpt        []
82     t4c5    = Transition  loc4c    loc5   (lab2i ++ lab2d)  []
83     t4c5e   = Transition  loc4c    loc5    lab4e            []
84     t34c    = Transition  loc3     loc4c   lab4             []
85     t3c4c   = Transition  loc3c    loc4c   lab4e            []
86     t4c5i   = Transition  loc4c    loc5    l23ci            []
87     t4c5gi  = Transition  loc4c    loc5    altIntrpt        []
88     t4c6    = Transition  loc4c    loc6   (lab2i ++ lab2d)  []
89     t65     = Transition  loc6     loc5    l23ci            []
90     t65gi   = Transition  loc6     loc5    altIntrpt        []
91     t12     = Transition  loc1     loc2    lab1             []
92     t12G    = Transition  loc1     loc2   (lab1 ++ resetG)  []
93     t23     = Transition  loc2     loc3    lab2i            []
94     t2c3    = Transition  loc2c    loc3    lab2i            []
95     t23c    = Transition  loc2     loc3c  (lab2i ++ lab2d)  []
96     t23ech  = Transition  loc2     loc3   (lab2i ++ lab2d)  []
97     t25     = if elem e hides
98               then Transition    loc2  loc5
99                 ([(Sync (VariableID "itau" []) Excl)]) []
100              else Transition    loc2  loc5   lab2i    []
101    t25r    =  Transition  loc2  loc5  ([(Sync (VariableID
102                   (show new_e) []) Excl)] ++ labpath) []
103    new_e   =  head [newname | (oldname, newname) <- renames,
          oldname == e]
104    t51     =  Transition  loc5  loc1  [lab3]          []
105    t33     =  Transition  loc3  loc3  [labTock]       []
106    t44     =  Transition  loc4  loc4  [labTock]       []
107    t35     =  Transition  loc3  loc5   lab4           []
108    t22     =  Transition  loc2  loc2  [labTock]       []
```

Translation of Prefix (4 of 5)

```
109     t21     = [(Transition loc2  loc1  [(Sync  (VariableID
110                  ((show ch) ++ "_exch")  []) Ques)] [])|ch <-
                        exChs']
111     t23e    = [(Transition loc2  loc3  [(Guard (ExpID ((show ch)
            ++ "_exch_ready")) )] [])|ch <- exChs']
112     t34     =  Transition  loc3  loc4   lab6  []
113     t45     =  Transition  loc4  loc5   lab4  []
114
115     lab1    = [Sync (VariableID (startEvent procName bid sid)
            []) Ques]
116
117     lab2i   | (elem e syncs)&&(null exChs')  =    -- check sync
118                 [(Guard (BinaryExp (ExpID ("g_" ++
119                     (eTag e syncMaps' ""))) Equal (Val 0))),
120                 (Update [(AssgExp (ExpID ("g_" ++
121                     (eTag e syncMaps' ""))) AddAssg (Val 1))])]
122             | (not $ null exChs') =
123                     if (elem e hides)
124                     then [(Sync (VariableID "itau_exch"  [])
                            Excl)]
125                     else [(Sync (VariableID ((show e ) ++
126                       "_exch")     []) Excl)]
127             | otherwise    =  lab4e
128
129     labpath = [(Update  [(AssgExp (ExpID "dp") AddAssg (Val 1)),
130               (AssgExp  ( ExpID   ("ep_" ++ bid ++ "_" ++ show
                        sid)) ASSIGNMENT TrueExp )])]
131                 -- Attaching path variable transition
132     -- Checks for exception
133     lab3    =  if elem e (fst excps)
134                then Sync (VariableID ("startExcp" ++ (show fid )
                        ) []) Excl
135                else Sync (VariableID ("startID" ++ bid ++ "_" ++
136                          show (sid+1)) []) Excl
137     lab4    = [(Sync (VariableID ((show e) ++ "___sync") [])
            Ques)]
138     lab4e   | e == Tock    = [(Sync (VariableID (show e) [])
            Ques)] ++ labpath  -- Sync on tocks
139             | elem e hides = [(Sync (VariableID ("itau") [])
                    Excl)]     -- itau for hiding event
140             | otherwise    = [(Sync (VariableID (show e) [])
                    Excl)]  ++ labpath  -- Fire normal event
141     lab6    = [(Guard (BinaryExp (ExpID ("g_" ++ (eTag e
            syncMaps' ""))) Equal (Val 0))), (Update [(AssgExp (ExpID
             ("g_" ++ (eTag e syncMaps' ""))) AddAssg (Val 1))])]
```

Translation of Prefix (5 of 5)

```
143     lab2d   = [(Update [(AssgExp  (ExpID ((show ch) ++ "
           _exch_ready")) AddAssg  (Val 1)) | ch <- exChs'])]
144     gIntrpt = [(Guard (BinaryExp (ExpID "gIntrpt") Equal   (Val
           1)))]
145     uIntrpt = [(Update [(AssgExp (ExpID "gIntrpt") AddAssg (Val
           1))])]
146     labTock = Sync (VariableID  "tock" [])  Ques
147     synch    = elem e syncs
148     exChoice = null exChs
149     interupt = null initIntrpts
150
151     -- Update sync points
152     syncMaps' = if elem e syncs
153                 then [(e, (show e) ++ bid ++ "_" ++ show sid )]
154                 else [] -- syncMaps_
155
156     -- Combine the synchronisations together
157     syncMapUpdate = syncMaps' ++ syncMap1
158
159     -- Replace renamed event with the new name
160     exChs' =  if  ( null crs ) then exChs
161               else  (exChs \\ [es']) ++ [nn']
162
163     -- rename all events for blocking external choice
164     crs  = [(es, nn) | (es, nn) <- renames, ch <- exChs, ch ==
           es]
165     (es', nn') = head crs
166
167     {- Update used names and then remove external choice and
           interrupt if any, after the first event. -}
168     usedNames' = (syncs, syncMaps, hides, renames, [], intrpts,
           [], excps)
169
170     -- Finally recursive call for subsequent translation.
171     (ta1, sync1, syncMap1) = transTA p []  bid (sid+1) fid
           usedNames'
```

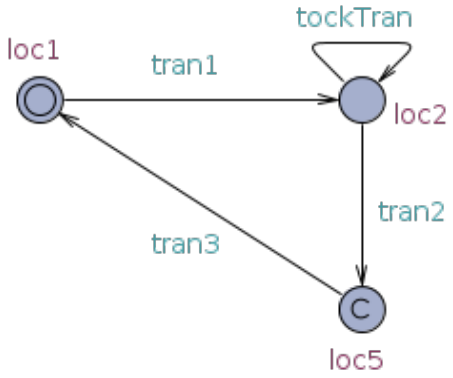


Figure 15: A structure of TA for the translation of Prefix.

As discussed in Section 2.1, the operator prefix is a binary operator that combines an event with a process, syntactically in the form of `event->Process`. The prefix event is translated according to one of the eight possible cases for a process that takes part in synchronisation, external choice and interrupt. Each case defines a different behaviour for the prefix event in the translation rule.

In Figure 15, we annotate the structure of the TA for translation of an event in case 1. The TA has 3 locations and four transitions, as defined in Lines 27–29 and Lines 31–37 respectively. The TA begins with transition `tran1` for performing flow action, and then on location `loc2` either the TA performs the action `tock` to record the progress of time and return to the same location, or perform the translated action on transition `tran2` that leads to location `loc3`. Then, on transition `tran2`, the TA performs another flow action to activate the subsequent TA. The remaining 7 cases follow a similar pattern.

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

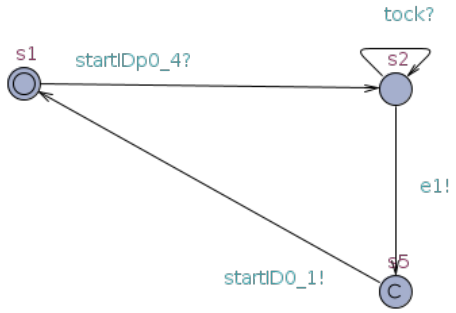
Cases 5 to 8 are cases that involve synchronisations. Case 5 defines a translation of an event that is part of synchronisation only, which means that multiple processes synchronise on performing the event. Case 6 is for the translation of an event that is part of both synchronisations and interrupts.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

For each of these 8 cases, Rule 2.5 defines a separate TA for translating the behaviour of a prefix event. Definition of all the locations and transitions of these eight possible TAs generates a long list of definitions that makes the size of the rule very large.

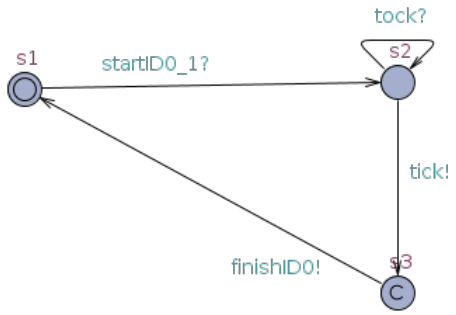
Here, we present a high-level definition of the rule for the main part, which omits the detailed description of locations and transitions of all the possible output TA for this translation rule. In Figure 15, we map the names with the structure of TA defined in case 1 of the translation rule. Example 2.8 illustrates using the translation Rule 2.5 in translating a process.

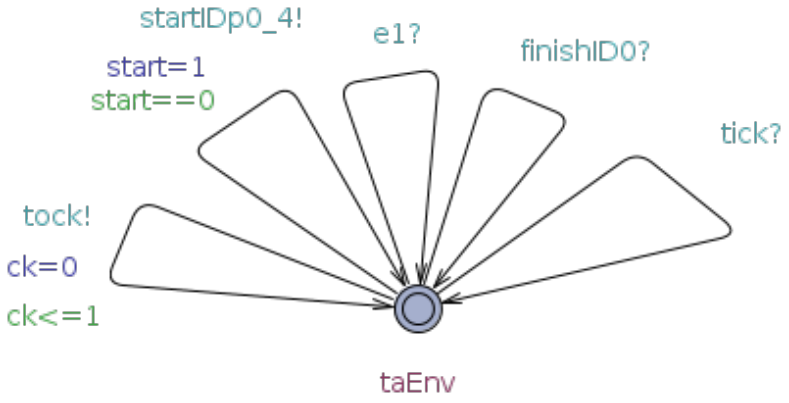
Example 2.8. An example to demonstrate using Rule 2.5 in translating a process `e1->SKIP`.

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




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





5]

Example 2.8 illustrates using Rule 2.5 in translating a process `e1->SKIP`, which is translated into a list containing three TA. The first Ta captures the translation of the event `e1` using Rule 2.5. The second TA captures the translation of the subsequent process `SKIP` using Rule 2.3. The last TA is an environment TA for the list of the translated TA.

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

2.3.6. Translation of WAIT n

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

Rule 2.6. 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
```

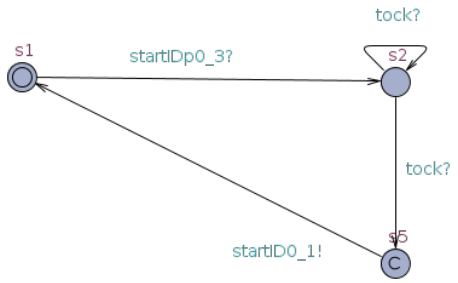
Rule 2.6 describes a translation of delay process `WAIT(n)`, which is translated in terms of two constructs: `Prefix` and `SKIP`, previously defined in Rule 2.3 and Rule 2.5, respectively. In the syntax of *tock-CSP*, this is expressed as:

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

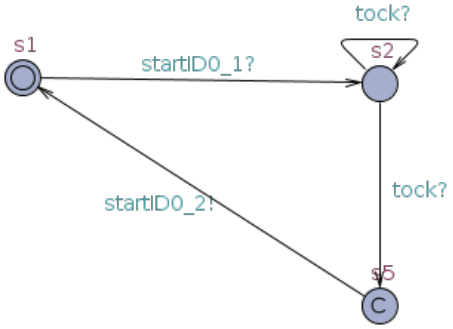
The process `WAIT(n)` is translated into a list of TA, which performs the event `tock` n times until the value `n` becomes `0` and the the TA behaves as `SKIP`. The base case is translated according to Rule 2.6. While the remaining cases are translated according to Rule 2.5. The following example illustrates using the rule in translating a process.

Example 2.9. An example for translating a delay process `WAIT(2)`, which expresses a delay of 2 units time. The process is translated as follows.

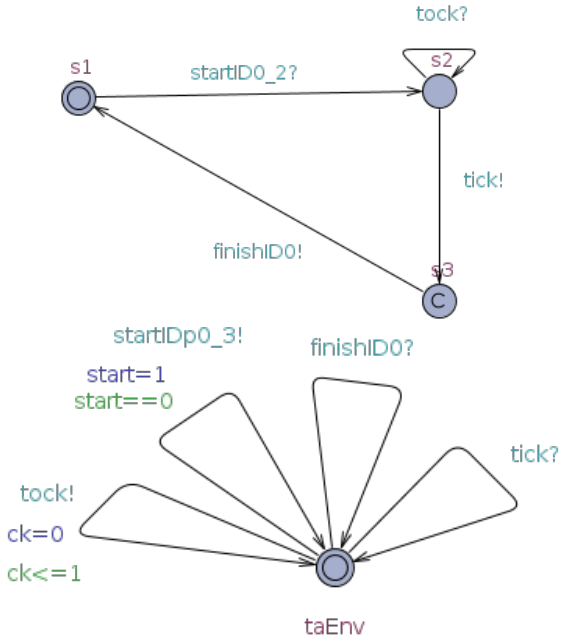
```
transTA (WAIT 2) "p0_3" "0" 0 0
        ([], [], [], [], [], [], [], ([],[])) =
        [
```




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




```
7           ] ++
8           transTA (WAIT 0) "" "0" 2 0
9                   ([], [], [], [], [], [], [], ([],[])) =
```




```
10          ]
```

Example 2.9 illustrates using Rule 2.6 in translating the process `WAIT(2)`. Initially, the example defines the function `transTA` for the construct `(WAIT n)` and its required arguments: process is `WAIT(2)`, process name is "p0_3", `branchID` is "0", `startID` is 0, `finishID` is 0 and `usedNames` is the remaining empty lists for the collection of names that are empty at the beginning. The translation produces a list of TA `TA0, TA1 and TA2` in the example.

Details behaviour of the output TA is as follows. First, `TA0` synchronises on the flow action `startIDp0_3`, which connects the environment with the first TA in the list of the translated TA. Then, on location `s2`, `TA0` performs the time event `tock` at least once and then performs the subsequent flow action `startID01`, which connects two

TA0 and TA1. In this case, TA1 is similar to the previous TA0 because both of them capture the translation of the event tock; TA1 performs the second action tock and then performs another flow action startID02, which connects TA1 and TA2. Lastly, TA2 synchronises on the flow action startID02 and then either TA2 performs the action tock and remains in the same location; or TA2 performs the action tick and then immediately proceeds to a terminating action finishID0, which indicates a successful termination of the translated process. These three TAs describe the translation of the process WAIT(2).

2.3.7. Translation of Waitu n (Strict delay)

This section describes the translation of process Waitu n, a strict delay of n time units. 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 2.7 describes a translation of strict delay. In Figure 16, we annotate the structure of the output TA with the names used in the translation rule. The structure of the TA has 2 locations and three transitions as defined in Lines 6–8 and Lines 10–14. Then, Line 16 extracts the used names for interrupt. And Lines 18–20 defines the labels of the transitions. Lines 22–27 defines the guards for controlling the deadlines. Finally, Lines 30–31 reset the deadline in case of a translating process that has recursive calls.

The behaviour of the output TA begins on transition tran1, where the TA synchronises on a flow action (defined in Line 18). Then, on Location loc2 (defined in Line 6), either TA follows transition tran2 or tran3. On transition tran2 (define in Line 10), the TA checks the delay guard dlguard (defined in Line 22), if it is true the TA performs the time event tock and update the delay timer with the expression dlupdate (Lines 23–24). Alternatively, if the guard dlguard is false, the second guard dlguard2 becomes true (Line 27), which enables the TA to perform the next flow action on transition tran3, as well as resetting the timer in the expression t_reset (Lines 30–31). This transition completes the behaviour of the translated TA. The following Example 2.10 illustrates using the rule in translating a process.

Rule 2.7. 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 the deadline time
    t_reset = [(Update [(AssgExp (ExpID "tdeadline"
                                 ASSIGNMENT (Val 0)) ] )  ]
```

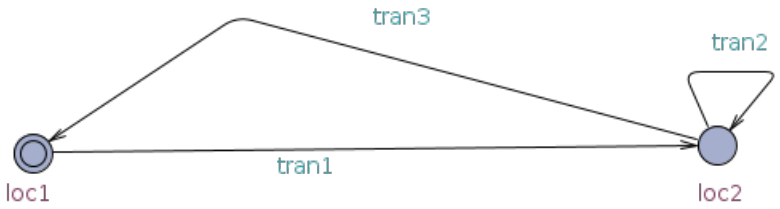
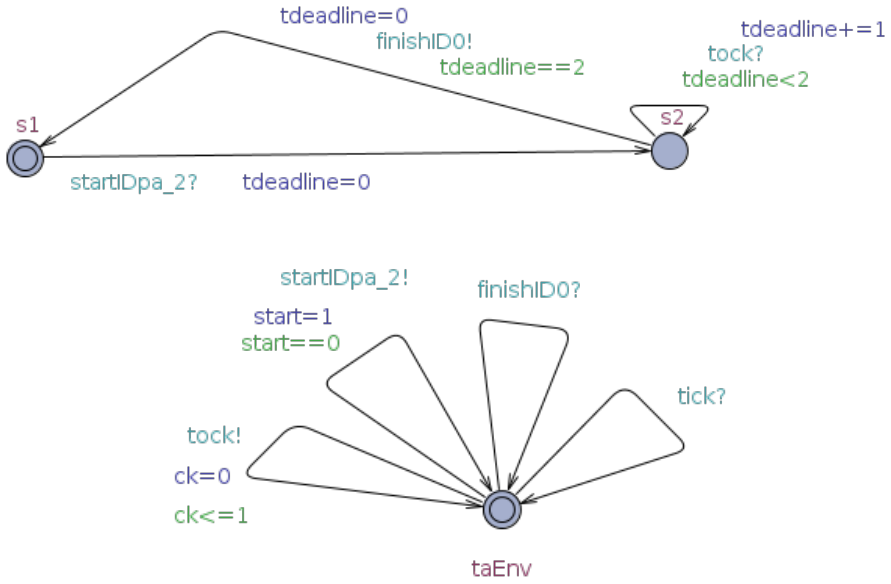


Figure 16: A structure of a TA for a translation of strict delay.

Example 2.10. An example for translating a process `Waitu(2)` a strict delay of 2 time units is illustrated in this example.

```
1 transTA (Waitu 2) "pa_2" "0" 0 0
2         ([], [], [], [], [], [], [], ([],[]))
3         ↪ [
```




```
]
```

Example 2.10 illustrates using Rule 2.7 in translating a process `Waitu 2`. The example translates the process `Waitu 2` into a list of TA that contains two TAs. In the beginning, the example applies the function `transTA` on the required parameters: process is `Waitu 2`, process name is `pa_2`, `branchID` is "0", `startID` is 0, `finishID` is 0, `usedNames` is a tuple of empty elements, each rule begins with empty used names. As the translation evolves, we build a collection of the names used in the translation of a process.

In the resulting output TA, the first 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.

2.3.8. 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 2.8. 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
```

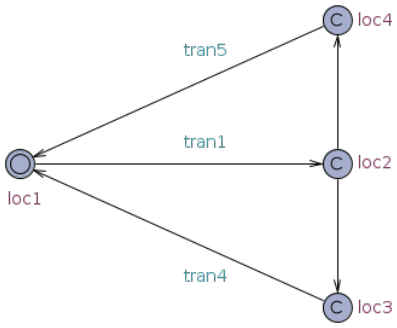


Figure 17: A structure of a TA for translating Internal choice.

Internal choice is a binary operator that combines two processes `P1` and `P2`. Rule 2.8 translates the operator of internal choice into a TA that coordinates a list of translated TA `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2` respectively, that are composed with internal choice operator.

In Figure 17, we annotate the structure of the output TA with the names used in the translation rule. The output TA begins on transition `tran1` for performing a flow action that connects the TA with the network of the TA. Then, the output TA follows one of the two silent transitions that lead to transition `tran4` and `tran5` respectively. On transition `tran4` the TA activates the list of TA `Tp1` and on transition `tran5` the TA activates the list of TA `Tp2`.

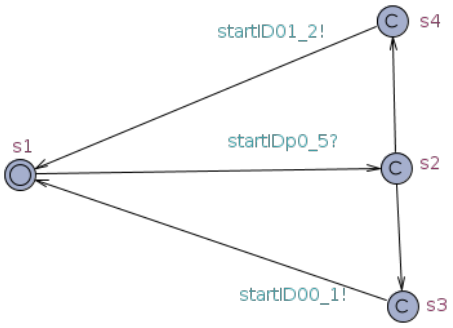
Details of Rule 2.8 are as follows. Line 1 defines the function `transTA` for the construct `IntChoice` and the 5 required parameters for this rule. Line 2 describes the output tuple that contains three elements, a list of translated TA, a list of synchronisation actions and a list of identifiers for identifying each synchronisation action.

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

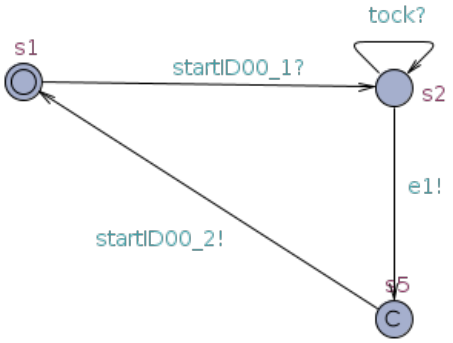
The behaviour of the output TA begins with a flow action (defined in Line 20). Then, on location `loc2`, the TA follows one of the two silent transitions, that is either `tran2` or `tran3`. Transition `tran2` leads to transition `tran4`, where the TA performs another flow action (Line 22) that activates `Tp1`. While transition `tran3` leads to transition `tran5`, where the TA performs a flow action (define Line 25) that activates `Tp2`. The following Example 2.11 illustrates using this Rule 2.8 in translating a process.

Example 2.11. An example for translating a process that composes two processes with internal choice.

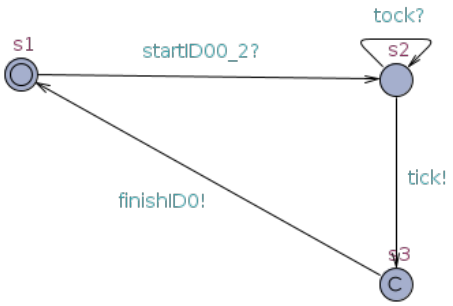
```
1 transTA((e1->SKIP)|-|(e2->SKIP)) =
2     [
```



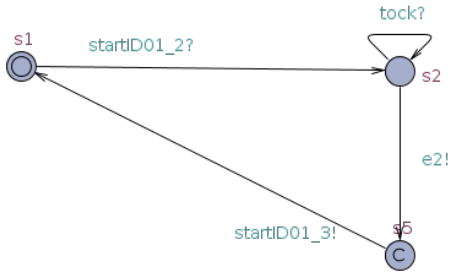

```
3       ] ++ ta1 ++ ta2
4   where
5   ta1 = transTA(e1->SKIP)
6       = [
```



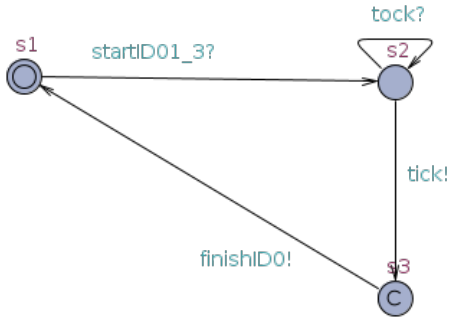

```
7
8         ] ++ transTA(SKIP)
9           = [
```

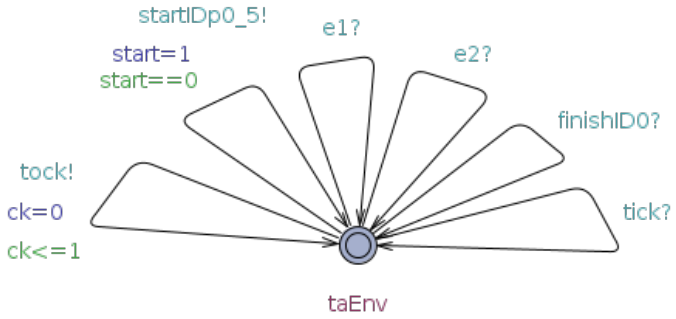



```
11      ]
12
13  ta2 = transTA(e2->SKIP)
14      = [
```




```
15
16      ] ++ = transTA(SKIP)
17           = [
```






```
]
```

Example 2.11 translates the process `((e1->SKIP)|-|(e2->SKIP))` into a list containing 5 TA. The first TA is a translation of the operator internal choice. Second and third TA are translations of the LHS process `(e1->SKIP)`. Where the second TA is a translation of the prefix event `e1` using Rule 2.5. And the third TA is a translation of the subsequent process `SKIP` using Rule 2.3. Fourth and fifth TA are translations of the RHS process `(e2->SKIP)`. Fourth TA is a translation of the prefix event `e2` using Rule 2.5. While, the fifth TA is a translation of the subsequent process `SKIP` using Rule 2.3. Finally, the last TA is an environment TA for the list of the translated

TA. This completes the list of the output TA that capture the behaviour of the process `((e1->SKIP)|-|(e2->SKIP))`.

2.3.9. Translation of External Choice

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

Rule 2.9 defines a translation of external choice. The operator of external choice ([]) is another binary operator that combines two processes `P1` and `P2`. Rule 2.9 translates the operator external choice into a TA that coordinates a lists of translated TA: `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2`, respectively.

In Figure 18, we annotate the structure of the output TA with the names used in the translation rule. The output TA has three transitions, and three locations defined in Lines 7–10 and 12–15, respectively. Then, Lines 17–21 define the corresponding labels of the transitions. Lines 27 extracts the initials of the external choice and then updates the initials in Lines 31–36. Finally, Lines 39–42 define the subsequent translation of processes `P1` and `P2`, which produces list of TA: `Tp1` and `Tp2`, respectively.

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

An essential part of translating external choice is translating both processes such that choosing one process blocks the behaviour of the other process. We achieved this, with additional transitions in the first TA of each of the translated processes for the external choice, as discussed in Section 2.2. In the parameters of the translation function `transTA`, the parameter `usedNames` contains the initials of the processes for external choices, which is updated in Lines 31–36, and then use in translating each process, specifically in constructing the transition of blocking external choice that has co-actions (initials of the other processes) for blocking the process that is not chosen by the environment. The following Example 2.12 illustrates using this rule in translating a process that composes two processes with the operator of external choice.

Rule 2.9. Translation of External Choice

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

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

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

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

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

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

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

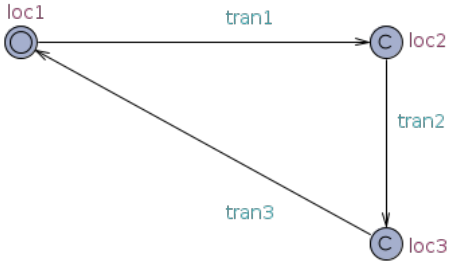
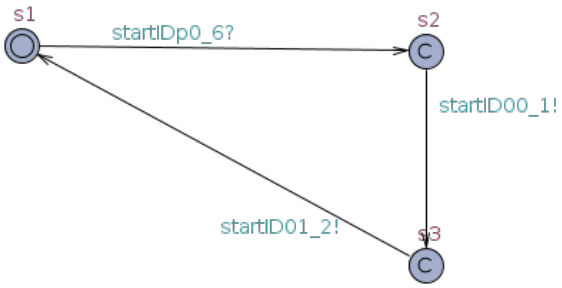


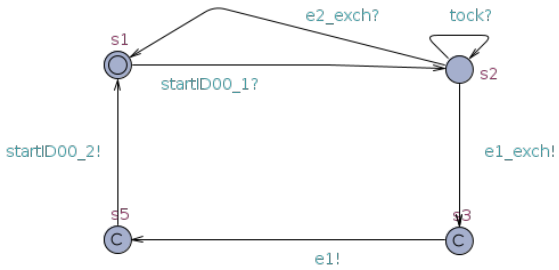
Figure 18: A structure of the control TA for the translation of external choice.

Example 2.12. An example of translating a process that composes two processes with the operator of external choice.

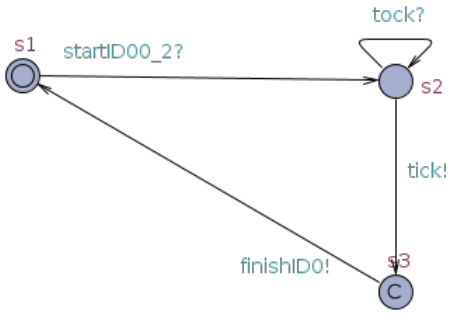
```
transTA((e1->SKIP)[](e2->SKIP))
  = [
```



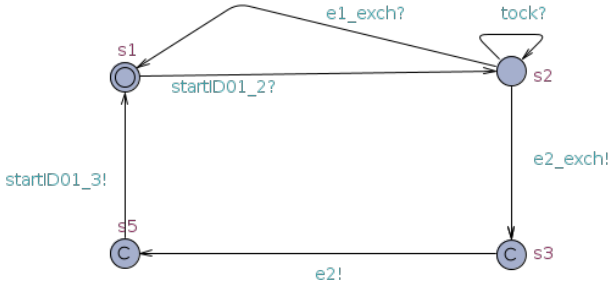

```
        ] ++ ta1 ++ ta2
where
  ta1 = transTA(e1->SKIP)
      = [
```



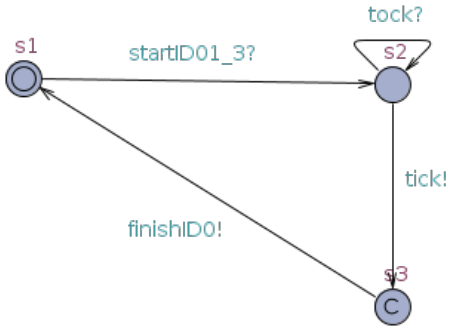

```
        ] ++ transTA(SKIP)
        = [
```

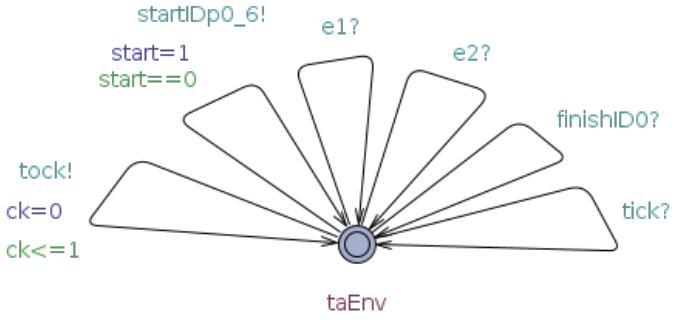



```
49              ]
50      ta2 = transTA(e2->SKIP)
51          = [
```




```
52              ] ++ transTA(SKIP)
53                 = [
```





]

Example 2.12 illustrates using Rule 2.9 in translating a process `((e1->SKIP)[](e2->SKIP))` into a list of TA that contains 5 TA. The first TA is a translation of the operator external choice. The TA has three transitions, each labelled with a flow action, `startIDp0_6` `startID00_1` and `startId01_2`. Initially, the behaviour of the TA synchronises on the first flow action `startIDp0_6` and then immediately performs the two subsequent flow actions `startID00_1` and `startId01_2` that activates the list of TA for the translations of the processes: `(e1->SKIP)` and `(e2->SKIP)`, respectively.

Second and third TA are translations of the LHS process `e1->SKIP`. Second TA is a translation the event `e1`, which synchronises on the flow action `startID00_1` and moves to location `s2` where the TA has three possible transitions: `e1_exch?` `e2_exch?` and `tock?`. On the transition `tock?`, the TA performs the action `tock` for the progress of time. On transition `e2_exch?` the TA performs a blocking event when the environment chooses the other action `e2`. Lastly, on transition `e1_exch!` the TA performs the action `e1_exch!` when the environment chooses the action `e1` for the behaviour `Tp1`. First, the action `e1_exch!` synchronise with its co-action `e1_exch?` to block the alternative behaviour of `Tp2`, and then immediately proceeds with performing the chosen action `e1` that leads to the subsequent flow action `startID00_2`, which activates the subsequent TA third TA. The third TA is a translation of the subsequent process `SKIP` for the LHS process `e1->SKIP` translated with Rule 2.3.

Fourth and Fifth TA are translations of the RHS process `(e2->SKIP)`. Fourth TA is a translation of the event `e2` using Rule 2.5, similar to the previous translation of the first TA; and the fifth TA is a translation of the remaining process `SKIP` using Rule 2.3. Finally, the last TA is an environment TA for the list of the translated TA, as defined in Section 2.2. This completes the description of translating the process `(e1->SKIP)[](e2->SKIP)` into a list of TA.

2.3.10. Translation of Sequential Composition

This section describes a translation of the 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 2.10. 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 for sequential composition is another binary operator that composes two processes `P1` and `P2` sequentially. Like the previous translation rules, this rule translates the operator sequential composition into a control TA, which coordinates the list of TA `Tp1` and `Tp2` for the translation of the two processes `P1` and `P2`.

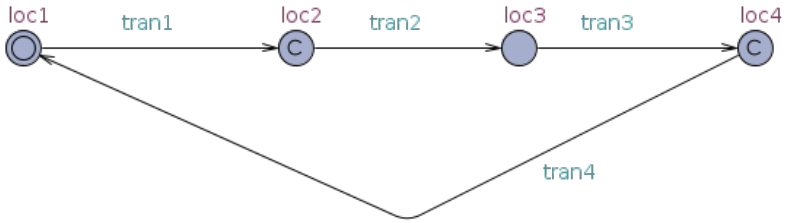
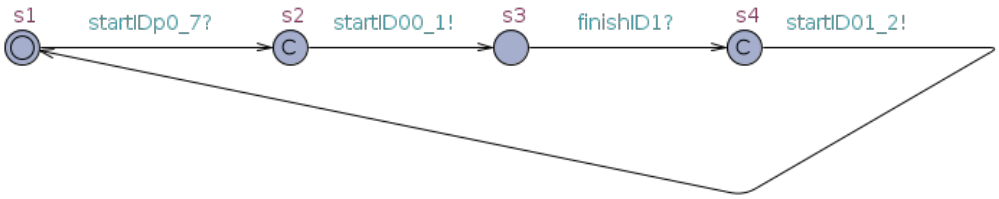


Figure 19: A structure of the control TA for the translation of sequential composition.

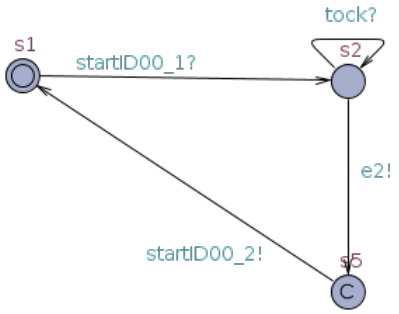
In Figure 19, we annotate the structure of the output TA with the names used in the translation Rule 2.10. The TA has four locations and four transitions that are defined in Lines 7–11 and Lines 13–17, respectively. In Rule 2.10, the behaviour of the output TA begins with synchronising on the first flow action (Line 13) and then immediately performs another two flow action on transition `tran2` (Line 14) to activate the translation of the LHS process `Tp1`. After that, the control TA waits on location `loc3` until the TA synchronises on a terminating action on transition `tran3` (Line 15), which indicates the termination of the first process `Tp1` and then immediately activates `Tp2` which proceeds up to its termination point. The following Example 2.13 illustrates using this rule in translating a process.

Example 2.13. An example for translating a process that composes two processes with the operator for sequential composition.

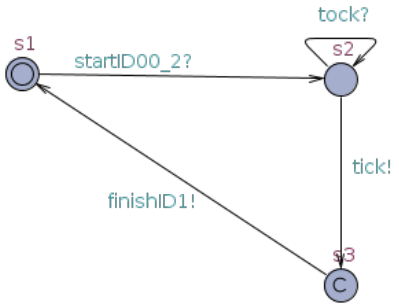
```
1 transTA((e2->SKIP);(e1->SKIP)) = [
```



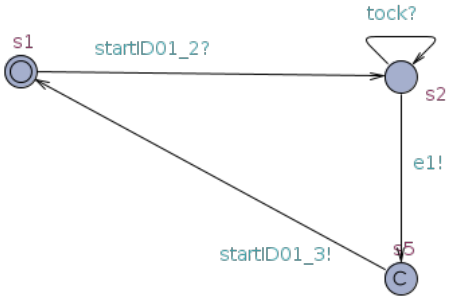

```
2     ] ++ ta1 ++ ta2
3 where
4 ta1 = transTA(e2->SKIP)
5     = [
```



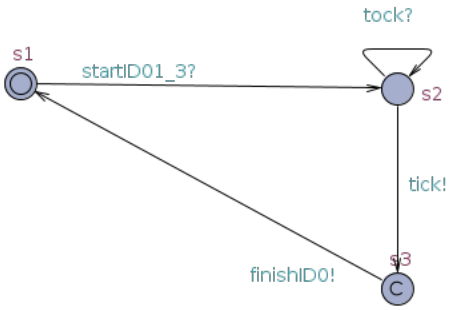
```
6    ] ++ transTA(SKIP)
7       = [
```

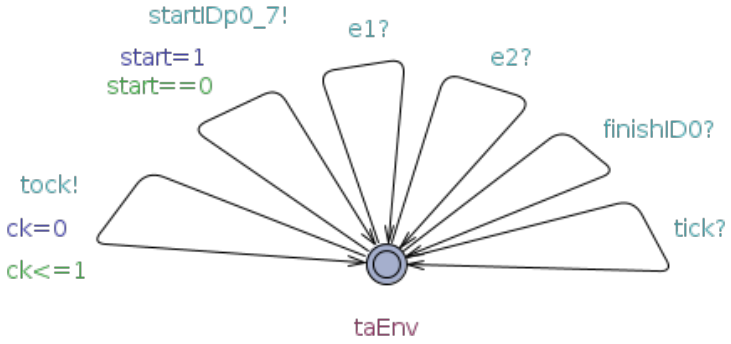



```
8            ]
9  ta2 = transTA(e1->SKIP)
10      = [
```




```
11      ] ++ = transTA(SKIP)
12           = [
```





13]

Example 2.13 illustrates using Rule 2.10 in translating the process `((e2->SKIP);(e1->SKIP))` into a list of TA that contains 5 TA. The first TA is a translation of the operator sequential composition using Rule 2.10. The second and third TA are translations of the LHS process `(e2-> SKIP)`, while the fourth and fifth TA are translations of the RHS process `(e1-> SKIP)`. Finally, the last TA is an environment TA for the list of the translated TA.

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

2.3.11. Translation of Generalised Parallel

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

Rule 2.11. Translation of Generalised Parallel

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

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

The operator generalised parallel is another binary operator, which is composed of two processes `P1` and `P2` that run in parallel and synchronise on a specified set of synchronisation events. The construct `GenPar` is translated into two TA: a control TA and a synchronisation TA. The synchronisation TA (Definition 2.10) coordinates the synchronisation of the translated processes `Tp1` and `Tp2`. While the control TA coordinates the behaviour of the translated processes `Tp1` and `Tp2`.

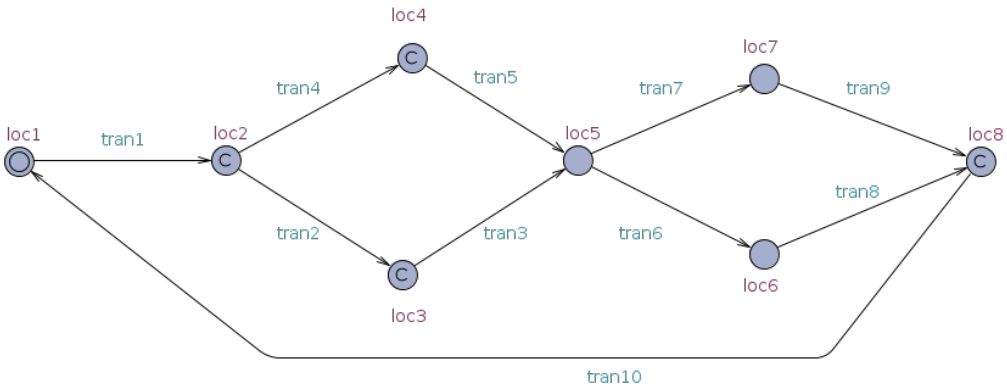


Figure 20: A structure of TA for the translation of operator Generalised Parallel

In Figure 20, 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 ten transitions, as defined 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 another new name that contains an updated list of synchronisation names `syncEv` in Lines 38–40. Then, Lines 41–44 is a recursive call for the translation of the processes P1 and P2, which produces the list of TA, `Tp1` and `Tp2`.

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

The translated processes synchronise on a multi-synchronisation action, which synchronises more than two translated TA: at least two translated processes and the Environment TA. As highlighted in the translation strategy (Section 2.2), in handling multi-synchronisation, we adopt a centralised approach [44, 45] for using a separate controller in handling multi-synchronisation. In this work, we implement the approach in a functional style with Haskell. In Definition 2.10, the function `syncTA` coordinates the synchronisation of multi-synchronisation actions.

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

When the synchronisation controller receives all the synchronisation requests, which indicates that all the synchronisation clients are ready for the synchronisation, then a guard for performing the multi-synchronisation action is enabled, and `syncTA` communicates the multi-synchronisation action to the environment and then also immediately broadcast the multi-synchronisation action that responds to all the awaiting client TA.

On receiving the broadcast multi-synchronisation response, all the awaiting client TA synchronise and proceed. An example of using this rule in translating a process is illustrated in the following Example 2.14, which demonstrates using the rule in translating concurrent processes that are composed with the operator generalised parallel `GenPar` in the specification of the process.

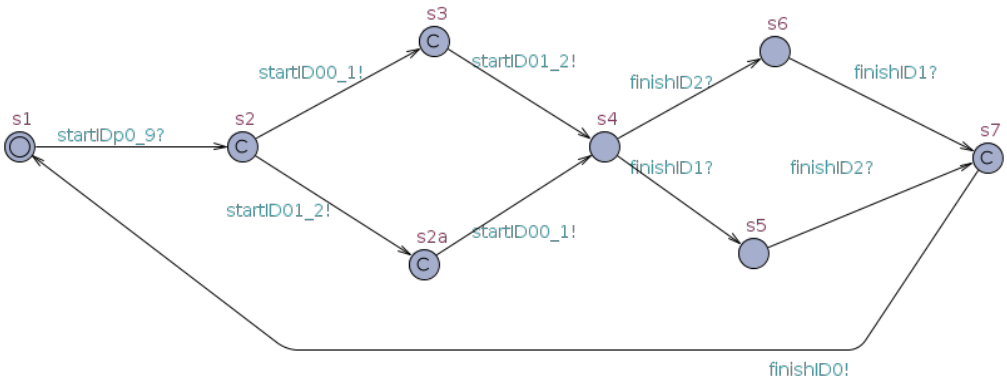
In Definition 2.10, we define a function for the synchronisation TA `syncTA` which takes two parameters, a list of synchronisation actions and a list of pairs that assign an identifier to each synchronisation action. The output of the function is a TA with an identifier "syncTA" (Line 3). The output TA has one starting location and one location for each synchronisation action as defined in Definition 2.10 (Lines 4–7). Line 8 defines a function for generating the transitions of the TA. Each synchronisation action has two transitions one from the initial location and the second transition back to the initial location. The first transition has a guard that is only enabled when all the synchronisation TA becomes ready for the synchronisation. This is illustrated in Example 2.14.

Definition 2.10. Synchronisation TA

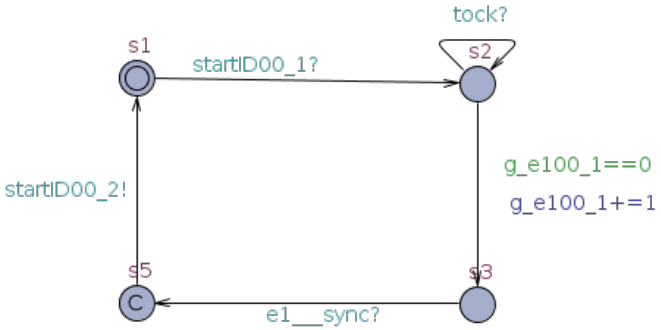
```
1  syncTA :: [Event] -> [SyncPoint] -> TA
2  syncTA     events     syncMaps   =
3      TA "SyncTA" [] [] (loc:locs) [] (Init loc) trans
4      where
5        loc  = Location "SyncPoint" "SyncPoint" EmptyLabel None
6        locs = [(Location ("s"++ show e) ("s"++ show e) EmptyLabel
7                          CommittedLoc) | e <- uniq events]
8        trans = transGen  loc (uniq events) syncMaps events
9
10 -- Generates transitions for the sync controller
11     transGen :: Location->[Event]->[SyncPoint]->[Event]->[
           Transition]
12     transGen    l0        []        _            _     = []
13     transGen    l0        (e:es)   syncMaps     syncs  =
14         [(Transition
15            l0 l
16            (Sync (VariableID (show e) []) Excl),
17            (Guard
18               (ExpID
19                 ((addExpr
20                   [("g_" ++ tag)|(e1, tag) <- syncMaps, e == e1
                         ])
21                   ++ "␣==␣" ++
22                   show ((length [e1 | e1 <- syncs, e == e1 ]) +
                         1) ) ) ),
23            (Update ([ AssgExp (ExpID ("g_" ++ tag))
24                       ASSIGNMENT (Val 0) |(e1, tag) <- syncMaps,
25                       e == e1] )) ] [])]
26         ++ [Transition
27              l l0
28              [(Sync (VariableID ((show e) ++ "___sync") []) Excl
                    )] [] ]
29         ++ (transGen l0 es syncMaps syncs)
30         where
31           l = (Location ("s"++ show e) ("s"++ show e)
32                         EmptyLabel CommittedLoc)
```

Example 2.14. An example that illustrates using both Rule 2.11 and the definition of synchronisation controller (Definition 2.10) in translating a process.

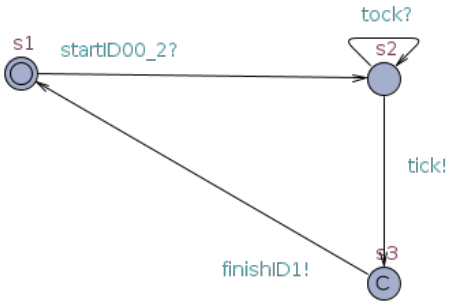
```
1 transTA((e1->SKIP)[|{e1}|](e1->SKIP)) = [
```



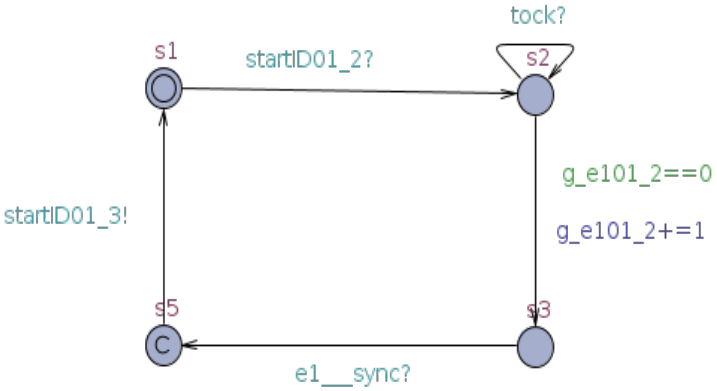

```
2        ] ++ ta1 ++ ta2
3
4 where
5 ta1 = transTA(e1->SKIP)
6     = [
```



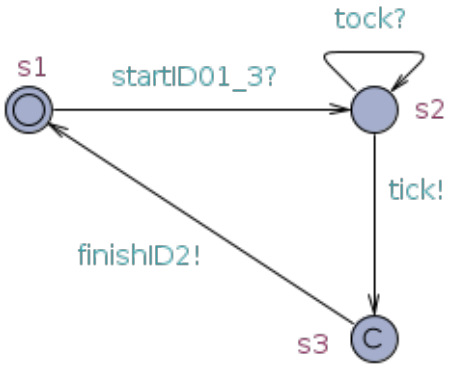

```
7   ] ++ transTA(SKIP)
8       = [
```

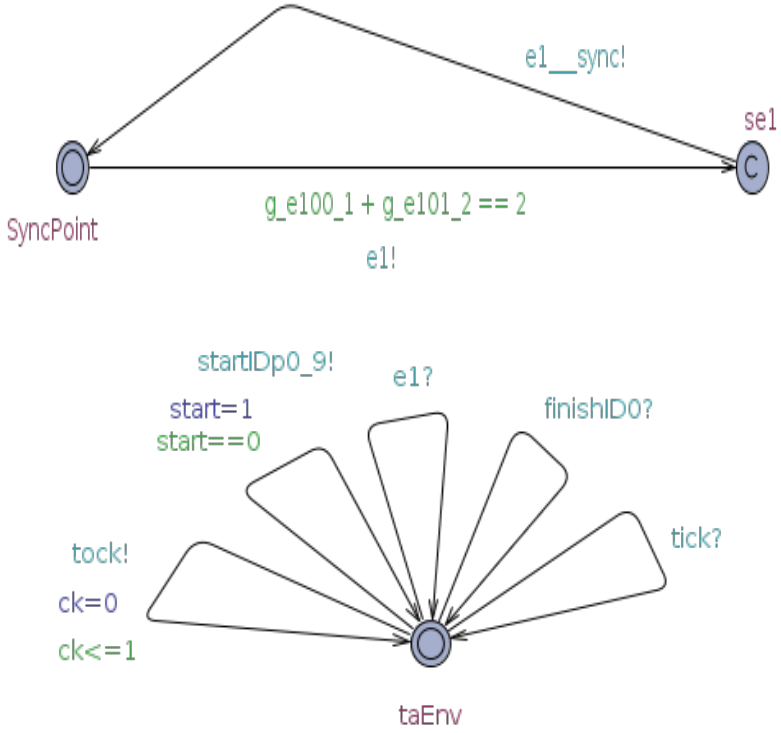



```
9             ]
10
11  ta2 = transTA(e2->SKIP)
12        = [
```




```
13          ] ++ transTA(SKIP)
14            = [
```





]

Example 2.14 illustrates using Rule 2.11 in translating the process `(e1->SKIP)[|{e1}|](e1->SKIP)` into a list of TAs that contains seven TAs. The first TA captures the translation of the operator generalised parallel. The second and third TA captures the translation of the LHS process. The fourth and fifth TA captures the translation of the RHS process. The sixth TA is a synchronisation TA that coordinates the synchronisation of the action `e1`. Finally, the last TA is an environment TA for the list of the translated TA for the process `(e1->SKIP)[|{e1}|](e1->SKIP)`.

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

The second TA synchronises on the flow action `startID00_1` and then updates its guard to indicate its readiness to synchronise on the multi-synchronisation action `e1`. Then, on receiving a response for the synchronisation, the TA synchronises on the broadcast multi-synchronisation action `e1___sync?`, which enables the TA to proceed

with performing another flow action that activated the third TA, which captures the translation of the subsequent process `SKIP` as described in Rule 2.3. Similarly, the fourth and fifth TA captures the translation of the RHS process `(e1->SKIP)`.

2.3.12. Translation of Interleaving

This section describes the translation of the 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.

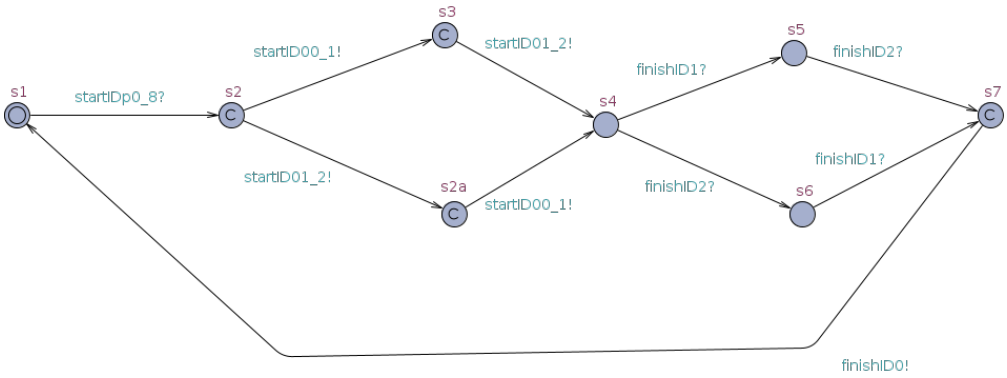
Rule 2.12. Translation of Interleaving

```
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
```

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

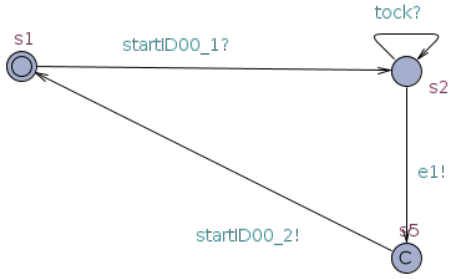
Example 2.15. An example of translating a *tock-CSP* process that composes processes with the operator interleaving using Rule 2.12.

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

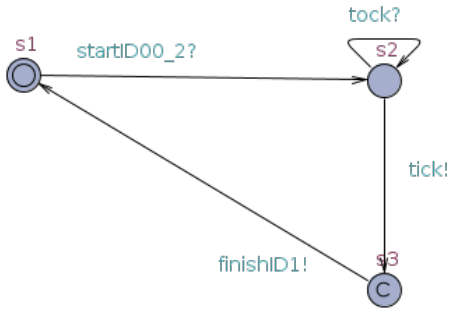



```
2        ] ++ ta1 ++ ta2
3 where
```

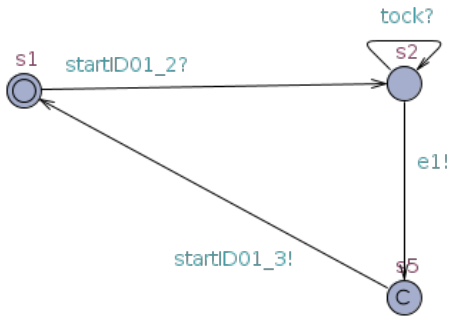
```
4 ta1 = transTA(e1->SKIP)
5      = [
```



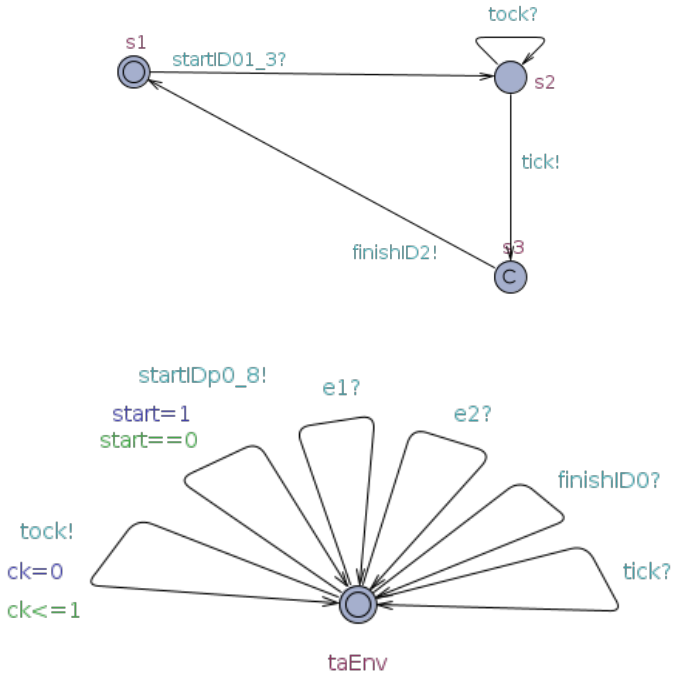

```
6    ] ++ transTA(SKIP)
7       = [
```




```
8           ]
9 ta2 = transTA(e1->SKIP)
10     = [
```




```
11        ] ++ transTA(SKIP)
12          = [
```



]

Example 2.15 illustrates using Rule 2.12 in translating a process into a list containing 6 TA. The first TA is a translation of the operator Interleaving. Second and third TA are translations of the LHS process `(e1->SKIP)`. Fourth and fifth TA are translations of the RHS process `e1->SKIP)`. The last TA is an environment TA for the list of the translated TA.

The behaviour of the TA is as follows. The first TA synchronises on the coordinating start event `startIDp0_8` and then immediately performs two flow actions `starID00_1` and `startID01_2` simultaneously that activate the translation of the LHS and RHS processes respectively. And then the first TA waits on location `s3` until it synchronises on a termination action, either `finishID1` or `finishID2`, and then waits for the second termination action `finishID1` or `finishID2` depending on the first terminating process. The action `finishID1` is termination action of the translated LHS process `(e1->SKIP)`, and `finishID1` is the termination action of the translated RHS process `(e1->SKIP)`. Then, the first TA performs another termination action to record the termination of the whole process.

The second TA is a translation of the event `e1` using Rule 2.5. While, the third TA is a translation of the subsequent process `SKIP` using Rule 2.3. Also, TA3 is a translation of the event `e2` using Rule 2.5. While TA4 is a translation of the subsequent process `SKIP` using Rule 2.3. This completes the description the translated TA in Example 2.15 that demonstrates using Rule 2.12 in translating the process `(e1->SKIP)|||(e1->SKIP)` into a list of TAs.

2.3.13. Translation of Interrupt

This section describes the translation of the 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 2.13. 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''
```

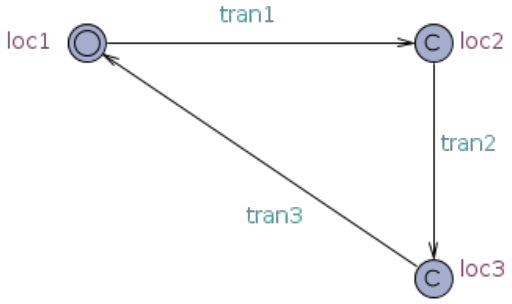


Figure 21: A TA for the structure of the translation of the operator interrupt.

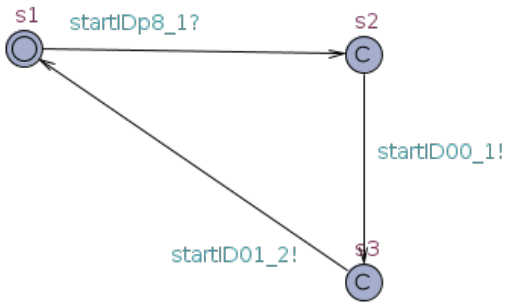
The operator Interrupt is also another binary operator that comprises two processes `P1` and `P2` in such a way that the process `P1` begins its behaviour that can be interrupted by process `P2`, whenever process `P2` performs an event. The operator Interrupt is translated into a TA that coordinates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

In Figure 21, we annotate the structure of the translated TA with the names used in the translation rule. The translated TA has 3 locations and three transitions defined in Lines 7–10 and Lines 12–15. Lines 17–22 define labels for the transitions. Line 24 extracts the list of used names for interrupt. Lines 28–29 provides a new name for the updated list of the initials of the interrupting process `p2`. Also, Lines 31–34 provides a new name for the updated tuples of the used names `usedNames`. Lines 36–40 define the subsequent translation of the LHS and RHS processes, that is `Tp1` and `Tp2`, respectively.

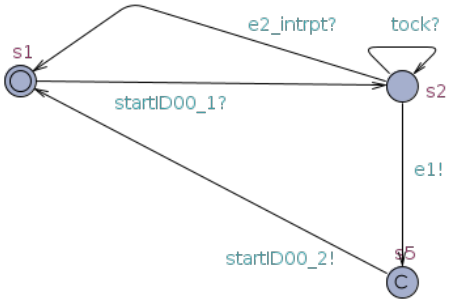
The behaviour of the control TA begins on transition `tran1` for performing a flow action. And then immediately activates the translation of the processes `Tp1` and `Tp2`. For the translation of the interrupted process `Tp1`, each TA in the list `Tp1` has an additional interrupting transition in each stable location, as described in the Translation strategy. The additional transition provides a co-action of the initials of the interrupting process `Tp2`, which enables `Tp2` to interrupt `Tp1` at any stable location. The following example 2.16 demonstrates using the rule in translating a process.

Example 2.16. An example of using Rule 2.13 in translating the process `((e1-> SKIP)/\(e2-> SKIP))` into a list of TA as follows.

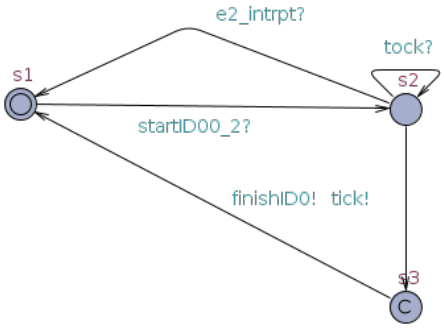
```
1 transTA((e1-> SKIP)/\(e2-> SKIP)) = [
```



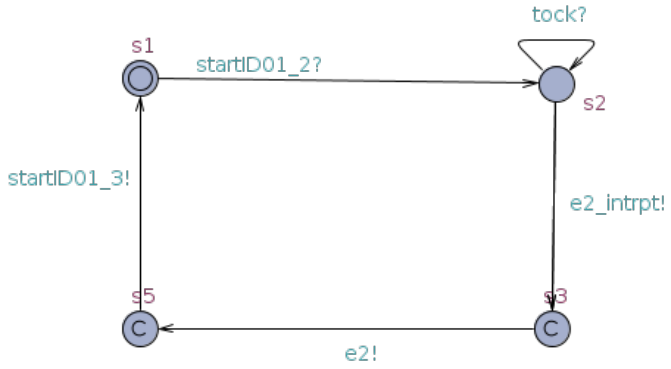

```
2        ] ++ ta1 ++ ta2
3
4 where
5 ta1 = transTA(e1->SKIP)
6     = [
```



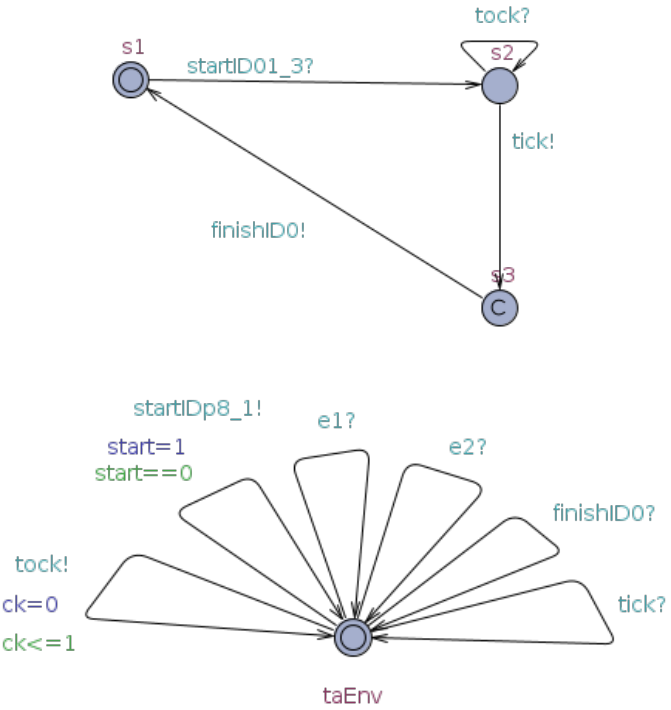

```
7   ] ++ transTA(SKIP)
8       = [
```



```
9            ]
10
11 ta2 = transTA(e2->SKIP)
12       = [
```




```
13         ] = transTA(SKIP)
14           = [
```




```
]
```

Example 2.16 demonstrates using Rule 2.13 a translation of the process into a list containing 5 TA. The first TA is a translation of the operator Interrupt. Second and

third TA are translations of the LHS process `(e1->SKIP)`. The second TA is a translation of the event `e1` using Rule 2.5. And the third TA2 is a translation of the subsequent process `SKIP` using Rule 2.6. While the fourth and fifth TAs are the translation of the RHS process `(e2->SKIP)`. Fourth TA is a translation of the event `e2`. Fifth TA is a translation of the subsequent process `SKIP`. The last TA is an environment TA for the list of translated TA.

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

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

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

2.3.14. 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 2.14. 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

  -- Assigns a new name for the updates list usednames
  (syncEv, syncPoint, hide, rename, exChs, intrr, iniIntrr,
     excps) = usedNames
  excps' = (((fst excps) ++ es), snd excps)
  usedNames' = (syncEv, syncPoint, hide, rename, exChs, intrr,
     iniIntrr, excps')

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

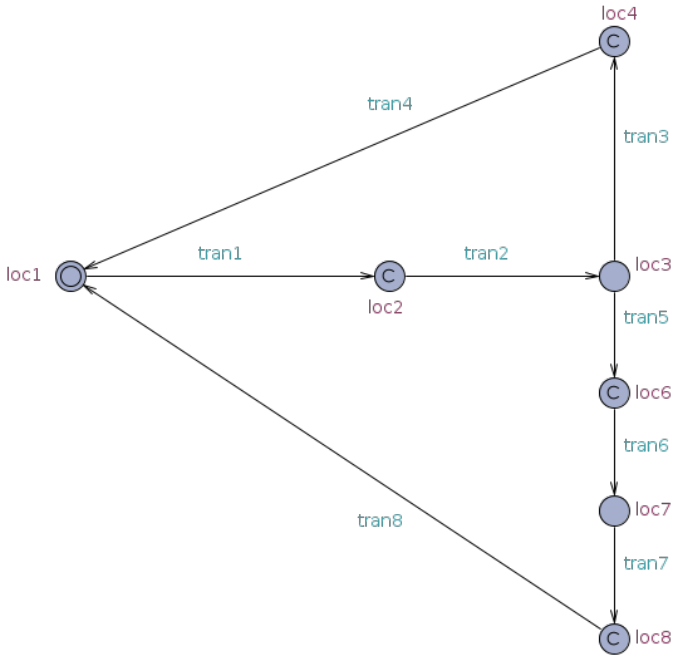


Figure 22: A structure of the control TA for the translation of the operator Exception.

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

In Figure 22, we annotate the structure of the control TA for the operator exception with the names used in the translation Rule 2.14. The translated TA has eight locations and eight transitions defined in Lines 6–13 and Lines 14–22 respectively. Then, Lines 23–31 define the labels of the transitions. Lines 33–38 extracts and updates the list of names `excps`, which we used for handling exceptions in the tuples `usedNames`. Finally, Lines 41–44 define a recursive call for the subsequent translation of the remaining processes.

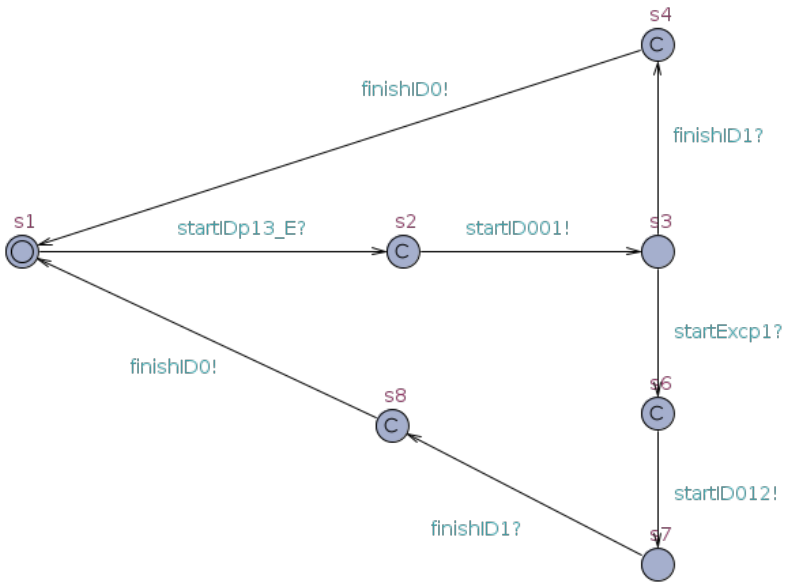
The behaviour of the control TA begins on transition `tran1` for performing a flow action. Then, on transition `tran2` the control TA performs another flow action that activates `Tp1`. After that, the control TA remains on location `loc3` until either `Tp1` terminates successfully or performs an exceptional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

Alternatively, if `Tp1` performs an exception action that raises an exception action, the control TA synchronises with its co-action on transition `tran5`, and then immediately

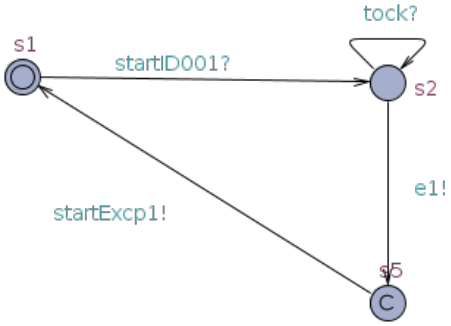
initiates the translation of `Tp2` on transition `tran6`, and then waits on location`loc7` until the translated list of TA `Tp2` performs a termination action and the control TA synchronises with the corresponding co-action on transition `tran7` and then on transition `tran8`, the control TA immediately performs another termination action for terminating the whole process. The following Example 2.17 illustrates using the Rule 2.14 in translating a process.

Example 2.17. An example that illustrates using Rule 2.14 in translating a process. This example translates the process `((e1->SKIP)[|{e1}|>(e2->SKIP))` into a list of TA as follows.

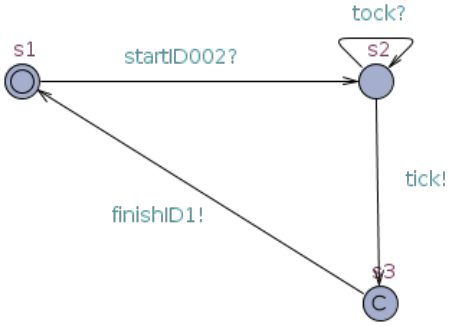
```
1 transTA((e1->SKIP)[|{e1}|>(e2->SKIP)) = [
```



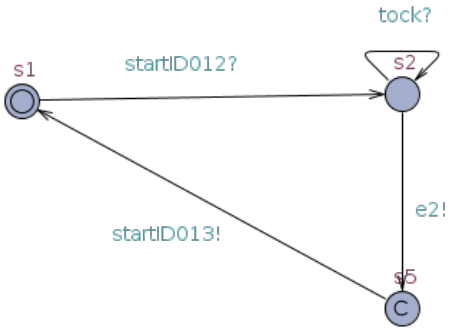

```
2       ] ++ ta1 ++ ta2
3
4 where
5 ta1 = transTA(e1->SKIP)
6     = [
```



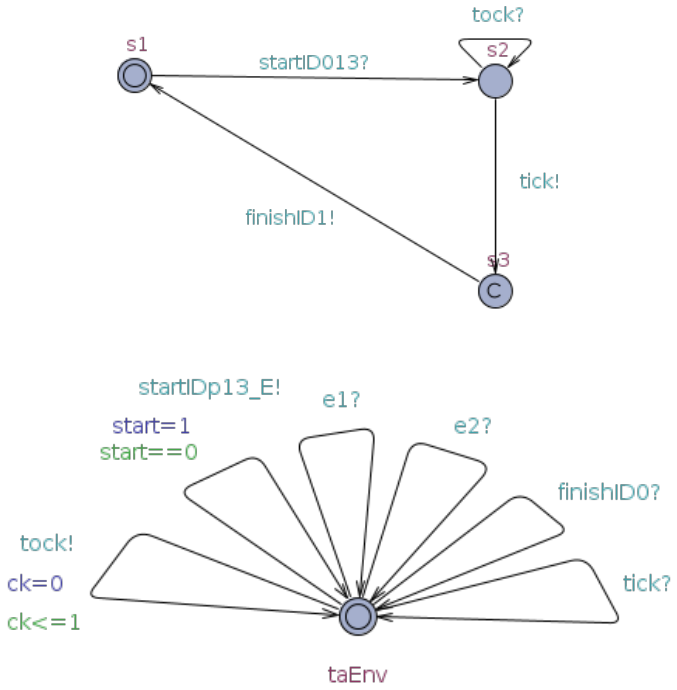

```
7    ] ++ transTA(SKIP)
8       = [
```




```
9           ]
10
11 ta2 = transTA(e2->SKIP)
12     = [
```




```
13        ] = transTA(SKIP)
14          = [
```



]

Example 2.17 translates the process `(e1->SKIP)[|{e1}|>(e2->SKIP)` into a list containing 5 TA. The first TA is a translation of the operator interrupt using Rule 2.14. Second and third TAs are translations of the LHS process `(e1->SKIP)`. Second TA captures the translation of the event `e1` using Rule 2.5. While the third TA captures the translation of the subsequent process `SKIP` using Rule 2.3. Similarly, the fourth and fifth TA are translations of the RHS process `(e2->SKIP)`. The fourth TA captures the translation of the event `e2` using Rule 2.5. The fifth TA captures the translation of the subsequent process `SKIP` using Rule 2.3.

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

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

2.3.15. Translation of Timeout

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

According to Roscoe [4], the operator timeout specifies a deadline for the LHS process to perform an event before the deadline or the process the RHS process begins its behaviour and the whole process behaves as the RHS process. In *tock-CSP*, this is express in term of internal choice and delay process as follows:

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

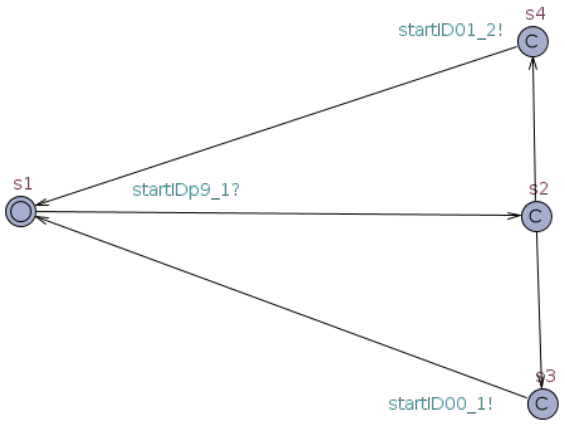
We follow a similar pattern in translating the operator timeout. We translate the operator in term of the two previous rules for translating internal choice (`|-|`) and a process delay `WAIT(d)`. Rule 2.15 expresses the translation rule and Example 2.18 demonstrates using the rule in translating a *tock-CSP* process.

Rule 2.15. Translation of Timeout

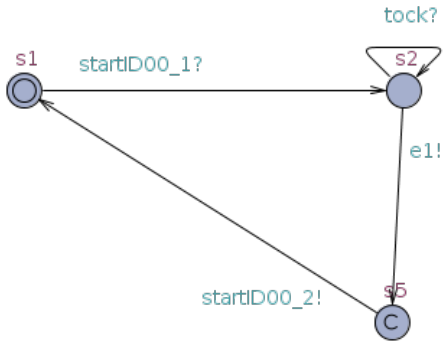
```
1 transTA (Timeout p1 p2 d) procName bid sid fid usedNames =
2         transTA (IntChoice p1 (Seq (WAIT d) p2 )) procName bid
              sid fid usedNames
```

Example 2.18. An example that illustrates a translation of the process using Rule 2.14. This example translates the process `((e1->SKIP)[2>(e2->SKIP))` into a list of TA as follows.

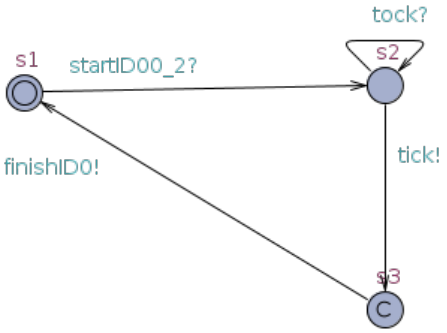
```
transTA ((e1->SKIP)[2>(e2->SKIP)) =
  transTA ((e1->SKIP)|~|(WAIT(2);(e2->SKIP))) = [
```



```
1        ] ++ ta1 ++ ta2
2
3  where
4  ta1 = transTA(e1->SKIP)
5      = [
```

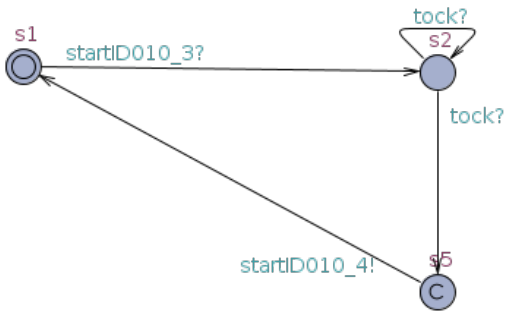



```
6    ] ++ transTA(SKIP)
7       = [
```

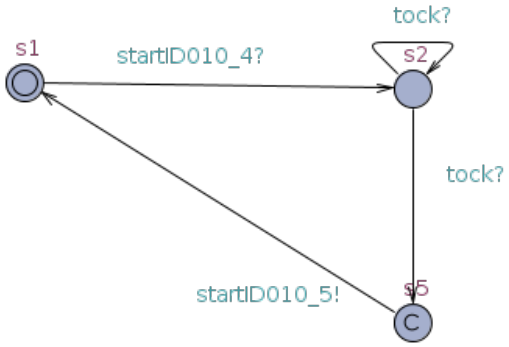



```
8           ]
9
10 ta2 = transTA ((WAIT 2);(e2->SKIP))
11     = [
```

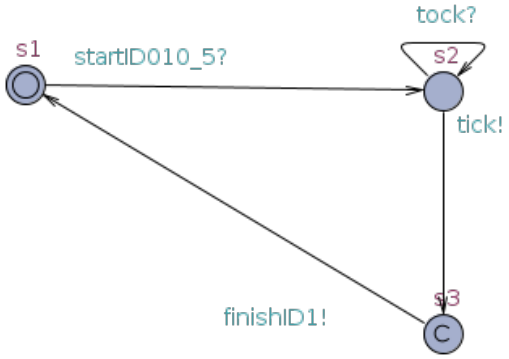
```
12        ] = ta21 ++ ta22
13 ta21 = transTA(WAIT 2)
14       = [
```



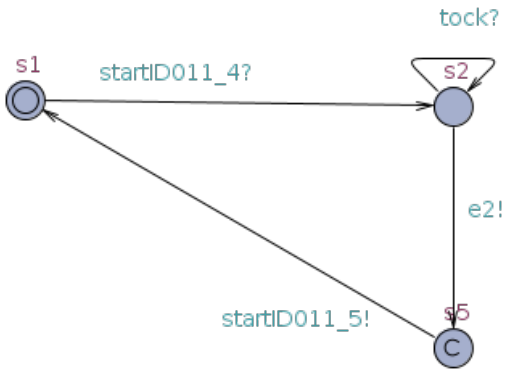

```
15    ] ++ transTA(WAIT 1)
16       = [
```



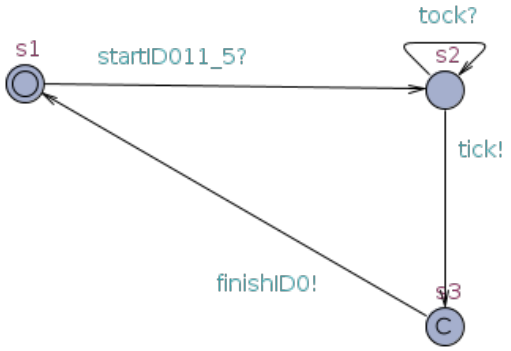

```
17    ] ++ transTA(SKIP)
18       = [
```

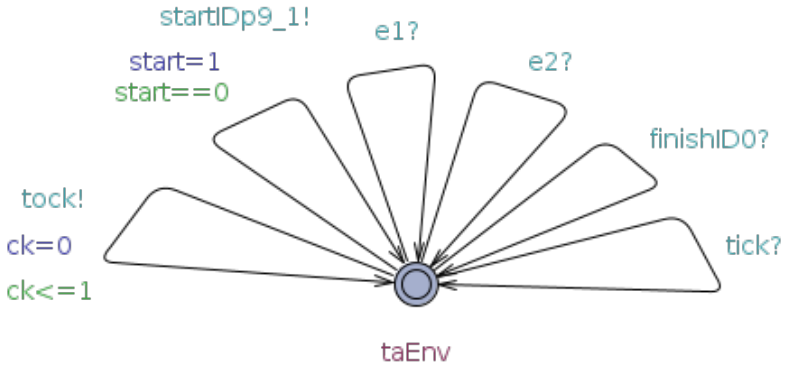


```
19            ]
20
21 ta22 = transTA(e2->SKIP)
22        = [
```




```
23    ] ++ transTA(SKIP)
24        = [
```






```
          ]
```

In Example 2.18, we illustrates using Rule 2.15 in the translation of the process `((e1->SKIP)[2>(e2->SKIP))` into a list of TA containing 10 TAs. The first TA is translation of the operator for internal choice. The second and third is translation of the LHS process `(e1->SKIP)`. The second TA captures the translation of the event `e1` according to Rule 2.5. The third TA captures the translation of the subsequent process `SKIP`. From the fourth to the ninth TA are translations of the RHS process `(WAIT(2);(e2->SKIP))`. The fourth TA is a translation of the operator sequential composition according to Rule 2.10. The fifth and sixth TA are translations of the delay process `WAIT(2)` according to Rule 2.6. The seventh TA captures the translation of the event `e2` according to Rule 2.5. The eighth TA captures the translation of the subsequent process `SKIP`. The last TA is an environment TA for the list of translated TA of the process `((e1->SKIP)[2>(e2->SKIP))`. This completes the description of an example that illustrates using Rule 2.15 in translating a process.

2.3.16. 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 2.16 defines a translation of the construct `Edeadline` into a TA. In Figure 23, we annotate the structure of the TA with the names used in the translation rule. The TA has three locations and four transitions, as defined in Lines 6–9 and Lines 11–16 respectively. Lines 18 – 32 define the labels of the transitions. Line 24–25 defines a label for resetting the timer. Lines 27–28 update the time with one time unit after every action `tock`. Lines 30–31 define guards for controlling the deadline. The following example illustrates using this Rule 2.16 in translating a process.

Rule 2.16. Translation of EDeadline (Event Deadline)

```
transTA (EDeadline e n) procName bid sid fid usedNames =
  ([(TA idTA [] [] locs [] (Init loc1) trans)], [], [])
  where
    idTA = "taDeadln" ++ 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] ++ t_reset)    []
    tran2 = Transition loc2 loc2 ([lab2] ++
            dlguard ++ dlupdate) []
    tran3 = Transition loc2 loc3 ([lab3] ++ dlguard2)   []
    tran4 = Transition loc3 loc1 [lab4] []
    trans = [tran1, tran2, tran3, tran4] ++
            (transIntrpt intrpts loc1 loc2)

    lab1 = Sync (VariableID (startEvent procName bid sid) [])
       Ques
    lab2 = Sync (VariableID "tock"    []) Ques
    lab3 = Sync (VariableID (show e)  []) Excl
    lab4 = Sync (VariableID ("finishID" ++ show fid)   []) Excl

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

    dlupdate = [(Update [(AssgExp (ExpID "tdeadline")
                                   AddAssg (Val 1) ) ] ) ]

    dlguard  =
        [(Guard (BinaryExp (ExpID "tdeadline") Lth (Val n)))]
    dlguard2 =
        [(Guard (BinaryExp (ExpID "tdeadline") Lte (Val n)))]

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

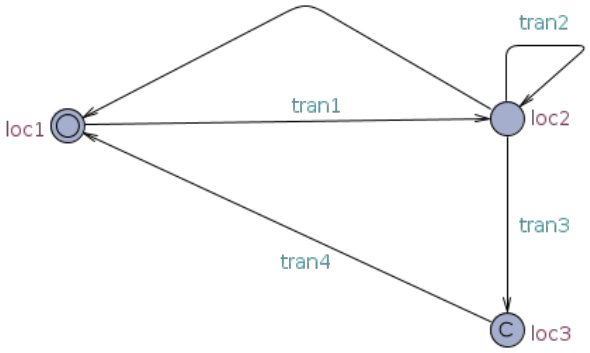
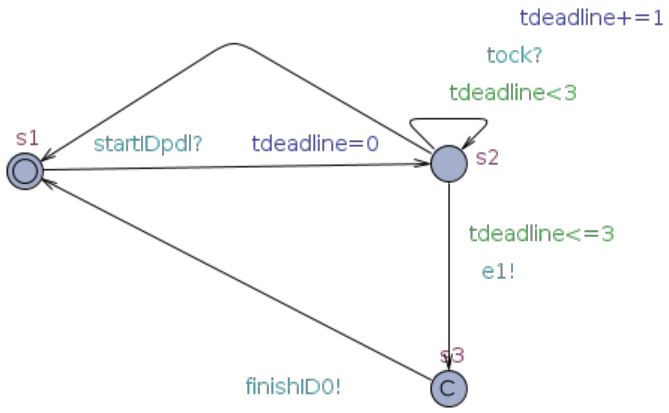
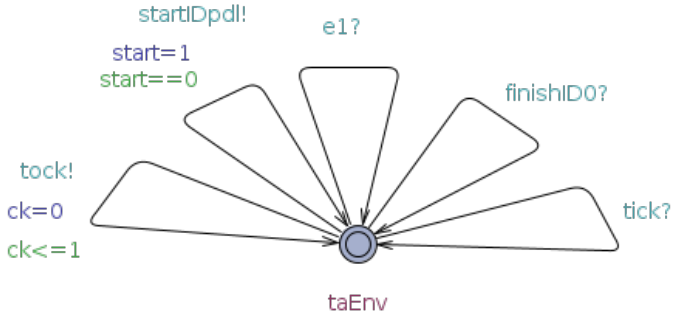


Figure 23: A structure of the control TA for the translation of the process Edeadline.

Example 2.19. This example illustrates using Rule 2.16 in translating the process `(EDeadline (e1, 3))` into a list of TA as follows.

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






```
    ]
```

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

2.3.17. Translation of Hiding

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

Rule 2.17. 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, exChs, intrrs, iniIntrrs, excps)
```

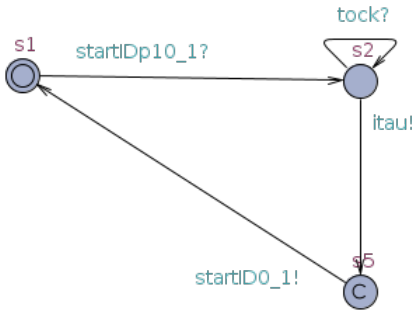
Rule 2.17 updates the used name for hiding `hides`, which is used in the subsequent translations that are handled in Rule 2.5. Line 1 defines the function `transTA` for the construct `Hiding`. Line 2 describes the output in terms of the function `transTA` with an updated name `usedNames'`, which contains an updated name `hides`. Lines 4–5 extract the name `hides` from the tuples of used names `usedNames`. Lines 8–9 updates the name `usedNames` with hiding events for subsequent translation.

Rule 2.5 checks the used name `hides` in translating each event. If an event is in the list of hiding events, Rule 2.5 translates the event into a special name `itau`. While, if an event is not part of the used names `hides`, Rule 2.5 translates the event with its name in the output TA. The following Example 2.20 illustrates using this rule in translating a process.

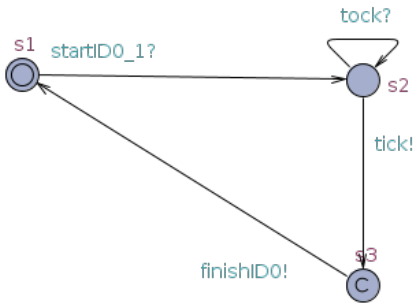
Example 2.20. This example demonstrates using Rule 2.17 in translating the process `((e1->SKIP)\{e1})` into a list of TA as follows.

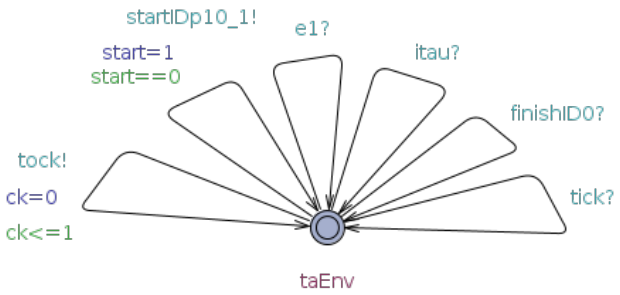
```
1 transTA((e1->SKIP)\{e1}   "p10_1" _ 0 0 usedNames ) =
```

```
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        = [
```




```
11    ] ++ transTA(SKIP)
12          = [
```






```
]
```

Example 2.20 illustrates a translation of a process using 2.17. The example translates the process `((e1->SKIP)\{e1})` into a list of TAs that contains three TAs. The first TA is a translation of the hiding event `e1` into a special name `itau`. The second TA is a translation of the subsequent process `SKIP` according to Rule 2.3. The third TA is an environment TA for the list of translated TAs.

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

2.3.18. Translation of Renaming

This section describes the translation of the 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 2.18. Translation of Renaming

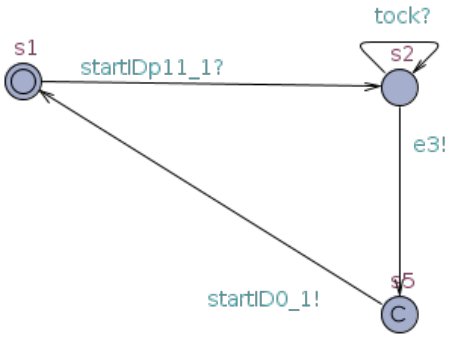
```
1 transTA (Rename p pes) procName bid sid fid usedNames
2  =  transTA    p       procName bid sid fid usedNames'
3     where
4        (syncs, syncPoints, hides, renames, exChs, intrrs,
5         iniIntrrs, excps) = usedNames
6
7        -- Updates the name renames in the list of usedNames
8      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

Translation of operator Renaming follows similar patterns with the previous Rule 2.17, except that, the parameter for renaming is a list of pairs, an event with its corresponding new name. This rule updates the used name `renames` from the tuples `usedNames`. Then in the subsequent translation, Rule 2.5 checks the updated used name `renames` in translating each event. If an event is in the list `renames`, Rule 2.5 replaces the event name with its corresponding new name, such that it appears with its new name in the translated TA.

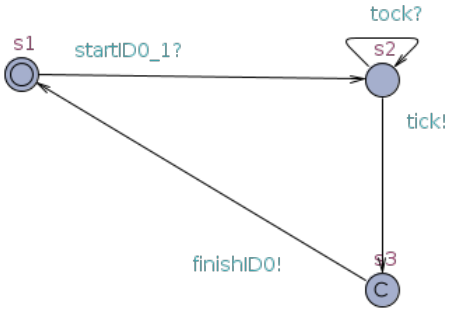
Example 2.21. An example for translating a process that demonstra a translation of the operator Renaming in translating the process `((e1->SKIP)[[e1<-e3]])`.

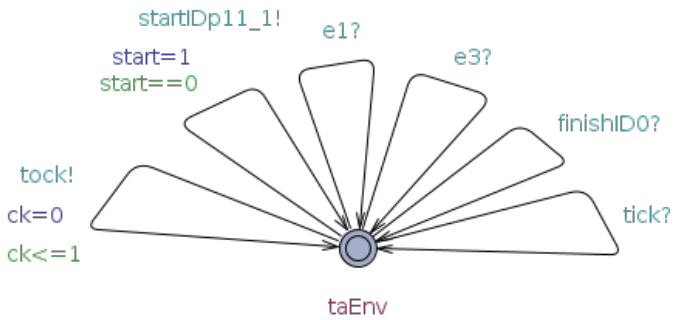
```
1 transTA(((e1-> SKIP)[[e1<-e3]]) "p11_1" [] 0 0 usedNames)
2     = transTA((e1->SKIP) "p11_1" [] 0 0 usedNames')
```

```
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)
9  transTA((e1->SKIP) "p11_1" [] 0 0 usedNames')
10     = [
```




```
11     ] ++ = transTA(SKIP)
12           = [
```





]

Example 2.21 illustrates using Rule 2.18 in translating a process into a list of TA that contains 3 TA. The first TA is a translation of the event `e1`, which appears with its new name `e3` in the translated TA. The second TA is a translation of the subsequent process `SKIP`. The last TA is an environment TA for the list of translated TA. This completes description of the translation of the process `((e1->SKIP)[[e1<-e3]])`.

2.3.19. Definition of Environment TA

This section defines an explicit environment TA for the translated TA of the Uppaal models. Definition 2.11 provides a Haskell function that expresses the structure of an explicit environment TA for the translated Uppaal system. Here we provide the definition of the environment TA, which we have seen various examples in the provided examples of the translation rules.

Definition 2.11. A Function for defining an environment TA

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

As highlighted in the translation strategy (Section 2.2), each process is translated into a list of TAs that includes an environment TA. The function defines the environment TA that has one location as defined in Line 5. While the remaining Lines 6–25 defines the transitions of the TA. First, Line 6 defines a common name used in defining the source and targets of all the transitions. Line 9 defines a transition for each action from the translated process. Lines 10–12 define the first starting flow action `startID0_0`, which initiates the behaviour of the translated TA (for the case of anonymous function). This starting transition has guards that block the environment from starting the behaviour multiple times. In the case of a named process, Lines 14–16 defines another transition for starting flow action that has the process name (for the case of a named process). Line 20 defines the final termination co-action for terminating the whole process. Line 21 defines a transition for a co-action of the translated action `tick`. Line 22 defines a transition for a co-action of hiding events `itau`. Lines 21–23 define a transition for the translated action `tock`, which is associated with a clock variable for recording the progress of time. This completes the definition of the environment TA. Also, this definition completes the presentation of the translation rules developed for translating *tock-CSP* into a list of TA.

2.4. Final Considerations

In this chapter, we discussed a technique for translating *tock-CSP* models into Uppaal models. The chapter begins with defining a BNF that served as the foundation of the translation work. The BNF describes the operators we considered for our targeted work. In essence, the developed BNF defines constructs for the selected operator of the *tock-CSP*.

Subsequently, we described the strategy we follow in achieving the translation work. Basically, we consider small size TAs joined together using additional flow actions, which we introduce in the translation work. We use the flow actions to coordinate the connections of the small TAs. The names of the flow actions were generated to be unique, and also provide a good structure for connecting the TA to form a network of TA. We consider using small sizes TAs in order to capture the compositional structure of *tock-CSP*. We used examples to illustrate using both the small TA and the flow actions that connect the translated TA.

Based on the developed translation strategy, we presented rules that precisely describe the translation of *tock-CSP* into TA. Each translation rule describes a translation of one construct of the BNF into TA. For precise description, we used Haskell notations for presenting the translation rules, where we defined a function transTA that defines a translation of each construct into TAs. The function `transTA` takes a *tock-CSP* model as an argument and provides an output list of TA that captures the behaviour of the input *tock-CSP* model. Examples were provided that illustrate using each rule in translating *tock-CSP* process. In the next chapter, we are going to discuss the evaluation of the translation technique.

Chapter Four

3. Evaluation

In this chapter, we describe the mechanisms we consider in evaluating the translation technique. In Section 3.1, we described a tool that implements the translation rules presented in Chapter 3. In Section 3.2, we describe another tool we developed for validating the translation tool. We use trace semantics for validating the translation technique. Thus, we develop a technique for generating and comparing traces of *tock-CSP* and TA.

Additionally, we use two forms of test cases for evaluating the translation tool, a collection of small processes and case studies. In Section 3.3, we describe a collection of small processes we used in assessing the correctness of the translation rules as part of the translation technique developed. In Section 3.4, we describe the second category of the test cases for evaluating the translation technique. Developing the translation technique give us a good opportunity for comparing the performance of the two model-checkers: FDR and Uppaal in analysing deadlock freedom, where the input *tock-CSP* process is constructed manually and the translated TA generated by our translation tool. Because of that, the results may not be a fair comparison, as the generated TA may not be the most efficient models for Uppaal. We describe the result of comparing the performance in Section 3.5. In Section 3.6, we describe a plan for mathematical proof of the translation technique. Finally, Section 3.7 provides a summary and conclusion of the chapter.

3.1. Mechanisation of the Translation Rules

In this section, we discuss a prototype tool we developed for automating the translation technique. After providing the translation rules, it is also useful to translate *tock-CSP* automatically, using the translation strategy to validate the translation technique. Besides, an automatic translation will enable considering the compatibility of the translation rules and other technical concerns. Thus, we create a tool for automatic translation based on our translation strategy (Chapter 3).

In implementing the translation tool, we continue using Haskell, the language we used previously in presenting the translation rules in Chapter 3, alongside using the Glasgow Haskell Compiler [46]. This reduces the complexity of combining the translation rules into a single system because the developed description of the translation rules in Haskell provides executable components of the translation tool. Figure 24 illustrates the structure of the translation tool. The implementation of the tool is available in the provided repository of the work [47].

Considering the functional structure of Haskell, we have developed another function `transform()` that encapsulates the translation rules `transTA()`, and the remaining two functions `syncTA()` and `envTA()` for synchronisation TA and environment TA, respectively (as defined in Section 2.3). The function `transform` uses the translation

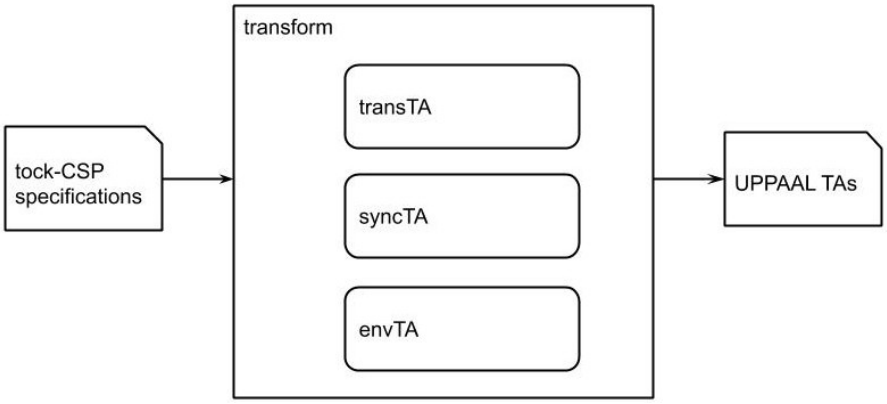


Figure 24: Structure of the function transform for the translation system

rules as part of the system that invokes the appropriate translation rule in translating each construct of the input *tock-CSP* specifications into the corresponding translated TA. Then, finally the function `transform` completes the translation with providing both synchronisation TA and an environment TA that closes the translated system, as described in Chapter 3.

In using the translation tool, the function `transform()` (Section 2.2) takes a valid name *tock-CSP* process as input Abstract Syntax Trees (AST), and initialises the required arguments for the three functions: `transTA`, `syncTA` and `envTA`. The input of the translation tool is a valid *tock-CSP* specification within the scope of this work.

The output of the function `transform()` is a network of TA for Uppaal, which is based on a provided Uppaal template for the internal representations of the networked TA, as shown in Figure 25. In this work, we encode the template as a data structure in Haskell, which captures the output of the translation technique into a suitable Uppaal TA. The definition of template TA has five major sections: header, declaration, network of TA, system definition and TCTL queries, as shown in Figure 25. First, the header section describes the configuration information of the translated TA. Second, a declaration section defines the terms used in the system. Third, a list of definitions for the list of TAs that form the networked TA. Fourth, the system definition instantiates the definitions of the TA into a system. Lastly, a list of queries describes requirement specifications of the system. An additional detailed structure of Uppaal TA is provided in Appendix B.

We provide an interface for running the translation tool, which has a list of commands for accessing the tool. The tool provides an interactive environment for running the translation tool, which accepts an input *tock-CSP*, either directly as input to the command prompt or uploading a *tock-CSP* file that contains specifications (in AST) of a system. This completes the description of the translation mechanism. In the next section, we will discuss the technique consider in generating and comparing the traces of *tock-CSP* and its corresponding translated TA.

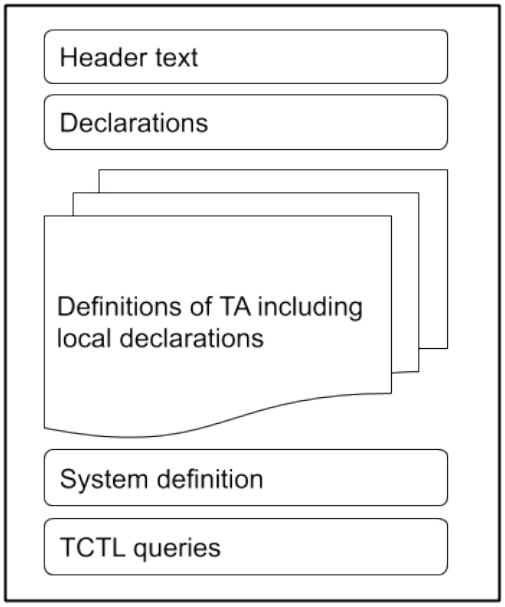


Figure 25: Structure of the UPPAAL models

3.2. Trace Analysis

This section discusses the mechanism we develop for evaluating the translation rules. A sound translation preserves properties of the source model. This is determined by comparing the behaviours of the source model and the translated model [48–51]. In this work, we use traces to compare the behaviour of the input *tock-CSP* models and the traces of the translated TA models.

We have developed another technique along with its software tool, which automates the validation technique using trace analysis. The structure of the trace analysis tool is in Figure 26, which has five components, a controller, a translation system (Section 3.1), FDR and UPPAAL as black-boxes, and two additional components for analysing traces in two stages. The tool uses the translation tool in translating *tock-CSP* model into TA, and then uses both FDR and UPPAAL as black boxes in generating sets of finite traces. Then, the tool compares the generated traces in two stages, 1st and 2nd stage.

The benefit of adding the 1st stage is because it is easier to generate the traces with the 1st stage, unless for concurrency in which the 1st stage generates incomplete traces of a process that has concurrency, due the complexity of using UPPAAL in generating traces of concurrent processes. Therefore, we create the second stage to complement the 1st stage with the additional technique for using the power of FDR to generate traces of a process that contains concurrency. The system begins with the 1st stage if the traces are complete the system terminates. Otherwise, the system invokes the 2nd stage for computing the remaining traces.

In Figure 26, the controller connects the components of the trace analysis system,

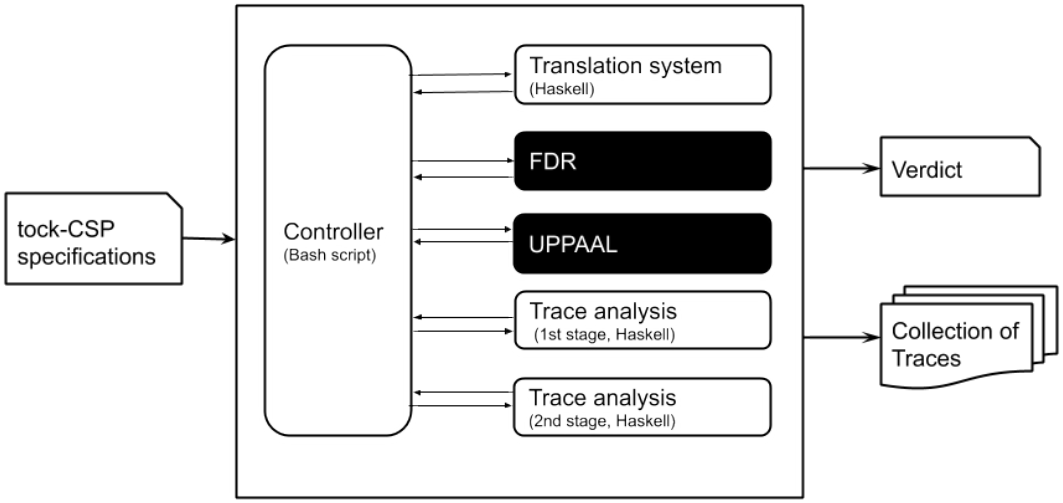


Figure 26: Structure of the trace analysis system

which invokes each component of the system one after the other, passing input and collecting output. The controller passes the required input to a component and collects output and then proceeds to pass the collected output as input to the subsequent component of the system for further analysis. We used two stages of analysing traces. In the first stage, we generate traces from both FDR and UPPAAL. If the traces do not match, we develop the second stage, where we use the power of FDR to complement UPPAAL in analysing the traces.

1st stage of trace analysis We describe the steps of the first stage as follows:

Step 1: The controller takes input specification *tock-CSP*.

Step 2: Invokes the translation system to get the translated TA.

Step 3: Invokes FDR to generate traces of the input *tock-CSP*.

Step 4: Invokes UPPAAL to generate traces of the translated TA.

Step 5: Compares the generated traces (1st stage of the trace analysis).

Step 6: If the generated traces from FDR and UPPAAL do not match, we create a second stage of the trace analysis, where we check if all the generated traces of *tock-CSP* are valid traces of the translated TA.

In the first stage of the traces analysis, we compare the generated traces of *tock-CSP* and TA. However, due to the nature of using TCTL language in formulating queries for

generating the traces, it is impossible to retrieve all the possible traces of TA, especially for the case of TA that captures concurrency. To address this complexity, we develop the second stage of the trace analysis, where we use the power of FDR to complement UPPAAL in generating traces. Thus, we verify all the generated traces of *tock-CSP* are valid traces of the translated TA, which validates the equivalence of the traces. Details of the second stage are as follows. As a running example for the second stage, we consider the following *tock-CSP* process.

```
P1 = e1->((e2->SKIP)|||(e3->SKIP))
```

2nd Stage of the trace analysis: steps of the second stage are as follows:

Step 1: The second stage of the technique begins with appending a special event `mark` to the process `P` to form another process `Pm`. This is express as follows:

```
Pm = P;(mark -> STOP)
```

Note: The selected special event `mark` must not be part of the process `P`.

Step 2: We create an auxiliary process that specifies the required length `n` of a trace. The process is expressed as follows.

```
Sl(n) = if n == 0 then (mark -> SKIP)
        else ([]ev : Events @ ev -> Sl(n-1))
```

The process $Sn(n)$ controls the size of a trace, such that either the process terminates after reaching a trace of size n or the process terminates after performing its last event. When $n == 0$, the process terminates after performing the special event `mark`; otherwise the process proceeds to perform any event from the set `Events` and decreases the value of the parameter `n` by 1, until $n == 0$.

Step 3: We put the process `Sn(n)` in parallel with the constructed process in Step 1. For the running example, the process is as follows:

```
Pmn = (Pm [|Events|] Sl(n))
```

In this case, either the process `Pm` terminates first or the process `Sl(n)` terminates after reaching the required length n of a trace, which forces the concurrent process `Pmn` to a deadlock after reaching the target length n for the required length of the traces.

Step 4: Then, we use FDR to verify an assertion `Pmn` refines `P`, which checks if the process `P` contains the traces of `Pmn`, as follows.

```
assert P [T= Pmn
```

For the running example, using the process `P` with $n > 3$, the assertion fails and yields a counterexample as follows:

```
t1 = <e1, e2, e3, m>
```

The counterexample is one of the complete traces of process `P` with the addition of the special event m. This is because of the formulated process `Pmn` performs the additional special event `mark`, which the process P cannot perform. In the case of a non-terminating process, the function `Sl(n)` forces the process to terminate after reaching the target length `n`. Then, we proceed to find out if the process `P` has another trace different from the generated trace `t1`.

Step 5 We convert the generated trace into a linear process.

For the running example, we convert the trace `t1` into the following linear process:

```
Pt1 = e1->e2->e3->mark->SKIP
```

Step 6: We use external choice to compose original process P with the constructed process Pt1; as `P[]Pt1`. Then, we use FDR again to check if the constructed process `P[]Pt1` contains the traces of the process `Pmn`, as follows.

```
(P[]Pt1) [T= Pmn
```

In the running example, the assertion fails and generates another counterexample, as follows:

```
t2 = <e1, e3, e2, mark>
```

Loop : We repeat Steps 5 and 6 until the assertion passes.

In the case of this running example, the assertion fails. Therefore, we execute the loop again and we repeat Steps 5 and 6.

Repeating Step 5: We convert the new trace `t2` into another linear process.

```
Pt2 = e1->e3->e2->mark->SKIP
```

Repeating Step 6: We use external choice to compose the new process `Pt2` with the previously formulated process `(P[]Pt1)` to form another process `(P[]Pt1[]Pt2)`. Then, we use FDR to verify if the new process `(P[]Pt1[]Pt2)` contains the traces of the process `Pm`, as follows.

```
(P[]Pt1[]Pt2) [T= Pmn
```

After repeating Step 6, the assertion passes, which indicates that the traces `t1` and `t2` are the only traces of length 3 for the input process

```
P1 = e1->(e2->SKIP)|||(e3->SKIP)
```

In generating multiple traces with FDR, like most of the model-checking tools, FDR was developed to produce only one trace (counterexample) at a time. As such, based on the testing technique developed in [52], we develop a technique that repeatedly invokes FDR for generating traces until we get the required traces. We used the developed technique in generating finite length of traces for the input *tock-CSP* specifications using FDR.

However, this technique for generating traces using FDR is not suitable for generating traces using Uppaal. This is due to the differences between the two systems. FDR was developed based on verification using refinement. Secondly, FDR uses the same language CSP for both specifications and verification of a system, while Uppaal was developed based on networks of TA. Also, Uppaal uses different languages for specification and verification; Uppaal uses TA for modelling the specifications of a system and TCTL for specifying the verification requirements.

Thus, we developed another technique for generating traces of TA. The technique we developed is based on a testing technique developed for Uppaal [53]. The basic idea of the testing technique [53, 54] was described using Uppaal as black-box in generating test-cases that achieve test coverage criterion in TA. The testing technique was developed to traverse every edge in TA to achieve edge coverage. The edge coverage criterion was formulated using reachability property with additional auxiliary variables `e1...en` of types Boolean; each path has a unique combination of these variables es. On traversing each path, we assign the value true to all the auxiliary variables in the path, which facilitates formulating another test case for another path that has a different combination of the auxiliary variables. In the end, exhausting all the possible paths provides a set of test cases that achieves test-coverage criterion in a network of TA that models a system [53, 54].

In generating the traces, we attach two auxiliary variables to each action that captures a translation of an event. The two variables are path identification variable and depth value, `epn` and `dp`, respectively, as shown in Figures 28, 29 and 31 (Example 3.1). The three TA capture the occurrence of the actions `e1, e2 and e3`.

Initially, the auxiliary variables of the forms `epx` are initialised to the value `false`, then traversing a path and capturing its trace assigns the value `true` to all the auxiliary variables in that path. While the second auxiliary variable `dp` is initialised to the value 0, traversing each action increases the depth `dp` with the value 1, which increases the length of the trace. This technique enables us to capture traces of TA within the required depth. We illustrate the technique in the following steps.

Example 3.1. Consider a running example of a system that performs an action `e1` and then performs two additional actions `e2` and `e3`, concurrently. We translate the system into a network of TA in Figures 28 – 33.

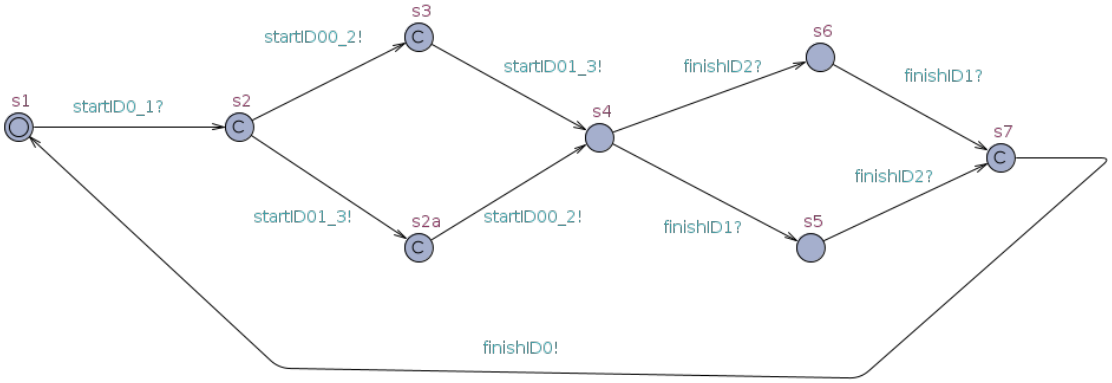


Figure 27: TA1 shows the translation of concurrency operator.

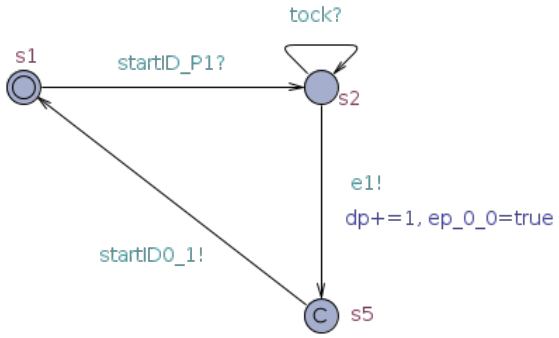


Figure 28: TA2 show the translation of the action e1, with the added auxiliary variables dp and ep_0_0

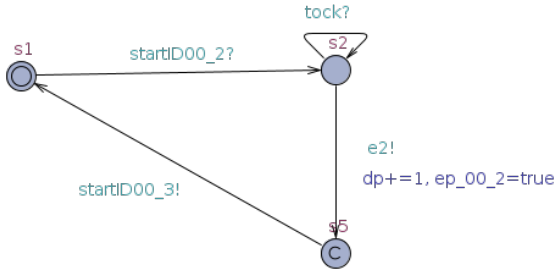


Figure 29: TA3 for translating the action e2 with the auxiliary variables

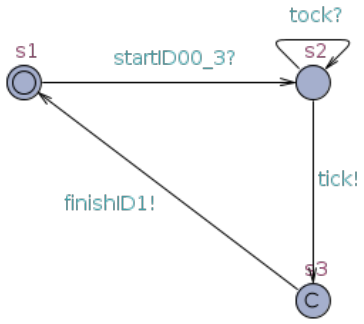


Figure 30: TA4 captures the termination process SKIP

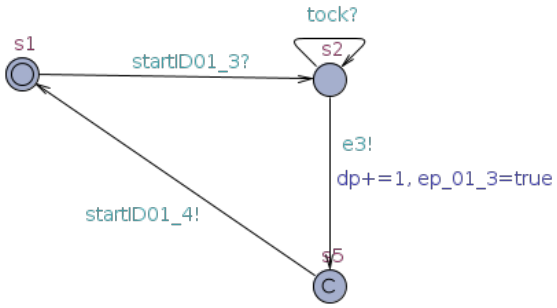


Figure 31: TA5 for translating the action e3 with auxiliary variables

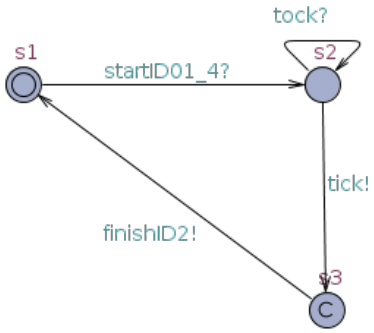


Figure 32: TA6 for translating the termination process SKIP

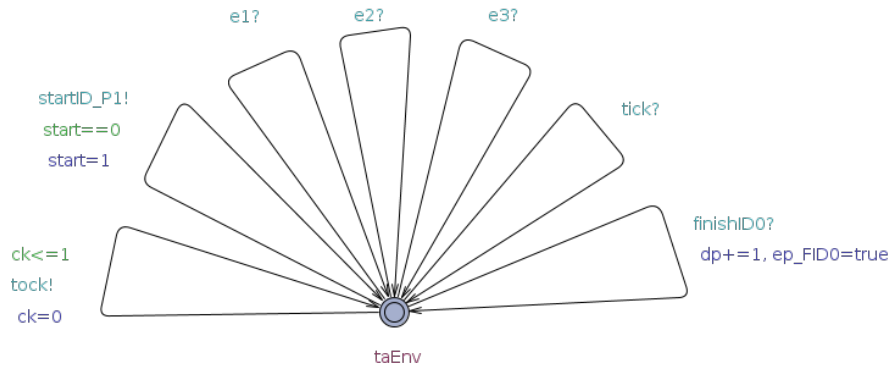


Figure 33: TA7 for an explicit environment of the translated TA

Step 1: We formulate a query for a requirement that the system should reach depth n for a trace of size n, as follows:

```
E<> (dp==n)
```

For the running example, we formulate a query to check that the system should reach a trace of length 3, expressed as follows:

```
E<> (dp==3)
```

Step 2: If the query passes, we get a trace $t1$ for path $p1$ from the initial state to the depth n. The path $p1$ is characterised with a list of the added auxiliary variables `e1...en`, such that all the auxiliary variables in the path are set to the value *true* on traversing the path.

For the running example, the query passes and returns the first trace

```
t1=[e1, e2, e3]
```

Also we characterise the path with a list of auxiliary variables

```
[ep_0_0, ep_00_2, ep_01_3]
```

that are set to true on generating the trace $t1$ in that path.

Step 3: We update the query by blocking the path $p1$, expressed as follows:

```
E<> ((dp == n) and  not p1)
```

Where `p1` is a conjunction of the list of auxiliary variables that identify the path `p1`, as in the case of this running example, we update the query by blocking the path `p1`, as follows:

```
E<> ((dp == 3) and not (ep_0_0 and ep_00_2 and ep_01_3))
```

Loop: We repeat steps 2 – 3 until the query fails.

Repeating Step 2: For the running example, checking for an alternative trace with the second query fails. This indicates that there is no alternative path for generating another trace. If the second query passes, we will get another trace $t2$ for an alternative path $p2$.

NOTE: However, there is another alternative trace `t2 = [e1, e3, e2]` that can be generated on traversing another path `[ep_0_0, ep_01_3, ep_00_2]`. However, it can be seen that the description of the two paths evaluate to the same logical value; even though they have a different order of the actions, which makes it difficult for UPPAAL to detect the second trace [7]. As a result of that, we develop Stage 2 for analysing the traces, which we used in generating traces that are difficult to generate in UPPAAL.

This is the main reason for generating traces in two stages. In stage 1, we generate traces using both TA and *tock-CSP* as well as using both FDR and UPPAAL, respectively. Then, we compare the traces if the traces do not match, we move to Stage 2. Since all the traces of *tock-CSP* have precise order in Stage 2, we use the traces in guiding UPPAAL to check if all the traces produced with FDR are valid traces of the translated TA. Details of Stage 2 are as follows.

We proceed to Stage 2 for checking if all the traces of the input *tock-CSP* are valid traces of the translated TA. We continue using the previous running example for the list of the translated TA in Figures 28 – 33. Previously, we generated the trace $t1 = \langle e1, e2, e3 \rangle$ for the TA model using UPPAAL. Also, in describing the technique for generating traces of *tock-CSP* using FDR, we generated the two traces $\{\langle e1, e2, e3 \rangle, \langle e1, e3, e2 \rangle\}$ for the input *tock-CSP* model.

Trace analysis Stage 2: the steps for the second stage of the trace analysis are as follows:

Step 1: We take the difference df between the two lists of traces.

```
df = (traces of tock-CSP) - (traces of TA)
```

For the running example, the difference between the two traces is:

```
df = [[e1, e2, e3], [e1, e3, e2]] - [[e1, e2, e3]]

∴ df = [[e1, e3, e2]]
```

[7] Perhaps, there might be another way of using the operator or other similar construct. This is left as part of the future investigation.

Step 2: For each trace *ti* in the traces df, we convert the trace *ti* into a linear TA *tiTA*, which has a special final state fs.

For the running example, we convert the trace [e1, e3, e2] into TA in Figure 34.

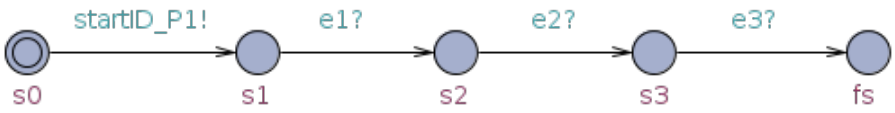


Figure 34: *tiTA* - a linear TA for the trace [e1, e3, e2]

Step 3: In the list of translated TA, we replace the environment TA with the linear TA *tiTA*.

Therefore, for the running example in Figures 28 – 33, we replace the environment TA Figure 33 with *tiTA* Figure 34.

Step 4: We formulate a query to check if the system reaches the final state fs of the trace TA *tiTA* (after replacing the environment TA).

For the running example, the query is as follows:

```
E <> fs
```

Step 5: If the query passes and the system finds a path to the final state fs of the TA *tiTA*, then the trace *ti* is a valid trace of the system. Otherwise, if the query fails, then there is no path to the final state of the *tiTA*. Therefore, the trace *ti* is not an acceptable trace of the translated TA.

In the case of the running example, the query passes. This indicates that the trace *ti* is a valid trace of the system. There is no other trace apart from the trace *ti*, because the difference between the two traces is *ti* only. Therefore, we complete the trace analysis.

This completes the descriptions of the techniques we used for extracting the traces of both *tock-CSP* and TA, using both FDR and UPPAAL. A prototype implementation of the trace analysis tool is available in the repository [47]. The tool has an interface that enables users to interact with the tool. We provide examples inside the prototype system that help users to understand the system. Figure 35 illustrates part of the interface, which displays the available commands for interacting with the tool.

The interface shows a brief description of the available commands. The first command, `load Filename` reads a file that contains *tock-CSP* specifications in AST. The second command `process number` analyses an existing process from the provided processes inside the system. In Figure 35, below the list of commands, there is a table of the provided processes with their corresponding numbers. The numbers are used for invoking each process from the interface. Figure 36 shows an example of analysing process number 20, `p2_0 = (e1->(STOP))|~|(e2->(STOP))`, which illustrates a translation of the construct internal choice. The third command `analyse ASTprocess` analyses a process (specified in AST). The fourth command

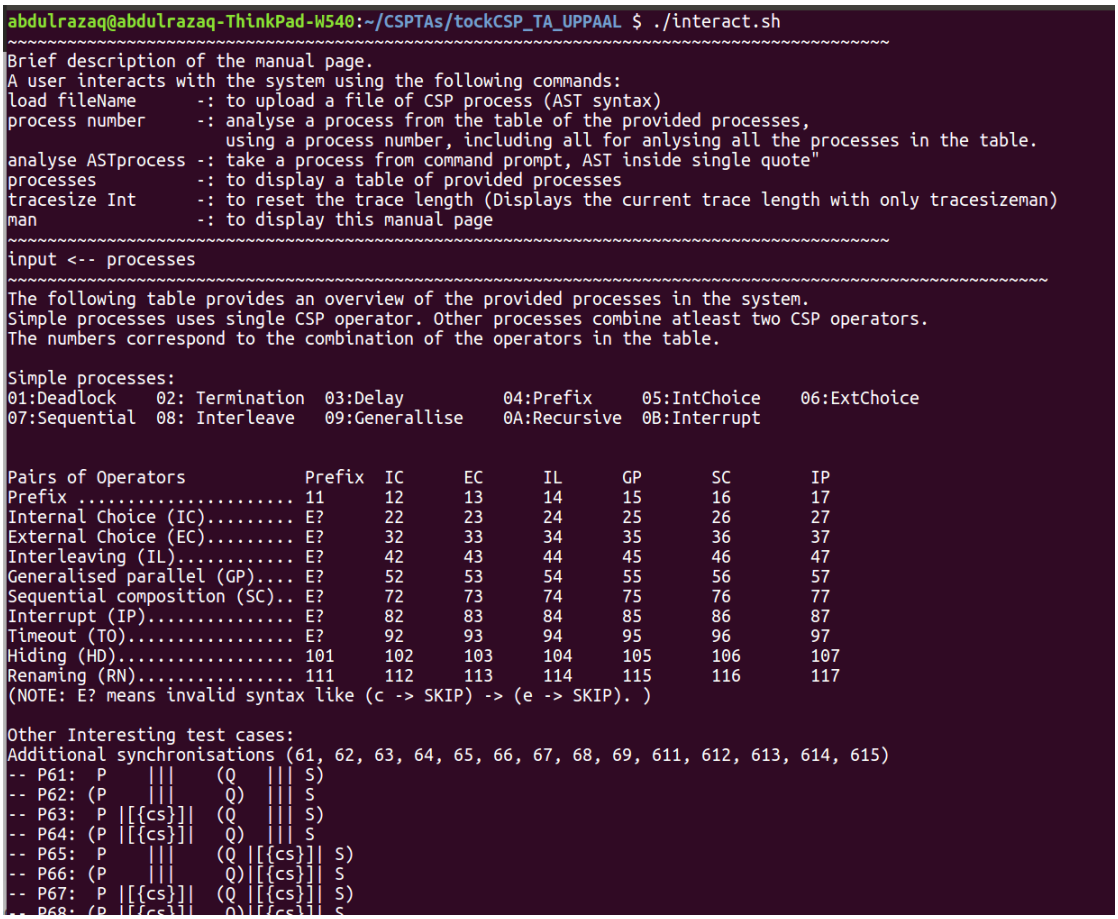


Figure 35: Interface of the trace analysis tool

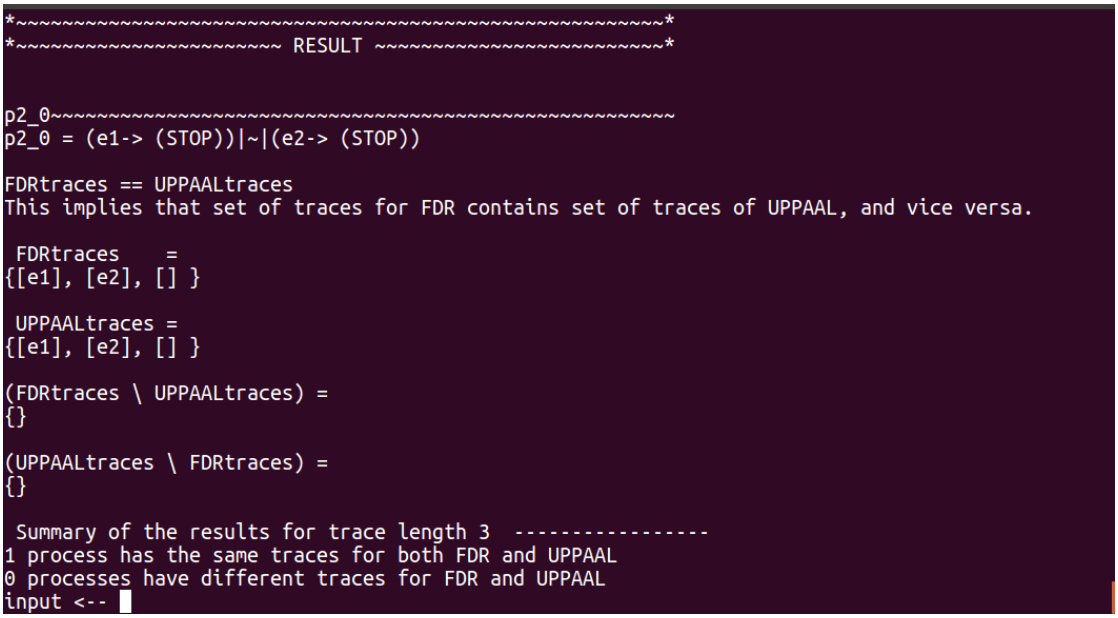


Figure 36: An example of using the trace analysis tool for comparing the traces.

`processes` displays a table of the provided processes in the system, as shown in Figure 35, below the list of commands. The fifth command `trace size` sets the required depth of traces for the analysis. Lastly, the command `man` displays the manual page of the tool.

Figure 36 shows a sample output of the trace analysis system, which displays a summary of comparing the generated traces with both FDR and UPPAAL. Also, the system generates a folder named `genFiles`, which contains files for the details of all the generated traces of both FDR and UPPAAL as well as the details of comparing the generated traces. This concludes the description of the tool we developed for analysing and comparing the traces of the input *tock-CSP* and its corresponding translated TA.

3.3. Experimental Evaluations

This section discusses the list of processes we used in evaluating the translation technique. In evaluating the translation technique, we consider an experimental approach, which enables us to use the trace semantics in justifying the correctness of the translation technique. We begin with formulating a list of processes that covers interesting conditions (test cases), ranging from basic processes to a list of processes that pair all the *tock-CSP* constructors in the provided BNF (Section 2.1).

After formulating the processes, we use the translation tool (see Section 3.1) to translate the *tock-CSP* processes into TA. Then, we used the traces analysis tool (see Section 3.2) to generate and compare the traces of these formulated processes. The results of comparing and analysing the traces enable us to reason about the correctness of the translation technique. For each of the finite processes within a specific length, we con-

sider traces of length 1, 2, 3 and length 10. We select these lengths as samples because we will not be able to cover all the possible lengths of the traces. In all the traces, the traces of the translated TAs model contain the traces of the original input *tock-CSP* process. The result justifies that the translated TA captures the behaviour of the input *tock-CSP* model.

Additionally, these formulated processes are part of the built-in processes provided in the translation tool. These provided processes help users to explore and understand the tool. Each process has a number, and we provide an interface for selecting each process using the provided processes number (additional details in the appendices). On entering a process number, the system displays the selected process, translates the process into TA, then generates and compares the traces of the selected process. Subsequently, the system displays the summarised results for comparing the traces. The list of provided built-in processes and the interface provide a good starting point for interacting and understanding the system.

3.4. Examples - Case Studies

In this section, we discuss a list of case studies that illustrate the application of the translation technique. In selecting the case studies, we consider the wider definition of robot in a wider sense as described by IEEE Standard 1872-2015 [55] From the literature, we consider six cases: a cash machine (ATM) [56], a modified version of the cash machine (ATM2) (Section 3.4.1), an automated barrier to a car park [36] (Section 3.4.2), a railway crossing system [4] (Section 3.4.3), a thermostat machine for monitoring ambient temperature [36] (Section 3.4.4) and a simple mobile system [57] (Section 3.4.5).

First, an overview of these selected case studies is provided in Table 1, while detailed specifications of the system are described in the following Sections. Second, we write *tock-CSP* specifications for the case studies, as listed below for each system. Third, we use the translation tool in translating the *tock-CSP* specifications into TA for Uppaal. Lastly, we use Uppaal for verifying the sample properties listed under the verification requirements that follow each of the case studies.

3.4.1. Cash Machine (ATM)

This example illustrates a translation of a cash machine that goes through cycles of accepting a card, requiring its PIN, and servicing one request before returning the card to the customer. The request can be cash withdrawal, transferring cash and checking a balance of the account. If the PIN is incorrect, the machine returns the card and continues with an operation that prepares the machine for accepting another card [56].

In Listing 2, we present the specification of the system cash machine in *tock-CSP*. Line 1 defines the channels used in describing the system. Line 3 defines the time required for each event. We assign 0 to indicate that each event happens instantly and takes no time to complete. Line 5 describes the beginning of the timed section

Case Studies	Tool	Average Timing (second)	States	Transitions	Events	States per Transitions
Thermostat	FDR	0.0052	7	16	5	0.4375
	UPPAAL	0.0106				
Bookshop Payment	FDR	0.0048	7	32	9	0.21875
	UPPAAL	0.0011				
Simple ATM	FDR	0.0051	15	33	15	0.4545
	UPPAAL	0.0084				
AutoBarrier	FDR	0.0059	35	84	10	0.4167
	UPPAAL	0.0134				
Rail Crossing	FDR	0.0054	80	361	12	0.2216
	UPPAAL	0.0072				

Table 1: Timing of the selected test cases with an overview of their structure.

in a CSP file for FDR verification [40]. Lines 7–8 define the process `ATM` that checks valid PIN. The process accepts the card and PIN and then internally decides whether the PIN is valid and behaves as the process `Options`. For invalid PIN the process returns the card and waits for another card. Lines 9–13 define the behaviour of the process `Options`, which provides three options: `withdrawCash`, `checkBalance` and `transfer`. The first option `withdrawCash` takes an amount of the required money, dispenses the required cash, returns the card and behaves as the process `ATM` (waiting for another card). The second option `checkBalance` displays the account balance, returns the card and behaves as the process `ATM`. Last option `transfer` accepts both an account number and amount for the transfer; it then returns the card and behaves as the process `ATM`. Finally, Line 15 describes an assertion for checking deadlock freedom in the specifications. This completes the specifications of the system cash machine in *tock-CSP*.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL automatically.

1. $A\Diamond$ `returnCard`
   The machine eventually returns the accepted card.

Listing 2: Specifications of Cash Machine (ATM)

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

OneStep(_) = 0

Timed(OneStep){

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

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

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

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

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

In a modified version in Listing 3, we extend the previous version of the cash machine that enables us to formulate additional interesting requirements that we are unable to specified in the previous version of the models. There are two modifications. First, the system retains the card after 5 seconds if the user did not take the card. Second modification, before dispensing the money, the system checks if there is enough balance for the requested amount to withdraw.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL, automatically.

1. $A\diamond$ `retainCard`
   The machine eventually retains the card (expected to fail).

Listing 3: Specifications of Modified Cash Machine (ATM2)

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

OneStep(_) = 0

Timed(OneStep){

ATM = card -> acceptPIN -> (correctPIN ->  Options)
                          |~| (incorrectPIN -> ReturnCard)

Options = (withdrawCash -> Withdrawal)
          [] (checkBalance -> displayBalance -> ReturnCard)
          [] (transfer -> acceptAccountNo -> acceptAmount ->
              ReturnCard)

ReturnCard =  returnCard -> ((takeCard -> ATM) [] (WAIT(5);(
    retainCard -> ATM)))

Withdrawal =  amount ->
             (enoughBalance -> dispenseCash -> returnCard -> ATM)
             |~| (insufficientBalance -> returnCard -> ATM)

assert ATM :[deadlock-free]

}
```

2. `returnCard --> retainCard`
   Returning the card leads to retaining the card (expected to fail, because it holds only if the user does not take the card).

3. `withdrawCash --> dispenseCash`
   withdrawing cash leads dispensing cash (expected to fail, because the requirement holds only if there is enough cash).

3.4.2. Automated Barrier

This example illustrates a translation of an automated barrier that accepts tickets and raises the barrier exactly two time units after dispensing the ticket. The system lowers the barrier one timed unit after receiving a signal through. If the signal is not received after 20 time units of raising the barrier, it emits a beep once per time unit until the system receives a response action either *through* or *reset*. The barrier is lowered one second after the occurrence of either of these events [36].

Listing 4: specification of Automated Barrier

```
1  channel tock, acceptTicket, dispenseTicket, raiseBarrier, beep,
2        through, reset, receivedSignal, lowerBarrier
3
4  OneStep(_) = 0
5  Timed(OneStep) {
6
7  AutoBarrier = acceptTicket -> dispenseTicket -> tock -> tock ->
8             raiseBarrier -> (Response [] (WAIT(20);NoResponse))
9
10 Response = receivedSignal -> tock -> lowerBarrier -> AutoBarrier
11
12 NoResponse = Beeping /\ Action
13
14 Beeping = beep -> tock -> Beeping
15
16 Action = ((through -> SKIP) [] (reset -> SKIP))
17             ;(lowerBarrier -> AutoBarrier)
18
19 assert AutoBarrier :[deadlock-free]
20 }
```

In Listing 4, we present the specifications of Automated Barrier in *tock-CSP* that serves as input to our translation technique. Lines 1–5 are similar to the previous example. Line 7–8 describe the process `AutoBarrier` that accepts and dispenses the ticket, then waits for two times unit before raising the barrier, then behaves as a process `Timeout`. Line 10 defines the process `Timeout(P, Q, d)` that waits d time unit for the process `P` to perform a visible action; after the deadline d, the process `Timeout` behaves as Q. Therefore, the process `Timeout(Response, NoResponse, 20)` waits for 20 time units for the process `Response`. After the deadline elapses, the process behaves as the process `NoResponse`. Line 12 defines the process `Response` that performs the event `receivedSignal` then waits one time unit before lowering the barrier and behaving as the process `AutoBarrier`. The process `NoResponse` performs beeping infinitely (Line 16) until the process is interrupted by the process `Action`. Line 18 defines the process `Action` that presents a choice of either performing the action `through` before terminating or performing the action `reset`, that leads to lowering the barrier and behaving as the process `AutoBarrier`. Finally, Line 21 defines an assertion for checking deadlock freedom in the system. This completes the specification of `AutoBarrier`.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with Uppaal automatically.

1. ($A \diamond$ *raiseBarrier*)
   The system eventually raises the barrier.

2. ($E \diamond$ *lowerBarrier*)
   The system eventually lowers the barrier.

3. `raiseBarrier --> lowerBarrier`
   Whenever the system raises the barrier, eventually the system will lower the barrier.

4. `receivedSignal --> lowerBarrier`
   After receiving a signal, eventually the system will lower the barrier.

5. `acceptTicket --> dispenseTicket`
   After the system accepts the ticket, eventually the system will dispense the accepted ticket.

6. `beep --> lowerBarrier`
   The sound beep leads to lowering the barrier.

7. `beep` $\mathcal{U}$ `(reset` *or* `through)`
   The system continues beeping until the system receives an action either reset or through.

3.4.3. Railway Crossing System

This example illustrates a rail crossing system that consists of three components: a train, a gate, and a gate controller. The gate should be up to allow traffic to pass when no train is approaching but should be lowered to obstruct traffic when a train is close to reaching the crossing. It is the task of the controller to monitor the approach of a train and to instruct the gate to lower within the appropriate time. The train is modelled at a high level of abstraction: the only relevant aspects of the train's behaviour are when the train is near the crossing, when it is entering the crossing, when it is leaving the crossing; and the minimum delays between these events [36,56].

In *tock-CSP*, the input specification of the system is in Listing 5. The gate controller receives two types of signal from the crossing sensors: `nearInd`, which informs the controller that the train is approaching, and `outInd`, which indicates that the train has left the crossing. It sends two types of signal to the crossing gate mechanism: down command and up command, which instruct the gate to go down and up, respectively. It also receives a confirmation from the gate. These five events form the visible events of the controller. The gate, modelled by GATE, responds to the commands sent by the controller. The additional events: up and down are included to model the position of the gate [36,56].

Listing 5: Specifications of the Railway Crossing System

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

OneStep(_) = 0

Timed(OneStep){

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

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

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

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

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

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

assert RailSystem :[deadlock-free]

}
```

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL automatically.

1. $A\diamond$ pass
   The train eventually passes the gate.

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

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

Listing 6: Specifications of Thermostat System

```
1  channel tock, tooHot, tooCold, turnOff, turnOn
2
3  OneStep(_) = 0
4
5  Timed(OneStep){
6
7  Thermostat = (tooHot -> tock -> tock -> turnOff -> Thermostat)
8               [] (tooCold -> tock -> tock -> turnOn -> Thermostat)
9
10 assert Thermostat :[deadlock-free]
11
12 }
```

3.4.4. Thermostat System

This example illustrates a thermostat that monitors ambient temperature and controls a valve to enable or disable a heating system. The system responds to two inputs, `tooHot` and `tooCold`, which is required to perform the events `turnOff` or `turnOn` in response to these two inputs, respectively, after two time units [36].

In Listing 6, we present the *tock-CSP* specification of the system named Thermostat. It serves as input to our translation technique. Line 1–5 are similar to example 1. Line 7 defines the process `Thermostat` that presents two choices. The first choice performs the action `tooHot`, then waits two times before turning off the system, and then behaves as `Thermostat`. The second choice is the event `tooCold` that also waits two times before turning the system off and behaves as the process `Thermostat`.

Verification requirements: after translating the system into TA, the following are sample requirements that can be specified with TCTL and verified with UPPAAL automatically.

1. `tooHot --> turnOff`
   Whenever the temperature is too hot, eventually the system will turn off,
2. `tooCold --> turnOn`
   Whenever the temperature is too cold, eventually the system will turn on
3. `turnOn` $\mathcal{U}$ `tooHot`
   The system remains on until it receives a signal `tooHot`.
4. `turnOff` $\mathcal{U}$ `tooCold`
   The system remains off until it receives a signal for `tooCold`.

Listing 7: Specification of Simple Mobile system

```
1  channel obstacle, tock, moveRet, moveCall, turnCall, turnReturn
2
3  OneStep(_) = 0
4
5  Timed(OneStep) {
6
7  mSystem      = EntryMoving /\ (obstacle-> SKIP);EntryTurning;
8                              WAIT(3);mSystem
9  EntryMoving  = EDeadline(moveCall, 0);EDeadline(moveRet, 0);
10                      WAIT(1);EntryMoving
11 EntryTurning = EDeadline(turnCall, 0); EDeadline(turnReturn, 0)
12
13 EDeadline(e, t) = (e -> SKIP) [|{e}|] (WAIT(t) /\ e->SKIP)
14
15 assert mSystem :[deadlock-free]
16 }
```

3.4.5. A Simple Mobile System

This example illustrates a simple mobile system that moves at a specified linear velocity lv and observes the presence of an obstacle. When the system detects an obstacle, the system turns at a specified angular velocity av and then continues moving, repeating the procedure again [57].

In Listing 7, we present the specification of the simple mobile system. Like the previous examples, Lines 1–5 are similar to the previous test cases. Line 7 defines the process `mSystem` that starts moving until it is interrupted by an obstacle, then turns for three unit time and continues moving by behaving as the process `mSystem`. Lines 10–11 defines the process `EntryMoving` that moves for one time unit repeatedly. Line 13 defines the process `EntryTurning` for turning the system. Line 15 defines the process `Edeadline(e, t)` that specifies a deadline t for the event e. Finally, Line 17 defines an assertion for checking deadlock freedom in the specifications. This completes the specification of the simple mobile system.

Verification requirements: after using our translation technique to translate *tock-CSP* into TA, the following sample requirements are specified with TCTL and verified with UPPAAL automatically.

1. `A<> obstacle`
   The system eventually detects an obstacle [8].
2. `moveCall --> obstacle`

[8] If an obstacle exists in the environment, otherwise the system will continue wondering in the operating environment.

Whenever the system moves, it will eventually detect an obstacle (if an obstacle exists in the environment).

3. `turnCall --> moveCall`
   Whenever the system turns, it will eventually moves and progress again.

4. `moveCall` $\mathcal{U}$ `obstacle`
   The system continues to move until it detects an obstacle (if an obstacle exists in the environment).

Overall, in this section, we presented sample test cases from the literature. We use a selection of six test cases considered in demonstrating the application of the translation strategy and its automated tool. Here, we provide detailed specifications of the test case as well as an overview of the specification in Table 1, which provides insight into the structure of the test cases. Also, we illustrate a selection of requirements that can be verified with TCTL, using Uppaal on the translated TA for the input *tock-CSP* specifications. As described in details in both Chapter 1 and 2, these selected requirements illustrate sample requirements that can be verified after translating the specifications.

These test cases help us in understanding both the strength and limitations of our translation technique. Also, these test cases helped us in improving and fixing errors associated with translating recursion.

3.5. Performance Evaluations

In this section, we illustrate one of the applications of our translation technique in comparing the performance of the two verification tools: FDR and Uppaal, specifically for automatic verification. We describe the resources as well as the procedure used for the performance evaluation. Previously in Section 3.3, we discussed a translation of a list of processes that enables us to compare the performance of FDR and Uppaalin this section. Evaluating the performance is another experiment that we carry out in taking advantage of the translation work.

In Section 3.3, we developed an evaluation tool that enables us to evaluate the translation technique by translating a list of formulated processes, which pairs all the considered constructors in the presented BNF of this research. We use the collection of these processes in comparing the performance of the two model-checkers: FDR and Uppaal in analysing deadlock freedom. We consider checking deadlock freedom using both FDR and Uppaal on the formulated processes constructed manually and the translated TA for Uppaal. Details of the processes are available in a provided repository of the work [47]. Each verification was repeated ten times. Details of the timings are available in the repository [47]. A summary of the average timing is presented in Figure 37, which provides an overview of comparing the performance of the tools: FDR and Uppaal.

Evaluating the performance is another experiment that we carry out for comparing the two model checkers. The graph in Figure 37 summarises the recorded timing

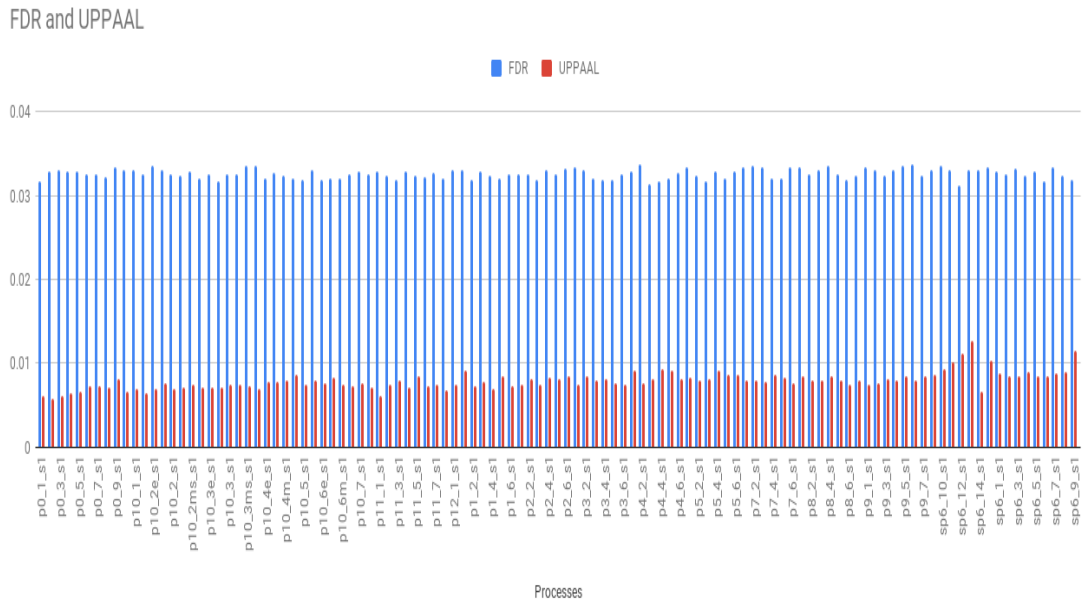


Figure 37: Performance analysis for comparing the performance of FDR and Uppaal in checking deadlock freedom (time unit in seconds).

for checking deadlock using both FDR and Uppaal, Checking deadlock is one of the common interesting checks that can be done with both FDR and Uppaal. Secondly, deadlock checks require an exhaustive verification for checking every state of each model, which makes it suitable for comparing performance.

For each process in the list of the processes we considered for the evaluation, we record the times of checking deadlock freedom using both FDR and Uppaal. Also, we record the average timing after repeating each of the checks ten times. We compare the recorded timings in Figure 37, which summarises the recorded average timing. The longer bars (blue) show the recorded timings for FDR, while shorter bars (red) show the recorded average timings for Uppaal. From the graph, it is clear that the average performance timings for FDR are above 0.03s, while the average performance timings for Uppaal is below 0.01s. This first part of the result shows that Uppaal is faster than FDR for verifying deadlock freedom.

However, with larger processes, FDR performs better than Uppaal, as shown in Table 1. The table shows the recorded average timing for each of the processes in the listed case studies (see Section 3.4). The remaining parameters provide an overview of the processes, which indicates the sizes and structures of the processes. This concludes the experiment carried out in comparing the performance of FDR and Uppaal. In the next section, we describe our plan of mathematical proof for justifying the translation rules.

3.6. An Overview of Mathematical Proofs

This section discusses our plan for using mathematical proofs in justifying the translation technique. So far, we use traces for the experimental evaluation of the translation technique. However, this is an approximation to establishing correctness with a finite set of traces. Proving correctness for the complete set of traces involves using mathematical proof. This is achieved using structural induction. An account of our initial effort to produce a proof is provided in this section. Here, we illustrate an early part of the proof with a proof of one of the base cases of the structural induction: a translation of the basic process *STOP*. In Appendix E, we provide additional details of the proof, which includes the base case *SKIP*, and the induction steps using the construct of Internal choice, External choice and Sequential composition.

For the proof of our translation function, we need to establish that, for each valid *tock-CSP* process `CSPproc`, written using the terms of the provided BNF (Section 2.1, Page 11), the following property holds.

```
1     forall P::CSPproc,
2         (traces_tockCSP P) = (traces_TA . transTA P)
```

Therefore, for each translation rule (Section 2.3), we need to show that the translated TA captures the behaviour of its translated construct of *tock-CSP* in the BNF (Section 2.3). In carrying out the proof we develop functions and tool that help us in developing and evaluating the proof. We use the tool in evaluating the steps of the proof to ensure that the steps are correct and consistence. The three major functions use in the proof are `trace'TA`, `traces_TA` and `traces_tockCSP`. Additional detail of these functions is available in Appendix E. The first function `traces'TA` takes a list of translated TA and computes their traces:

```
traces'TA :: [TA] -> [Traces]
```

The second function `traces_TA` takes a list of TA and returns its traces without the flow actions, define as follows:

```
traces_TA :: [TA] -> [Traces]
```

Lastly, the function `traces_tockCSP` takes a *tock-CSP* process and returns its traces, as defined by Roscoe [4].

```
traces_tockCSP :: CSPProc -> [Traces]
```

For example, let us consider `TA1` as the TA for the translation of *STOP* using Rule 2.1 (Page 30). `TA1` is define as follows:

```
TA1 =  [
```


```
          ]

  TA1 = [([s0, s1], s0,  [ck],
          [(startID ++ bid ++ _ ++ sid), tock],
          [(s0, (startID ++ bid ++ _ ++ sid), [], [], s1),
           (s1, tock, ck<=1, ck, s1)], [])
         ]
```

In the context of TA, a path [42,58] is a sequence of consecutive transitions that begins from the initial state, which may be an empty sequence. And a trace [42,58] (or word in the language of TA) is a sequence of actions in a given path. There is only one infinite path in TA1 with two transitions: first, a transition from location $s0$ to location $s1$; second, a transition from location $s1$ and back to the same location $s1$. The traces on the path of TA1 are expressed as follows:

```
traces'TA TA1 =
 []:[ "startID00":s |s <- [ replicate n "tock" | n <- [0..]]]
```

The function `traces'TA` computes the traces of TA1 as follows. The first empty sequence appears before the first transition. The action `startID00` happens on the first transition. The action `tock` happens on the second transition, which is repeated infinitely to produce the infinite traces on the path $\langle tock \rangle^n$.

The second function `traces_TA TA` is similar to `traces'TA TA` but removes all the coordinating actions (Definition 2.2) from the traces. In the case of `TA1` the traces are as follows:

```
tracesTA TA1 = []:[(replicate n "tock") | n <- [0..] ]
```

In the proof we use structural induction over the natural numbers.

```
1 forall n:Nat,
2     forall P::Proc,
3         traces_tockCSP n P = traces_TA n . transTA P
```

For the base case of the natural numbers ($n = 0$), we need to show that:

```
1  for n=0,
2     forall P::Proc,
3         traces_tockCSP 0 P = traces_TA 0 (transTA P)
```

This is illustrated as follows:

```
1 proofTrans :: CSPproc -> Int -> ProofLayout [[Event]]
2 proofTrans    p          0   =
3     traces_tockCSP 0 p
4     :=: --{traces_tockCSP.0,  l~>r}
5     [[]]
6     :=: --{traces_TA.0, r~>l}
7     traces_TA 0 (transTA p  ""  0  0  0 [])
8     :=: QED
```

In ensuring that the steps of the proof are correct, we used a simplistic tool *pcpl* to support the checking of the proof. Additional detail of the tool is also included in the appendix. The tool checks the syntactic correctness and type correctness of the formulae to ensure that the steps are consistent with one another. An illustration of using the tool *pcpl* can be seen above, in the proof of the base case for integer $n = 0$.

In using the tool we describe the proof as a function that takes CSP process and an integer as universally quantified variable used in the proof layout for describing the proof. The tool `pcpl` follows the layout to check the proof. The tool has a special symbol `:=:` that separates the steps of the proof (similar to the commonly used symbol =), and Haskell comments in the form of `--{}` to indicate reasons for the steps. For example, in the above part of the proof, the steps of the proof are in Lines 3, 5 and 7, separated by the symbol `:=:`. Lines 4 and 6 are comments for the justification of the steps enclosed inside the comment symbol `--{}`. Each reason is structured to provide a justification (mostly an identifier of a function); and direction of the application of the function: right (r) to left (l) or left to right, as `l<~r` and `l~>r`, respectively; and also a mapping of the substituted value(s) which we did not use in this illustration. Line 4 describes the reasons as the application of the function `traces_tockCSP` from left to right, while Line 6 describes a reason as the application of the function `traces_TA` from right to left. For the proof of this base case ($n = 0$), since we have used no properties of `P`, other than `P::CSPproc`, we can conclude, by the rule of generalisation:

```
1     forall P::Proc,
2         traces_tockCSP 0 P = traces_TA 0 (transTA P)
```

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

We start with the first rule (translation the construct of a basic process *STOP*; the first construct of the BNF (Section 1). For the proof by induction, we need to establish that:

```
1          traces_tockCSP STOP = traces_TA (transTA STOP)
```

We need to show that

```
1   (traces_tockCSP n STOP = traces_TA n (transTA STOP))
2     => (traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP))
```

This is illustrated as follows:

```
1 -- (traces_tockCSP n STOP = traces_TA n (transTA STOP))
```

```
--   => (traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP))
proofTrans    STOP        n   =
    traces_tockCSP (n+1) STOP
    :=: --{traces_tockCSP.stop, n<~n+1}
    [(replicate l "tock") | l <- [0..n+1]]
    :=: --{List comprehensions }
    [(replicate l "tock") | l <- [0..n]] ++ [replicate (n+1) "tock"]
    :=: --{traces_tockCSP.stop}
    (traces_tockCSP n STOP) ++ [replicate (n+1) "tock"]
    :=: --{Induction hypothesis}
    nub ((traces_TA n (transTA STOP "ta" 0 0 0 []))
        ++ [replicate (n+1) "tock"])
    :=: --{transTA.stop, l~>r}
    nub (traces_TA n taSTOP
         ++ [replicate (n+1) "tock"])
    :=: --{tracesTA.n, l~>r, tas<~STOP}
    nub ([t \\ ["startID0_0"] |
           t <- (traces'TA n taSTOP ) ]
         ++ [replicate (n+1) "tock"] )
    :=: --{traces'TA.stop, l~>r}
    [t \\ flowActions taSTOP
            |t <- ([("startID0_0"):s
                |s <- [(replicate l "tock")|l <- [0..n] ] ] ) ]
    ++ [replicate (n+1) "tock"]
    :=: --{Introducing the connection action}
    [t \\ flowActions taSTOP
        | t <- ([("startID0_0"):s
            |s <- [(replicate l "tock")|l <- [0..n] ] ])  ]
    ++ [ t \\ ["startID0_0"]
            | t <- [["startID0_0"] ++ (replicate (n+1) "tock")] ]
    :=: --{List comprehensions}
    [t \\ flowActions taSTOP
            | t <- ([("startID0_0"):s
                |s <- [(replicate l "tock")|l <- [0..n+1] ] ] ) ]
    :=: --{traces'TA.stop, r~>l}
    nub ([t \\ flowActions taSTOP
           | t <- (traces'TA (n+1) taSTOP ) ] )
    :=: --{tracesTA.n, r~>l, n<~n+1, tas<~taSTOP}
    nub (traces_TA  (n+1)  taSTOP )
    :=: --{trans.stop, r->l}
    nub (traces_TA   (n+1)  (transTA STOP "ta" 0 0 0 []))
    :=: QED
        where
           taSTOP = [([ "s0", "s1"], "s0",  ["ck"], ["startID0_0"], [
               "tock"], [("s0", "startID0_0", [], [], "s1"), ("s1", "
               tock", "ck<=1", "ck", "s1")], [] )]

    ∴ traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP)
```

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

In this section, we provide an overview of the proof with illustration of the proof for the base case. Additional details of the proof is included in the appendix.

3.7. Final Considerations

In this chapter, we described the evaluation approaches we considered in evaluating the translation technique. We discussed the implementation of the translation rules into a tool that automates the translation technique. The tool illustrates how the rules fit together as a system that automates the translation technique. This gives us confidence in using the translation rules as part of the translation strategy.

We used two types of test-cases for the evaluation. First, a collection of small processes that pair all the constructs of the *tock-CSP* within the scope of this work. The small processes have traces of sizes within the range 0–10, except recursive processes that have unbounded traces. The second collection of test cases is processes of reasonable size systems for evaluating the translation technique. However, their traces are infinite due to the nature of the timed system that must allow the progress of time.

We performed the experiment on Ubuntu 16.04.6 LTS (Xenial Xerus) running on Intel Core i7-4700MQ (4C/8T, 2.4GHz) processor with 32GB RAM (4x 8GB modules). This gives us a good opportunity for comparing the performance of FDR and Uppaal, in analysing deadlock freedom. It would be interesting to see a general comparison of the tools, but that is a topic for future work. In this work, we take an average of ten running for each process. The result shows that Uppaal performs better on small processes. However, with more substantial processes, FDR performs better than Uppaal in comparing the performance experiment on the larger test-cases, as described in this chapter.

Additionally, we have described an initial mathematical proof for establishing the correctness of the translation rules, given that mathematical proofs provide greater certainty than trace analysis. Thus, we plan to use structural induction to prove the translation rules. The mathematical proof will give higher assurance for justifying the translation rules.

These two validation considerations serve to validate the correctness of the translation technique, and shows that our strategy is a promising basis for a translating *tock-CSP* into TA. In the next chapter, we evaluate the presented work, present conclusion, and discuss future work.

Chapter Five

4. Summary and Conclusion

We conclude by summarising the research work conducted in this study, in Section 4.1. Then, in Section 4.2, we discuss the limitations of the work and recommend future directions for improving the work.

4.1. Summary

The most fundamental question emerged from early exploratory work was how to improve formal techniques of verifying temporal specifications that are compatible with the existing resources of developing robotics, especially the ones that have a formal basis (Chapter 1). In investigating this question, we studied the available formal techniques for verifying temporal specifications. We found that various techniques and tools have been developed for supporting the advancement of robotics software. However, there is less attention given to verifying the correctness of the robotics applications; for instance, few techniques have been developed to support the verification of the robotics software.

In the literature, we studied the existing technique and tools available in the area of software engineering that focus on three directions: Formal Method, DSML and Temporal specifications. We found that many DSMLs have been developed to improve the advancement of robotics software systems. However, most of the techniques lack a formal basis, and there is little proper support for verification, specifically in verifying temporal specifications.

In the area of formal method, *tock-CSP* has been developed for modelling temporal specifications with the support of FDR for automatic verification. Among the studied DSMLs, RoboChart uses *tock-CSP* for verifying temporal specifications. Also, we find that there are existing projects and tools that provide facilities for verifying temporal specification in a continuous-time model, particularly in the formal verification of real-time systems. For instance, Uppaal, KRONOS and PRISM are popular real-time verification tools. From the literature, Uppaal is the most advanced and most efficient tool that supports the verification of TA with TCTL.

In Chapter 2, we presented a proposed technique for translating *tock-CSP* into TA for Uppaal, which facilitates using temporal logic, and automatic support with Uppaal in verifying *tock-CSP* models. This translation technique provides a way of using TCTL in specifying liveness requirements that are difficult to specify and verify in *tock-CSP*. Also, the result of this work sheds additional insight into the complex relationship between *tock-CSP* and TA, as well as the connection between the refinement model and temporal logic model.

We extended *tock-CSP* with the additional support of real-time model-checker Uppaal for verifying temporal specifications, especially for the kind of requirements that are difficult to verify with FDR, such as liveness specifications. We developed a translation

technique for translating *tock-CSP* into TA that facilitates using UPPAAL for automatic verification of *tock-CSP*, which improves the facilities of verifying temporal specification in RoboChart and other existing works that are based on *tock-CSP*.

In developing the translation technique, we provided a BNF that serves as a link for connecting existing works in *tock-CSP* with our translation technique. We developed a translation strategy for the translation technique. Based on the strategy, we came up with translation rules that describe the translation of each construct of the *tock-CSP* into TA, within the scope of our work (Chapter 2).

We considered translating *tock-CSP* models into a network of small TAs that meets the specifications of *tock-CSP*. This is due to the compositional characteristics of *tock-CSP* (inherited from CSP) which is not available in TA, and it is challenging translating *tock-CSP* into a single TA. With our approach of using a network of small TAs, we captured the specifications of the *tock-CSP* into TA, which facilitates using TCTL for the verification of *tock-CSP* models.

In justifying the correctness of the translation strategy (Chapter 3), we used trace analysis to compare the behaviour of the input *tock-CSP* with the behaviour of the translated TA. We generated the traces in two stages. In stage 1, we generated traces from both FDR and UPPAAL; if the traces did not match, we created stage 2 of the trace analysis. In stage 2, we used UPPAAL to verify that all the traces of *tock-CSP* are valid traces of the translated TA. In this case, we used the power of FDR to complement the power of UPPAAL in generating traces. This is because UPPAAL is not capable of generating traces that evaluate to the same logical value irrespective of the permutation of the events.

In evaluating the translation technique, we used two categories of test cases. First, we used a collection of formulated processes for pairing all the essential *tock-CSP* constructs, within the scope of this translation work. Second, a selection of test cases from the literature were used that helped us to understand the strengths and limitations of the translation work. The selection of the test cases is in the wider sense as defined from the IEEE Standard 1872-2015. The collection of these processes gives us the opportunity to compare the two model checkers: FDR and UPPAAL in analysing deadlock freedom where the input *tock-CSP* process is constructed manually and the translated TA generated by our translation tool. Because of that, the results may not be a fair comparison, as the generated TA may not be the most efficient models for UPPAAL. The result shows that FDR performs better in the larger processes, while UPPAAL performs better in the smaller processes (Chapter 3). However, the boundary between the range of the processes remains unknown; an interesting investigation for future work.

Considering that trace analysis is limited to bounded traces, which is limited to covering safety properties, it is clear that trace analysis covers a partial notion of correctness that will not be able to cover infinite traces. Thus, we planned to extend the trace analysis by using mathematical proofs to establish the correctness of the translation techniques that will cover infinite traces. This was achieved with structural induction. We provided an illustration of the proof for the base cases and an induction step. Completing the proof will provide an additional convincing justification for the technique and its supporting tool. This concludes our contribution for this research

work.

4.2. Future work

In this section, we discuss the limitations that are beyond the scope of our work. Also, we provide insight into further work that can improve this work in a promising direction of the future work.

So far, in this work, we have used trace analysis to justify the correctness of the translation work. We provided a good plan for using structural induction to prove the correctness of the translation rules. Completing the proof is beyond the scope of this work. An immediate recommended future work is to complete the proof for the correctness of the translation technique. Besides, the scope of this work is limited by trace analysis. It will be interesting if future work will consider expanding the scope of the proof to include refusal and divergences.

Unifying Theories of Programming (UTP) provides a unified framework for comparing semantics of programs that facilitates reasoning about programming theories as well as links between programmes and theories. To increase the strength of the translation work, it will be interesting to see an expansion of this work by establishing the correctness of the translation technique via UTP. For instance, by building a semantic link between *tock-CSP* and TA using operational semantics of both sides.

The provided BNF (Chapter 3) covers the essential operators of *tock-CSP*. A natural extension of this work is to expand the BNF to cover all the operators of *tock-CSP*. This will improve the application of the translation technique to translate a broader category of systems.

In this work, we considered a network of small TAs to develop the translation technique due to the problem of handling the compositional structure of *tock-CSP*, as mentioned in Chapter 3. It will be interesting to explore translating the *tock-CSP* into a large size TA, instead of the small sizes. This has the potential of improving the performance of Uppaal for automatic verification of the translated TA models.

Also, using the translation technique to translate extensive case studies will help to further understand the strengths and robustness of our translation work, as well as additional scope that will limit the application of the translation technique. For instance, in translating the case studies (Section 3.4), we understand the role of translating data which makes it necessary to omit additional interesting case studies that we may like to include, such as Triple Redundancy Protocol, Grid system and Conway puzzle. Expanding the work to include translating data will be a good addition to this work.

Additionally, currently, we translated the event `tock` into an action that is controlled by a timed clock in Uppaal. A recommended next step in this work is to relate the notion of `tock` to the notion of time in TA and getting rid of *tock* as an action; such that we can be able to translate a process like `P=tock->tock->tock-SKIP` (or `WAIT(3)`) into a single TA, like in Figure 38, which has clock `ck>3` that captures the time. This additional extension will help us to explore additional interesting facilities of Uppaal for verifying temporal specifications.

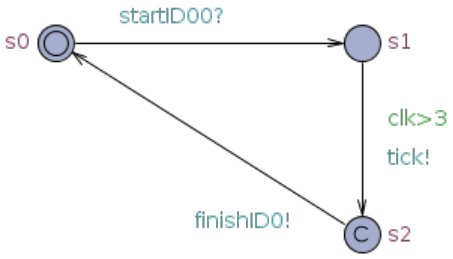


Figure 38: A sketch of a sample translated TA that illustrates a translation of the process $WAIT(3)$ with clock $clk > 3$.

Adding a GUI to the translation tool will be a significant improvement to the tool. For instance, it will be interesting to see users interacting with the tool through the GUI. Additionally, incorporating feedback from the results of the analysis directly to the GUI will make it easier to understand the analysis.

This translation work opens up other directions for future work. For instance, translating the refinement specifications into TCTL will be a good addition to this work. This will make it easy to communicate results between FDR and Uppaal.

Another interesting extension of this work is calculating the complexity of the translation technique and the generated networks of timed automata. This will be useful for further work in improving the efficiency of both the translation technique and the translated TA. Although this may not be simple work because many factors need to be considered in describing the complexity, apart from the size of the input process, other factors include a compositional structure of the input process, a combination of operators used in describing the input process, the structure of the input process (such as recursive, deadlock, and termination) and also both concurrency and the number of synchronisation points in the process.

Another interesting work is to expand the performance experiment (Section 3.5) to find out additional details for answering the pending question of why Uppaal performs better in smaller processes and FDR performs better in larger processes. It will be interesting to know the clear boundary between the performance of the two tools in the future extension of this work.

It would be interesting to see how our translation work will match the evolution of RoboChart models. At the time conducting this work, RoboChart was still evolving. As such, we focus on *tock-CSP* that captures the generated semantics of the RoboChart models. Part of the future work should investigate the application of this work when a stable version of RoboChart become available.

In addition, it will be interesting to include other tools such that we can understand the strengths of these tools and utilise them appropriately. There are still many other aspects that can be further explored in this research direction. These include consideration of translating *tock-CSP* into KRONOS, which also provides a verification engine for verifying TA in modelling and verification of system specifications. It has a different approach from Uppaal. This completes the work we carried out in this research.

Appendices

A. List of Small Processes for the Experimental Evaluation

This section describes an overview of the list of processes we use for the experimental evaluation. Table 2 describes the constant processes, while Table 3 describes a list for processes for pairing the constructs we consider in this work. The processes in Table 3 illustrates an exhaustive pairing of the constructs with one another.

01	STOP	Deadlock (Dl)
02	Stopu	Urgent Deadlock (UD)
03	SKIP	Termination (Tn)
04	Skipu	Urgent Termination (UT)
05	WAIT(n)	Delay (Dy)
06	Waitu(n)	Strict Delay (SD)

Table 2: Basic processes for the experimental evaluation

BNF Constructors	Dl	UD	Tn	UT	Dy	SD	Px	IC	EC	Il	GP	SC	RP	To	It	ED	Hd	Rn	Ex
Prefix (Px)	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	1G	1H	1J
Internal Choice (IC)	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	2G	2H	2J
External Choice (EC)	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	3G	3H	3J
Interleaving (Il)	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	4G	4H	4J
Generallise Parallel (GP)	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	5G	5H	5J
Sequential (SC)	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	6G	6H	6J
Recursive Process (RP)	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	7G	7H	7J
Interrupt (It)	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F	8G	8H	8J
Timeout (To)	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F	9G	9H	9J
Event Deadline (ED)	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF	AG	AH	AJ
Hiding (Hd)	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	BG	BH	BJ
Renaming (Rn)	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF	CG	CH	CJ
Exception (Ex)	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF	DG	DH	DJ

Table 3: An overview of the small processes for pairing the constructors used in the experimental evaluation.

B. Abstract Syntax Tree of UPPAAL TA

Source [9]

```
BNF for the 4.x XTA format
            XTA ::= <Declaration>* <Instantiation>* <System>
    Declaration ::= <FunctionDecl> | <VariableDecl> | <TypeDecl> | <
        ProcDecl>
  Instantiation ::= ID ASSIGNMENT ID '(' <ArgList> ')' ';'
         System ::= 'system' ID (',' ID)* ';'

  ParameterList ::= '(' [ <Parameter> ( ',' <Parameter> )* ] ')'
      Parameter ::= <Type> [ '&' ] ID <ArrayDecl>*

   FunctionDecl ::= <Type> ID <ParameterList> <Block>

       ProcDecl ::= 'process' ID <ParameterList> '{' <ProcBody> '}'
       ProcBody ::= (<FunctionDecl> | <VariableDecl> | <TypeDecl>)*
                    <States> [<Commit>] [<Urgent>] <Init> [<
                       Transitions>]

         States ::= 'state' <StateDecl> (',' <StateDecl>)* ';'
      StateDecl ::= ID [ '{' <Expression> '}' ]

         Commit ::= 'commit' StateList ';'
         Urgent ::= 'urgent' StateList ';'
      StateList ::= ID (',' ID)*

           Init ::= 'init' ID ';'

    Transitions ::= 'trans' <Transition> (',' <TransitionOpt>)* ';'
     Transition ::= ID '->' ID <TransitionBody>
  TransitionOpt ::= Transition | '->' ID <TransitionBody>
 TransitionBody ::= '{' [<Guard>] [<Sync>] [<Assign>] '}'

          Guard ::= 'guard' <Expression> ';'
           Sync ::= 'sync' <Expression> ('!' | '?') ';'
         Assign ::= 'assign' <ExprList> ';'

       TypeDecl ::= 'typedef' <Type> <TypeIdList> (',' <TypeIdList>)*
            ';'
     TypeIdList ::= ID <ArrayDecl>*
BNF for variable declarations
   VariableDecl ::= <Type> <DeclId> (',' <DeclId>)* ';'
         DeclId ::= ID <ArrayDecl>* [ ASSIGNMENT <Initialiser> ]
    Initialiser ::= <Expression>
                 |  '{' <FieldInit> ( ',' <FieldInit> )* '}'
```

[9] Taken from UPPAAL documentation in this link `http://people.cs.aau.dk/~adavid/utap/syntax.html`

```
      FieldInit ::= [ ID ':' ] <Initialiser>

      ArrayDecl ::= '[' <Expression> ']'

           Type ::= <Prefix> ID [ <Range> ]
                 |  <Prefix> 'struct' '{' <FieldDecl>+ '}'
      FieldDecl ::= <Type> <FieldDeclId> (',' <FieldDeclId>)* ';'
    FieldDeclId ::= ID <ArrayDecl>*

         Prefix ::= ( [ 'urgent' ] [ 'broadcast' ] | ['const'] )
          Range ::= '[' <Expression> ',' <Expression> ']'


-- BNF for statements
    Block ::= '{' ( <VariableDecl> | <TypeDecl> )* <Statement>* '}'
Statement ::= <Block>
           |  ';'
           |  <Expression> ';'
           |  'for' '(' <ExprList> ';' <ExprList> ';'
                    <ExprList> ')' <Statement>
           |  'while' '(' <ExprList> ')' <Statement>
           |  'do' <Statement> 'while' '(' <ExprList> ')' ';'
           |  'if' '(' <ExprList> ')' <Statement>
                    [ 'else' <Statement> ]
           |  'break' ';'
           |  'continue' ';'
           |  'switch' '(' <ExprList> ')' '{' <Case>+ '}'
           |  'return' ';'
           |  'return' <Expression> ';'

Case      ::= 'case' <Expression> ':' <Statement>*
           |  'default' ':' <Statement>*

-- BNF for expressions
   ExprList ::= <Expression> ( ',' <Expression> )*
 Expression ::= ID
            |   NAT
            |   'true' | 'false'
            |   ID '(' <ArgList> ')'
            |   <Expression> '[' <Expression> ']'
            |   '(' <Expression> ')'
            |   <Expression> '++' | '++' <Expression>
            |   <Expression> '--' | '--' <Expression>
            |   <Expression> <AssignOp> <Expression>
            |   <UnaryOp> <Expression>
            |   <Expression> <Rel> <Expression>
            |   <Expression> <BinIntOp> <Expression>
            |   <Expression> <BinBoolOp> <Expression>
            |   <Expression> '?' <Expression> ':' <Expression>
```

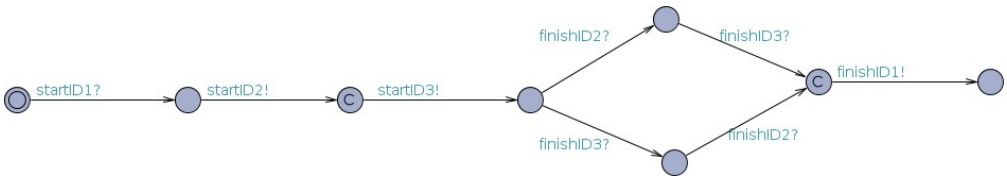
```
90            |   <Expression> '.' ID>
91
92    AssignOp ::= ASSIGNMENT | '+=' | '-=' | '*=' | '/=' | '%='
93             | '|=' | '&=' | '^=' | '<<=' | '>>='
94     UnaryOp ::= '-' | '!'
95         Rel ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
96    BinIntOp ::= '+' | '-' | '*' | '/' | '%' | '&' | '|' | '^' | '<<'
       | '>>'
97   BinBoolOp ::= '&&' | '||'
98     ArgList ::= [ <Expression> ( ',' <Expression> )* ]
```

C. Additional Examples

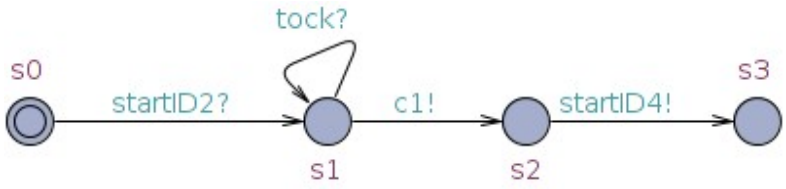
Additional examples are provided here to illustrate possible synchronisation patterns of the generallised parallel operator. The process P1, Q1 and R1 can be arbitrary processes. But for the purpose of illustration, we consider simple definition for each of these processes P1, Q1 and R1 as follows:
P1 = `c1->cs->SKIP`
Q1 = `c2->cs->SKIP`
R1 = `c3->cs->SKIP`

Example C.1. $P1[|\{cs\}|](Q1[|\emptyset|]R1)$

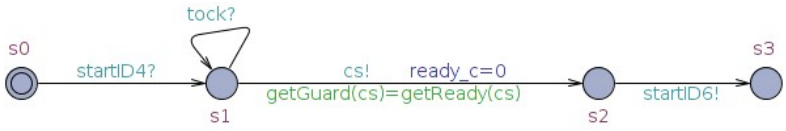
```
transform(P[|{cs}|](Q1[|∅|]R1)))
 = transTA(P1[|{cs}|](Q1[|∅|]R1)), startID1, finishID1, ∅, ∅, ∅,)
 = [
```



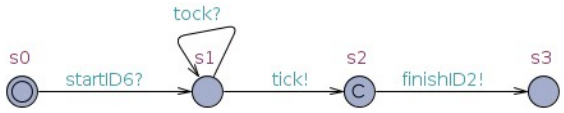

```
        ] ⌒ transTA(P1, startID2, finishID2, ∅, {cs} ∅)
        ⌒ transTA((Q1[|∅|]R1), startID3, finishID3, {cs}, ∅, ∅)

transTA(P1, startID2, finishID2, ∅, {cs} ∅)
 = transTA(c1->cs->SKIP, startID2, finishID2, ∅, {cs} ∅)
 = [
```



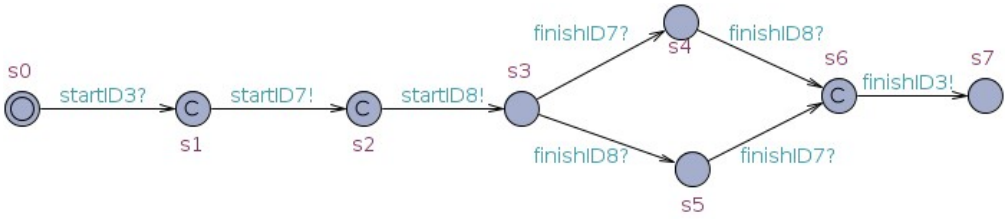

```
    ] ⌒ transTA(cs->SKIP, startID4, finishID2, ∅, {cs} ∅)
       = [
```



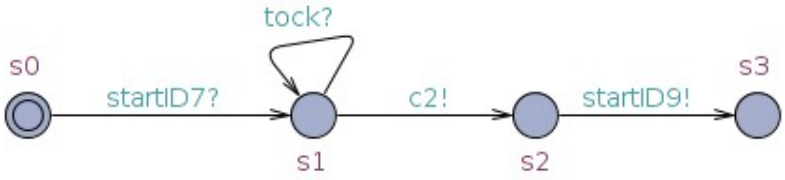
```
12      ] ⌢ transTA(SKIP, startID6, finishID2, ∅, {cs} ∅)
13          = [
```



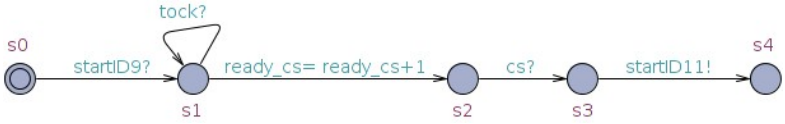

```
14 ]
15
16 = transTA((Q1[|∅|]R1), startID3, finishID3, {cs}, ∅, ∅)
17 = [
```



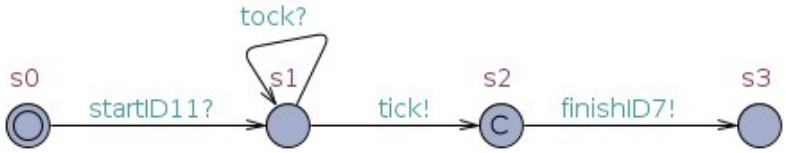

```
18          ] ⌢ transTA(Q1, startID7, finishID7, {cs}, ∅, ∅)
19          ⌢ transTA(R1, startID8, finishID8, {cs}, ∅, ∅)
20
21 transTA(Q1, startID7, finishID7,  {cs}, ∅, ∅)
22  = transTA(c2->cs->SKIP, startID7, finishID7,  {cs}, ∅, ∅)
23  =[
```



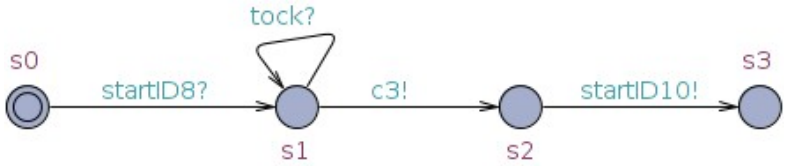

```
24        ] ⌢ transTA(cs->SKIP, startID9, finishID7, {cs}, ∅, ∅)
```



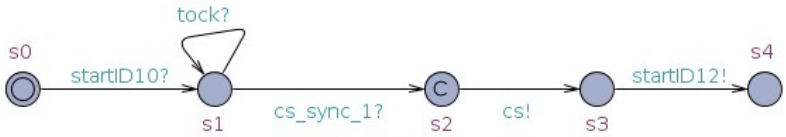

```
25        ] ⌢ transTA(SKIP, startID11, finishID7, {cs}, ∅, ∅)
26        = [
```



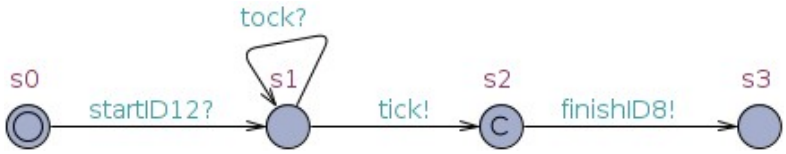

```
27 ]
28
29 transTA(R1, startID8, finishID8,  {cs}, ∅, ∅)
30  = transTA(c3->cs->SKIP, startID8, finishID8,  {cs}, ∅, ∅)
31  = [
```



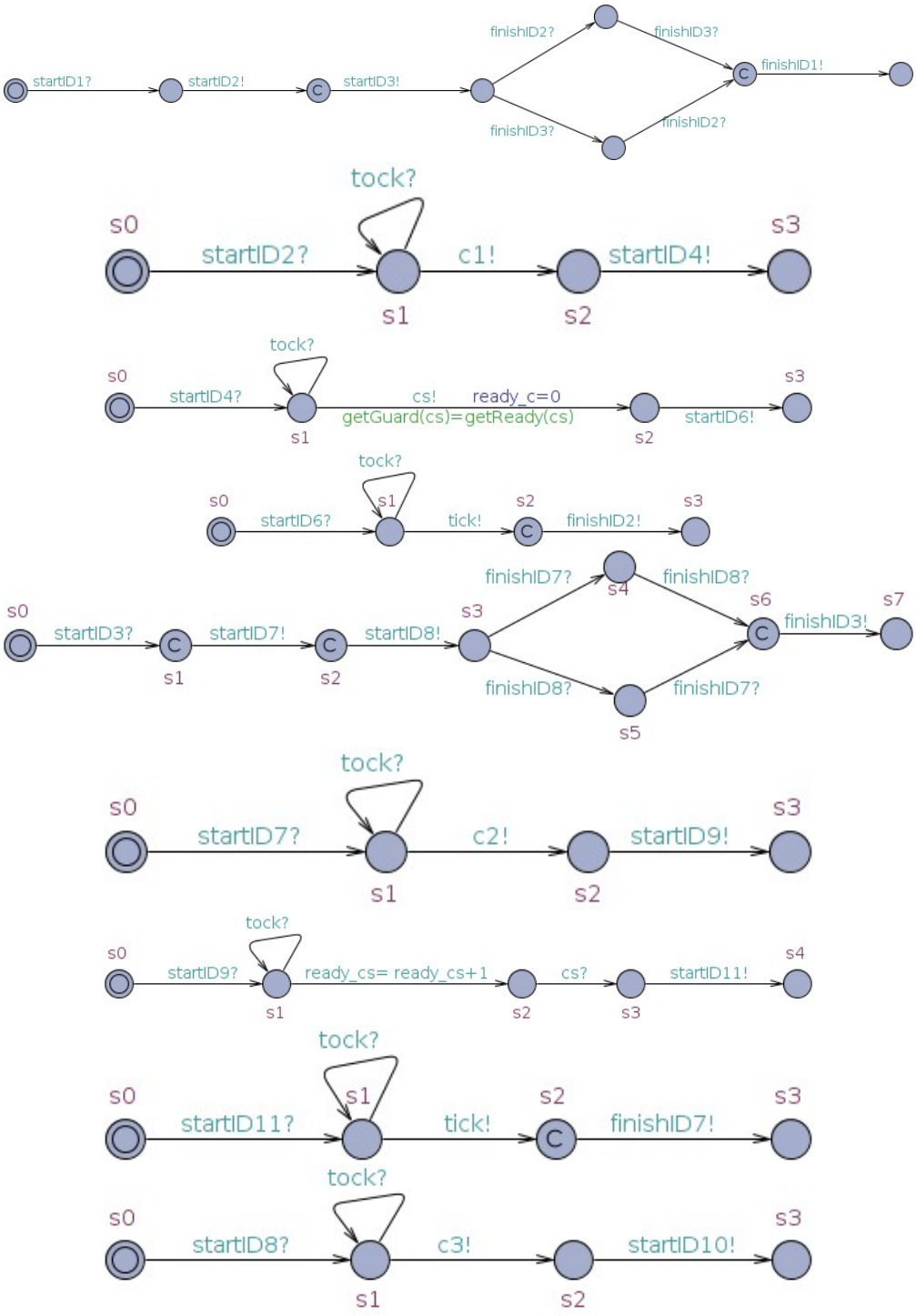

```
32      ] ⌢ transTA(cs->SKIP, startID10, finishID8, {cs}, ∅, ∅)
33      = [
```

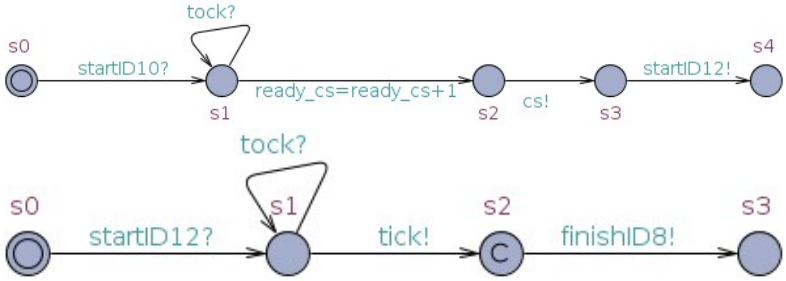



```
34      ] ⌢ transTA(SKIP, startID8, finishID8, {cs}, ∅, ∅)
35      = [
```




```
36 ]
37
38 ∴ transform(P1[|{cs}|](Q1[|∅|]R1))
39  = [
```

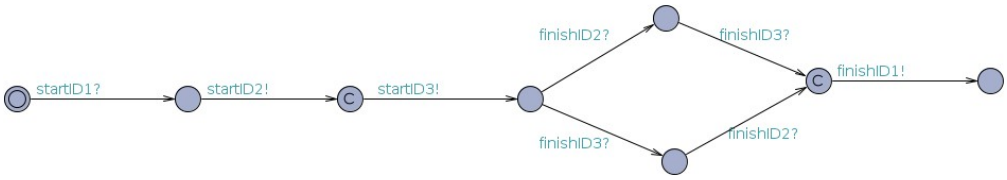

startID1?
startID2!
startID3!
finishID2?
finishID3?
finishID1!
finishID3?
finishID2?
tock?
s0
startID2?
c1!
startID4!
s3
s1
s2
tock?
s0
startID4?
cs!
ready_c=0
getGuard(cs)=getReady(cs)
startID6!
s3
s1
s2
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
s0
startID3?
startID7!
startID8!
s3
finishID7?
s4
finishID8?
s6
finishID3!
s7
finishID8?
finishID7?
s5
tock?
s0
startID7?
c2!
startID9!
s3
s1
s2
tock?
s0
startID9?
ready_cs= ready_cs+1
cs?
startID11!
s4
s1
s2
s3
tock?
s0
startID11?
s1
tick!
s2
finishID7!
s3
tock?
s0
startID8?
c3!
startID10!
s3
s1
s2



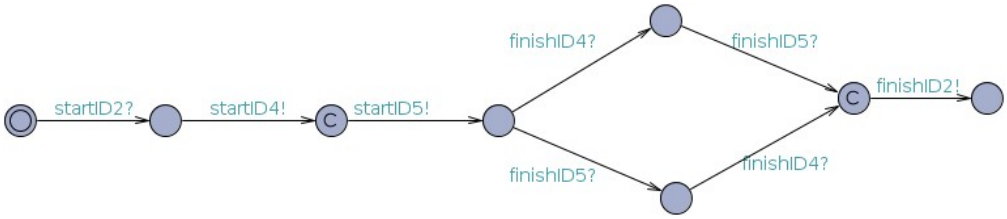
]

Example C.2. $(P[|\{cs\}|]Q1)[|\{\emptyset\}|]R1$

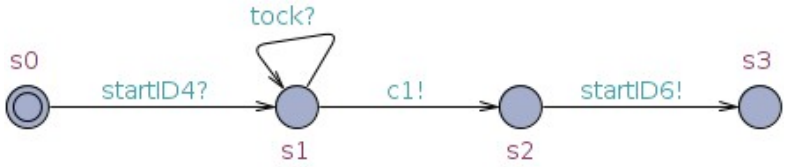
```
1 transform((P1[|{cs}|]Q)[|{ ∅ }|]R1))
2  = transTA((P1[|{cs}|]Q)[|∅|]R1), startID1, finishID1, ∅, ∅, ∅)
3  = [
```




```
4          ] ⌢ transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, ∅, ∅)
5          ⌢ transTA(R1, startID3, finishID3, ∅, ∅, ∅)
6
7  transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, ∅, ∅)
8   = [
```

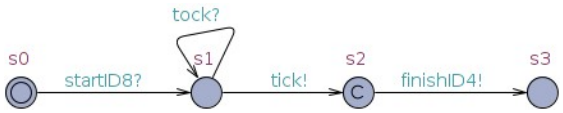



```
9          ] ⌢ transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
10         ⌢ transTA(Q1, startID5, finishID5, {cs}, ∅, ∅)
11
12
13 transTA(P1, startID4, finishID4,  ∅, {cs} ∅)
14  = transTA(c1->cs->SKIP, startID4, finishID4,  ∅, {cs}, ∅)
15  = [
```

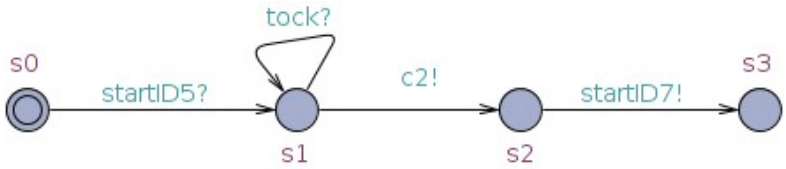



```
16      ] ⌢ transTA(cs->SKIP, startID6, finishID4, ∅, {cs}, ∅)
17        = [
```

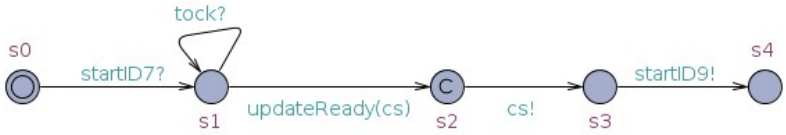

```
18      ] ⌢ transTA(SKIP, startID8, finishID4, ∅, {cs}, ∅)
19          = [
```



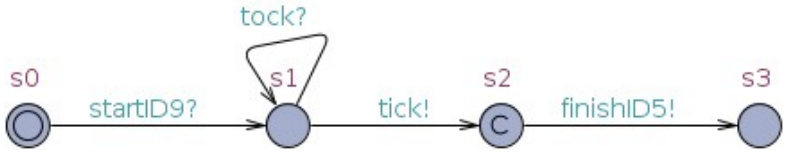

```
20 transTA(Q1, startID5, finishID5, {cs}, ∅, ∅)
21  = transTA(c2->cs->SKIP, startID5, finishID5, {cs}, ∅, ∅)
22      =[
```



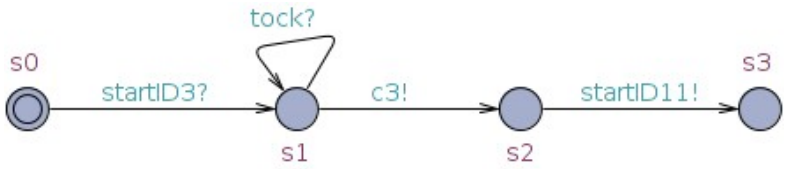

```
23       ] ⌢ transTA(cs->SKIP, startID7, finishID5, {cs}, ∅, ∅)
24         = [
```



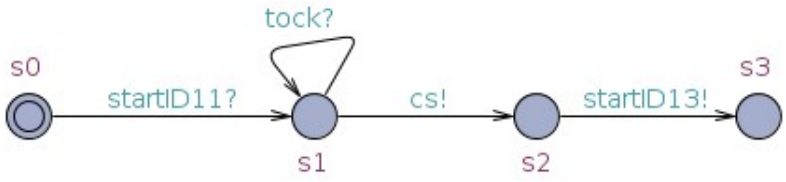

```
25       ] ⌢ transTA(SKIP, startID9, finishID5, {cs}, ∅, ∅)
26         = [
```



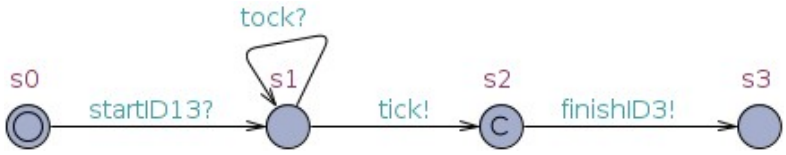

```
      ]
transTA(R1, startID3, finishID3, ∅, ∅, ∅)
 = transTA(c3->cs->SKIP, startID3, finishID3, ∅, ∅, ∅)
 = [
```




```
    ] ⌒ transTA(cs->SKIP, startID11, finishID3, ∅, ∅, ∅)
            =[
```

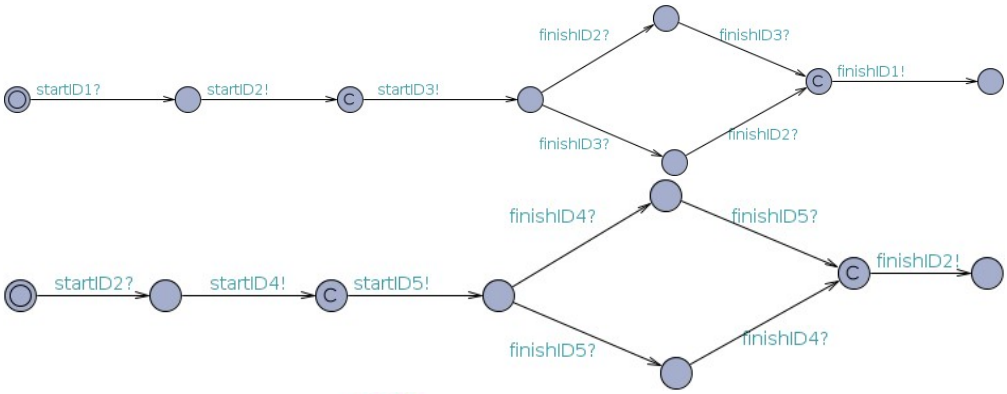


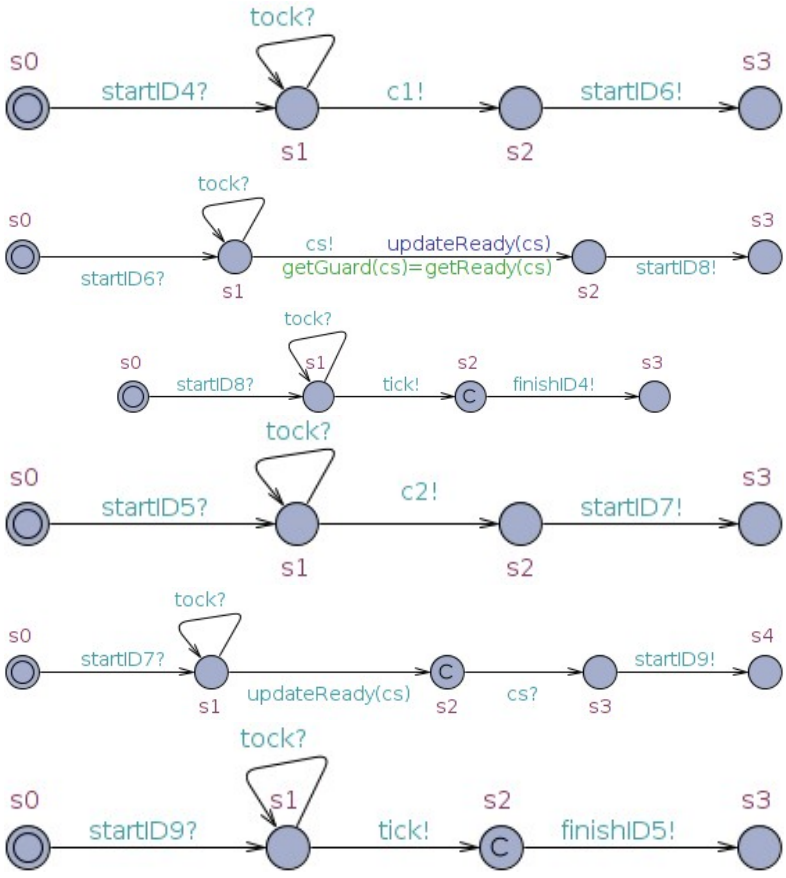

```
    ] ⌒ transTA(SKIP, startID13, finishID3, ∅, ∅, ∅)
          =[
```

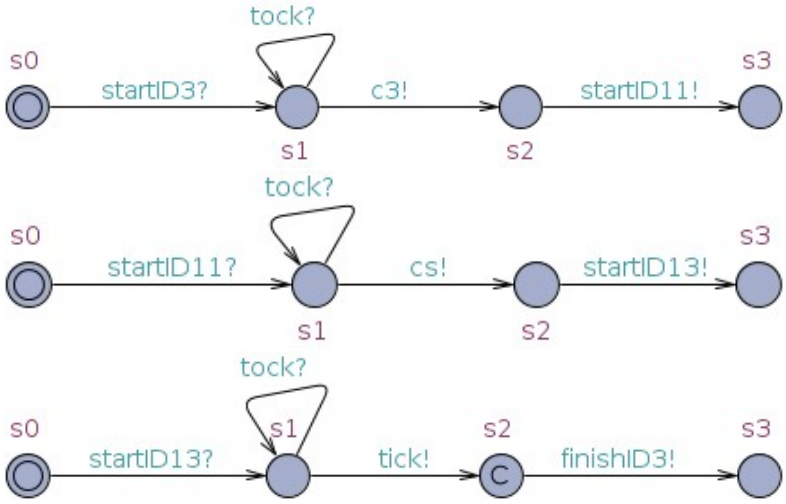



```
            ]
```

```
∴ transform((P1[|{cs}|]Q1)[|∅|]R1)
    = [
```



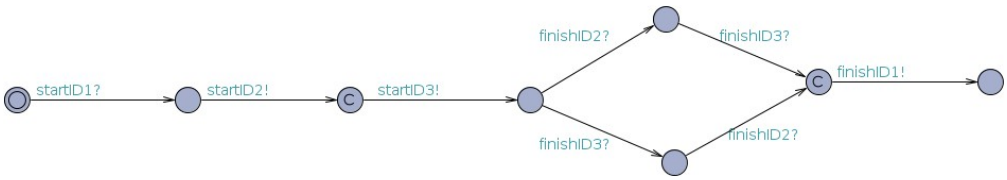




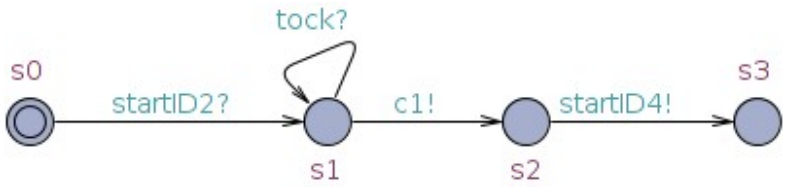
]

Example C.3. $P1[|\emptyset|](Q1[|\{cs\}|]R1)$

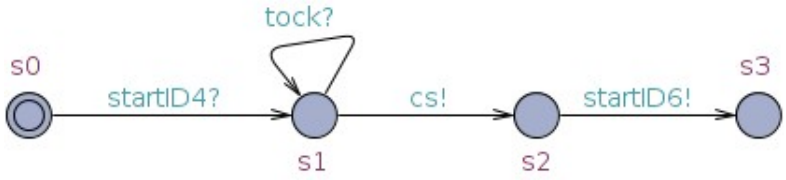
```
transform(P1[|∅|](Q1[|{cs}|]R1))
 = transTA(P1[|∅|](Q1[|{cs}|]R1), startID1, finishID1, ∅, ∅, ∅)
 = [
```



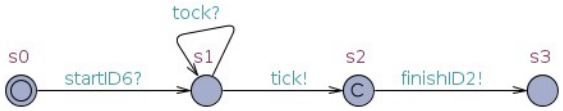

```
        ] ⌢ transTA(P1, startID2, finishID2, ∅, ∅, ∅)
        ⌢ transTA((Q1[|{cs}|]R1), startID3, finishID3, ∅, ∅, ∅)

transTA(P1, startID2, finishID2, ∅, ∅, ∅)
 = transTA(c1->cs->SKIP, startID2, finishID2, ∅, ∅, ∅)
 = [
```



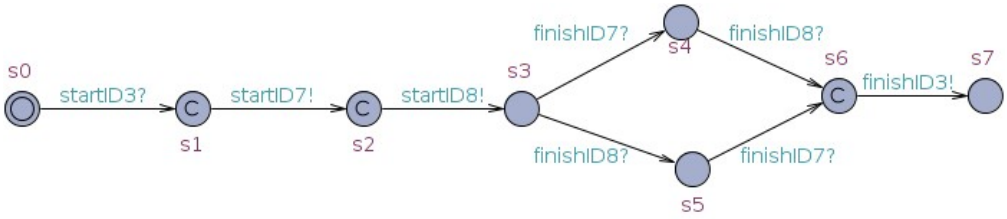

```
    ] ⌢ transTA(cs->SKIP, startID4, finishID2, ∅, ∅, ∅)
      = [
```



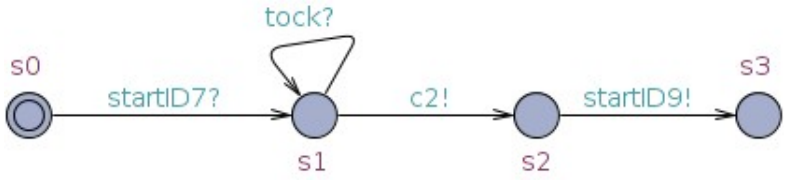

```
      ] ⌢ transTA(SKIP, startID6, finishID2, ∅, ∅, ∅)
          = [
```



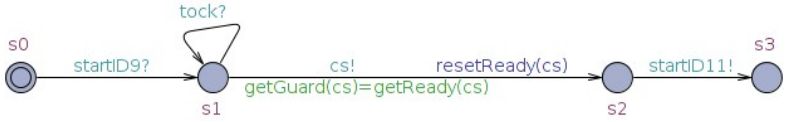

```
]

= transTA((Q1[|{cs}|]R1), startID3, finishID3, ∅, ∅, ∅)
= [
```



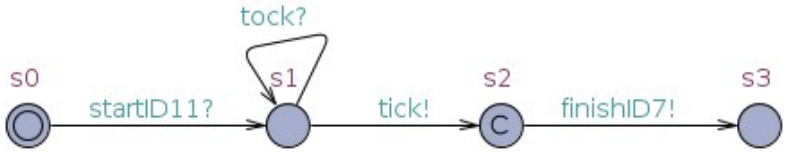

```
        ] ⌢ transTA(Q1, startID7, finishID7, ∅, {cs}, ∅)
        ⌢ transTA(R1, startID8, finishID8, {cs}, ∅, ∅)

transTA(Q1, startID7, finishID7, ∅, {cs}, ∅)
 = transTA(c2->cs->SKIP, startID7, finishID7, ∅, {cs}, ∅)
 = [
```



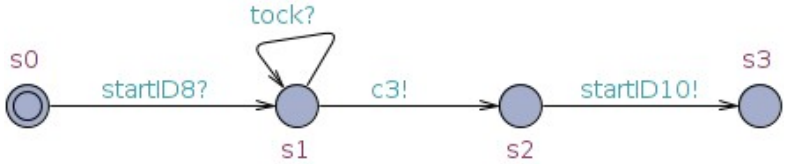

```
       ] ⌢ transTA(cs->SKIP, startID9, finishID7, ∅, {cs}, ∅)
       = [
```



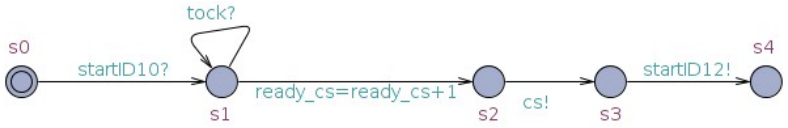

```
        ] ⌢ transTA(SKIP, startID11, finishID7, ∅, {cs}, ∅)
        = [
```



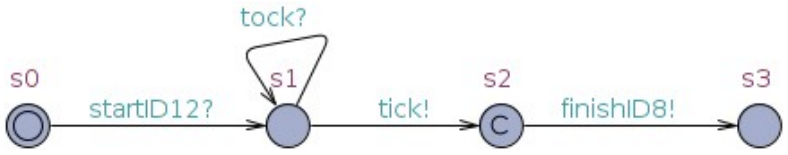

```
transTA(R1, startID8, finishID8, {cs}, ∅)
 = transTA(c3->cs->SKIP, startID8, finishID8, {cs}, ∅, ∅)
 = [
```



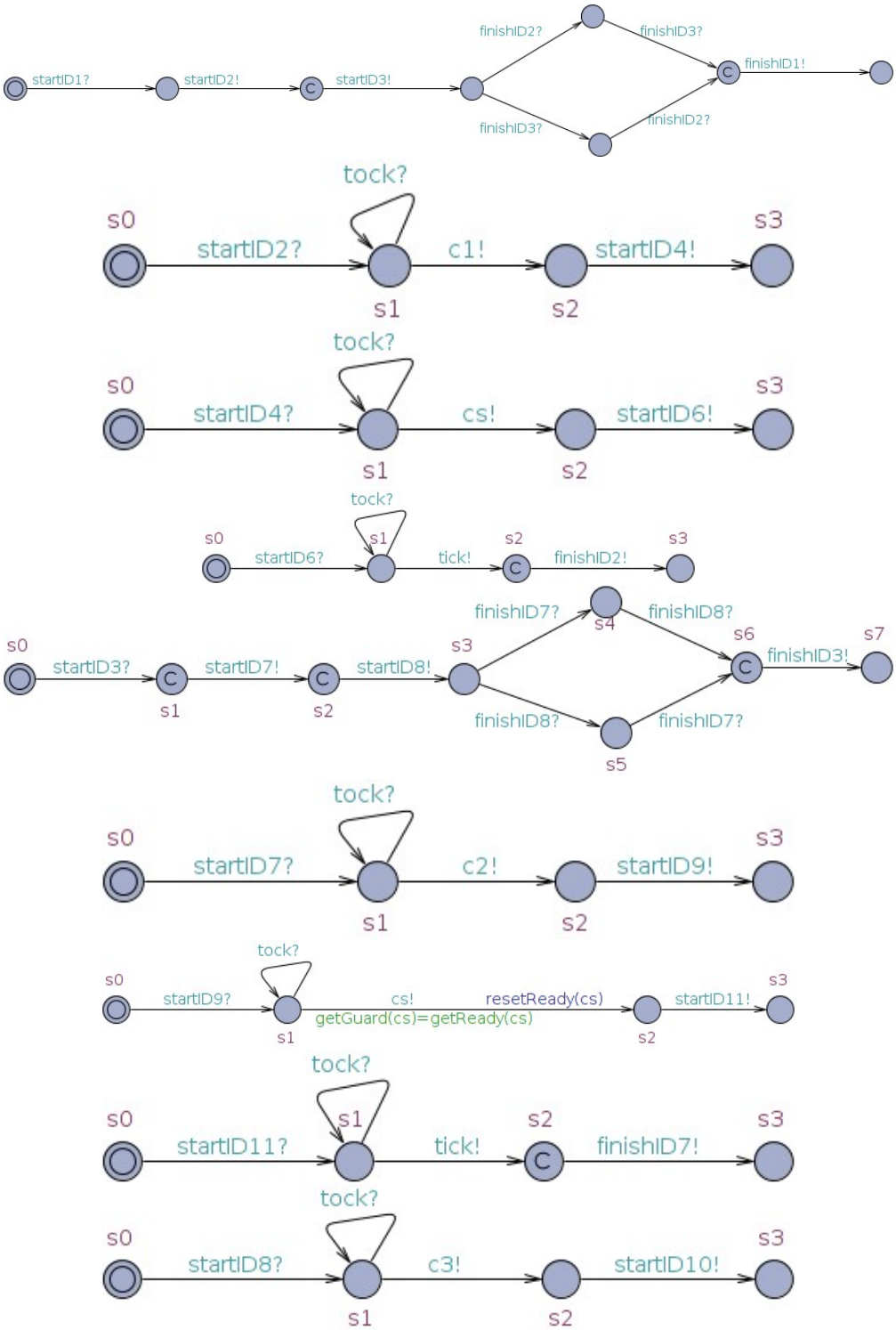

```
    ] ⌢ transTA(cs->SKIP, startID10, finishID8, {cs}, ∅, ∅)
     = [
```

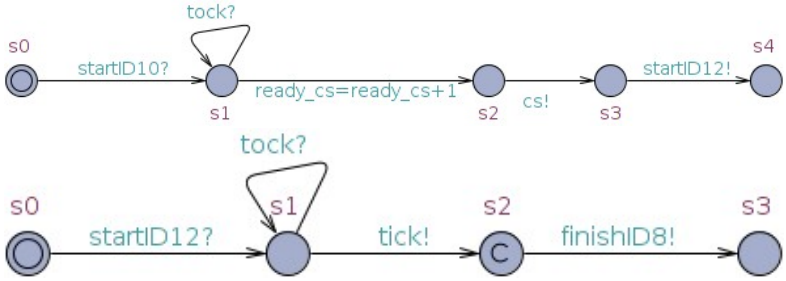



```
    ] ⌢ transTA(SKIP, startID12, finishID8, {cs}, ∅, ∅)
```




```
]

∴ transform(P1[|∅|](Q1[|{cs}|]R1))
 = [
```

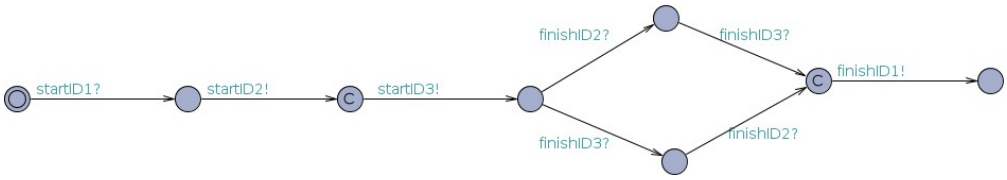

startID1?
startID2!
startID3!
finishID2?
finishID3?
finishID1!
finishID3?
finishID2?
tock?
s0
startID2?
c1!
startID4!
s3
s1
s2
tock?
s0
startID4?
cs!
startID6!
s3
s1
s2
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
finishID7?
s4
finishID8?
s0
startID3?
s1
startID7!
s2
startID8!
s3
s6
finishID3!
s7
finishID8?
finishID7?
s5
tock?
s0
startID7?
c2!
startID9!
s3
s1
s2
tock?
s0
startID9?
cs!
resetReady(cs)
getGuard(cs)=getReady(cs)
startID11!
s3
s1
s2
tock?
s0
startID11?
s1
tick!
s2
finishID7!
s3
tock?
s0
startID8?
c3!
startID10!
s3
s1
s2



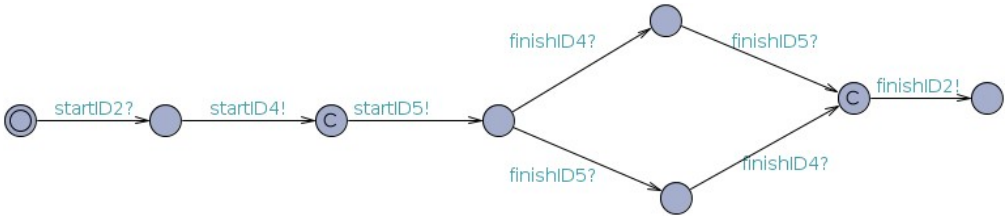
]

Example C.4. $(P1[|\emptyset|]Q1)[|\{cs\}|]R1$

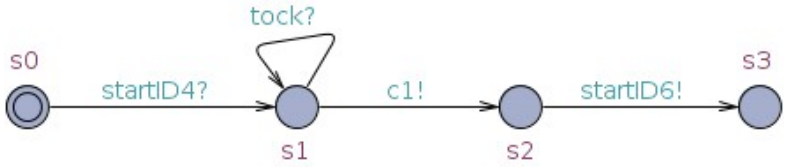
```
1 transform((P1[|∅|]Q1)[|{cs}|]R1)
2  = transTA((P1[|∅|]Q1)[|{cs}|]R1, startID1, finishID1, ∅, ∅, ∅)
3  = [
```




```
4          ] ⌢ transTA((P1[|∅|]Q1), startID2, finishID2, {cs}, ∅, ∅)
5          ⌢ transTA(R1, startID3, finishID3, {cs}, ∅, ∅)
6
7 transTA((P1[|∅|]Q1), startID2, finishID2, ∅, {cs}, ∅)
8  = [
```

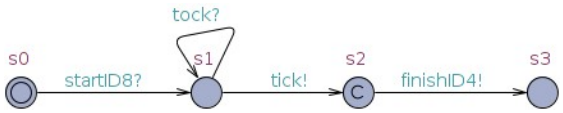



```
9          ] ⌢ transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
10         ⌢ transTA(Q1, startID5, finishID5, ∅, {cs}, ∅)
11
12
13 transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
14  = transTA(c1->cs->SKIP, startID4, finishID4, ∅, {cs}, ∅)
15  = [
```

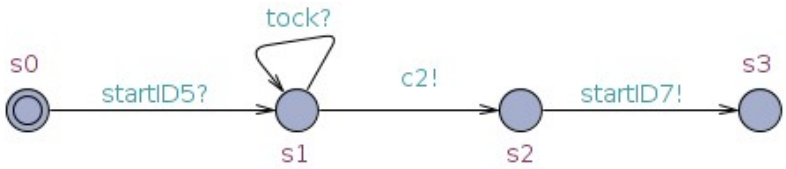



```
16      ] ⌒ transTA(cs->SKIP, startID6, finishID4, ∅, {cs}, ∅)
17          = [
```

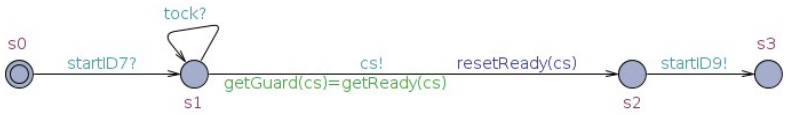

```
18       ] ⌒ transTA(SKIP, startID8, finishID4, ∅, {cs}, ∅)
19           = [
```



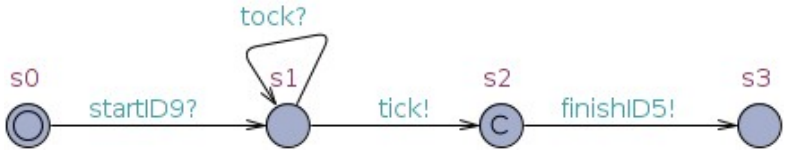

```
20 ]
21
22 transTA(Q1, startID5, finishID5, ∅, {cs}, ∅)
23  = transTA(c2->cs->SKIP, startID5, finishID5, ∅, {cs}, ∅)
24      =[
```




```
25        ] ⌒ transTA(cs->SKIP, startID7, finishID5, ∅, {cs}, ∅)
26            = [
```

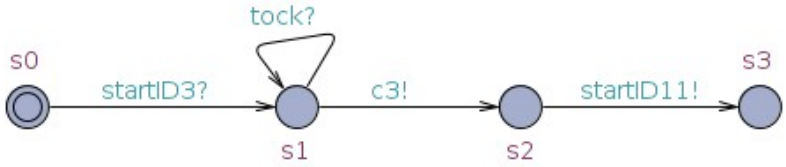



```
27        ] ⌒ transTA(SKIP, startID9, finishID5, ∅, {cs}, ∅)
28            = [
```

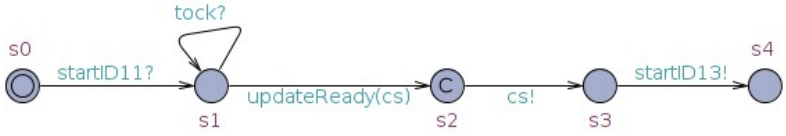


]

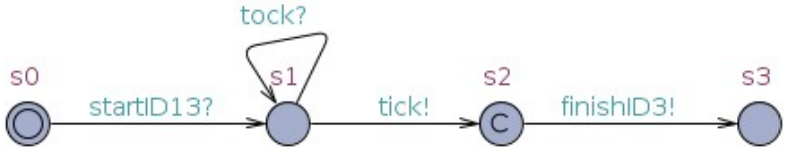
```
transTA(R1, startID3, finishID3, {cs}, ∅, ∅)
 = transTA(c3->cs->SKIP, startID3, finishID3, {cs}, ∅, ∅)
 = [
```



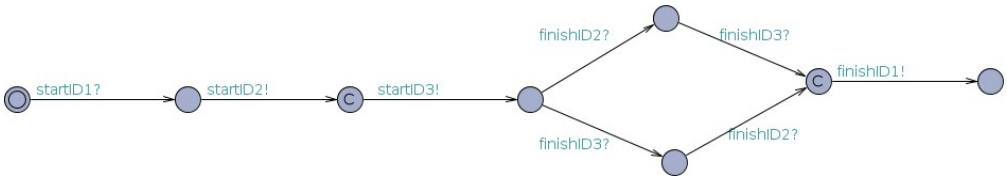

```
    ] ⌢ transTA(cs->SKIP, startID11, finishID3, {cs}, ∅, ∅)
            =[
```

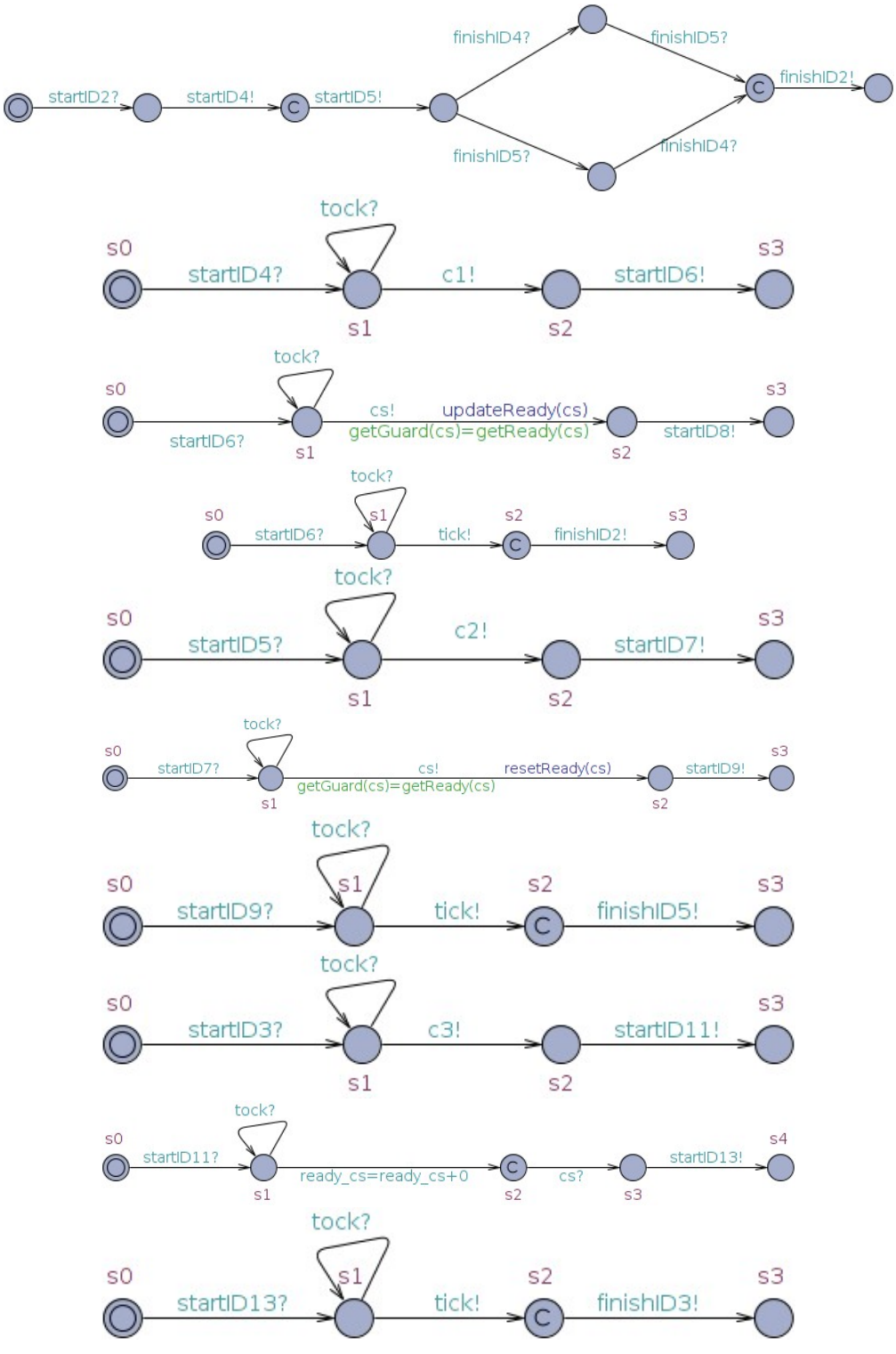



```
    ] ⌢ transTA(SKIP, startID13, finishID3, {cs}, ∅, ∅)
        =[
```




```
            ]

∴ transform((P1[|∅|]Q1)[|{cs}|]R1)
   = [
```

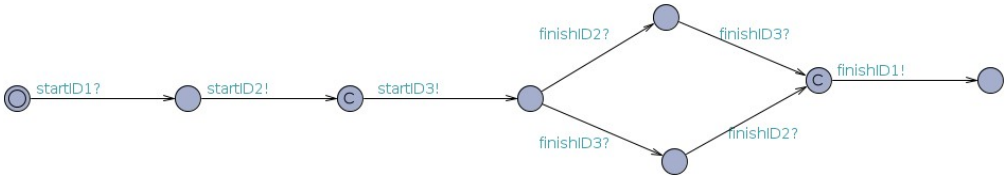



startID2?
startID4!
startID5!
finishID4?
finishID5?
finishID2!
finishID5?
finishID4?
tock?
s0
startID4?
c1!
startID6!
s3
s1
s2
tock?
s0
startID6?
cs!
getGuard(cs)=getReady(cs)
updateReady(cs)
startID8!
s3
s1
s2
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
tock?
s0
startID5?
c2!
startID7!
s3
s1
s2
tock?
s0
startID7?
cs!
getGuard(cs)=getReady(cs)
resetReady(cs)
startID9!
s3
s1
s2
tock?
s0
startID9?
s1
tick!
s2
finishID5!
s3
tock?
s0
startID3?
c3!
startID11!
s3
s1
s2
tock?
s0
startID11?
ready_cs=ready_cs+0
cs?
startID13!
s4
s1
s2
s3
tock?
s0
startID13?
s1
tick!
s2
finishID3!
s3

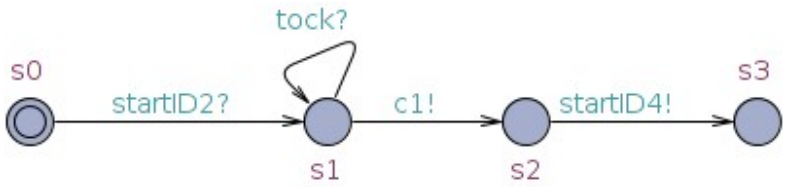
]

Example C.5. $P1[|\{cs\}|](Q1[|\{cs\}|]R1)$

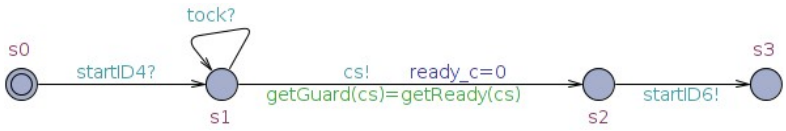
```
transform(P1[|{cs}|](Q1[|{cs}|]R1))
 = transTA(P1[|{cs}|](Q1[|{cs}|]R1), startID1, finishID1, ∅, ∅, ∅)
 = [
```



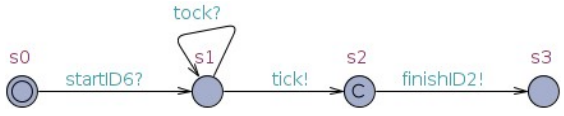

```
        ] ⌢ transTA(P1, startID2, finishID2, ∅, {cs}, ∅)
        ⌢ transTA((Q1[|{cs}|]R1), startID3, finishID3, {cs}, ∅, ∅)

transTA(P1, startID2, finishID2, ∅, {cs}, ∅)
 = transTA(c1->cs->SKIP, startID2, finishID2, ∅, {cs}, ∅)
 = [
```



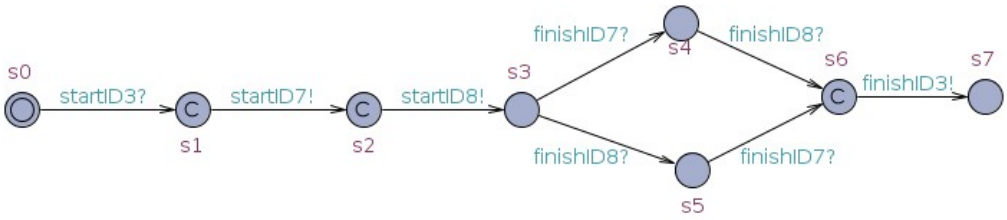

```
    ] ⌢ transTA(cs->SKIP, startID4, finishID2, ∅, {cs}, ∅)
      = [
```



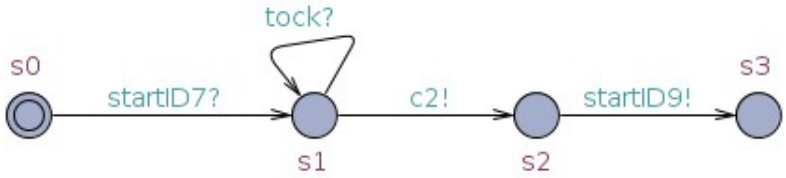

```
     ] ⌢ transTA(SKIP, startID6, finishID2, ∅, {cs}, ∅)
       = [
```



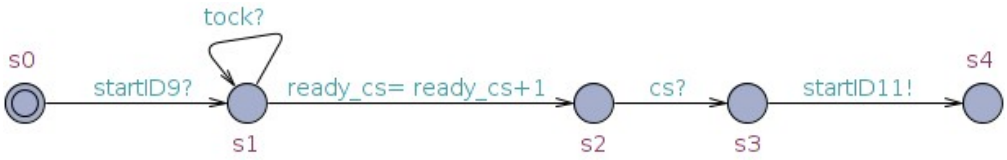

```
]
= transTA((Q1[|{cs}|]R1), startID3, finishID3, {cs}, ∅, ∅)
= [
```



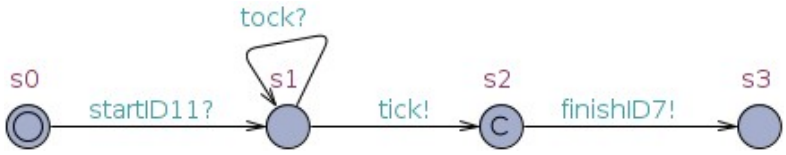

```
14          ] ⌢ transTA(Q1, startID7, finishID7, {cs}, {cs}, ∅)
15          ⌢ transTA(R1, startID8, finishID8, {cs}, ∅, ∅)
16
17 transTA(Q1, startID7, finishID7, {cs}, {cs}, ∅)
18  = transTA(c2->cs->SKIP, startID7, finishID7, {cs}, {cs}, ∅)
19  = [
```



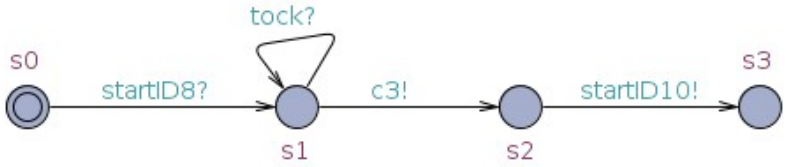

```
20        ] ⌢ transTA(cs->SKIP, startID9, finishID7, {cs}, {cs}, ∅)
```



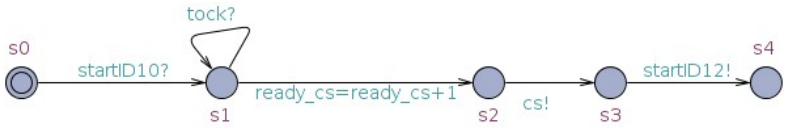

```
21       ] ⌢ transTA(SKIP, startID11, finishID7, {cs}, {cs}, ∅)
```



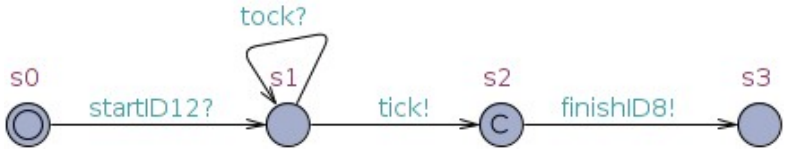

```
22 ]
23 transTA(R1, startID8, finishID8, {cs}, ∅, ∅)
24  = transTA(c3->cs->SKIP, startID8, finishID8, {cs}, ∅, ∅)
25  = [
```



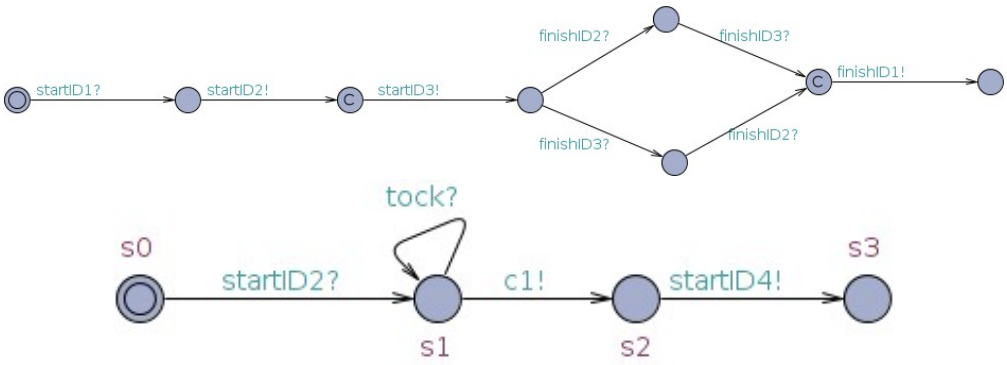

```
26      ] ⌢ transTA(cs->SKIP, startID10, finishID8, {cs}, ∅, ∅)
27        = [
```

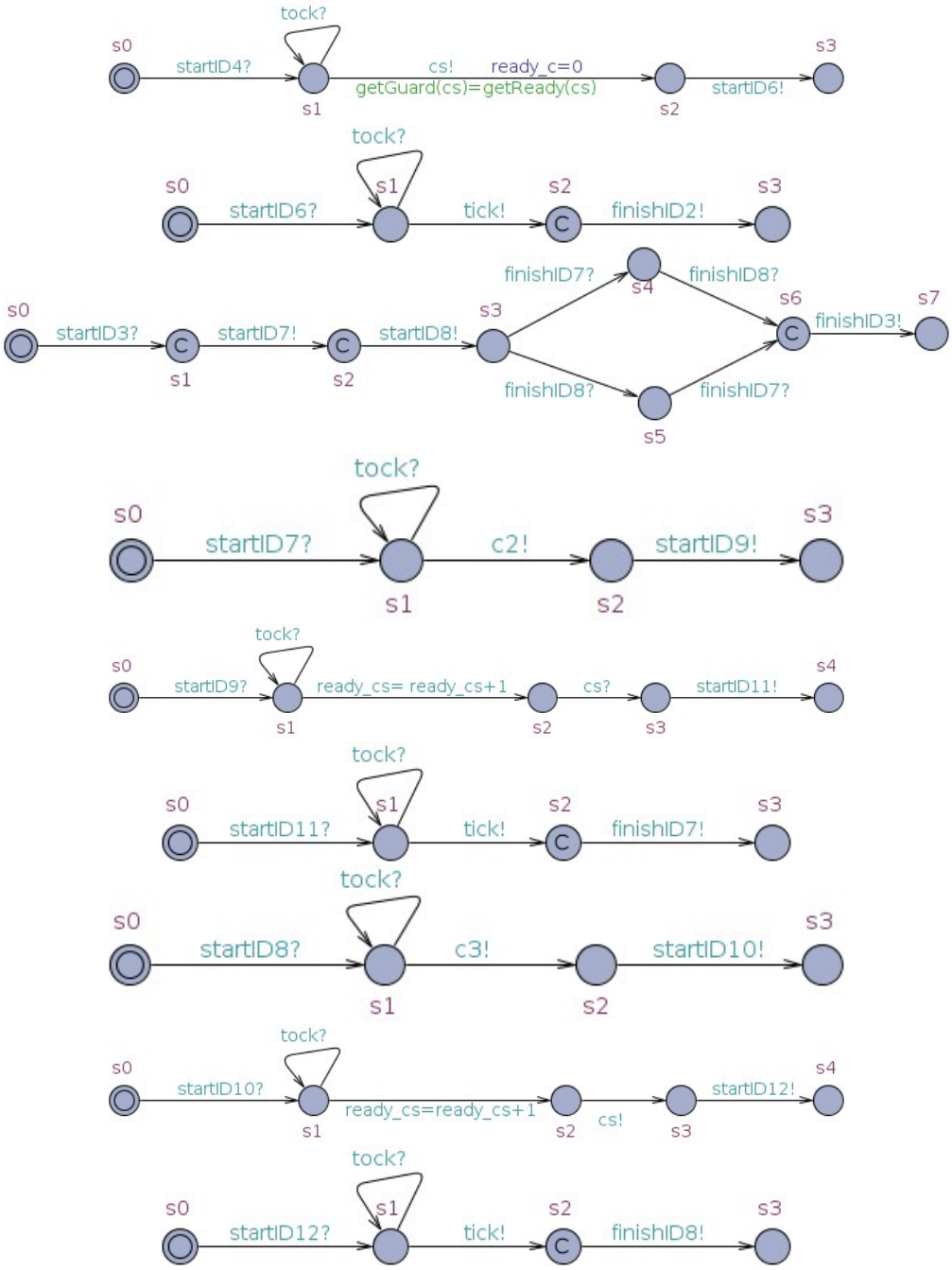



```
28      ] ⌢ transTA(SKIP, startID8, finishID8, {cs}, ∅, ∅)
29      = [
```




```
30 ]
31
32 ∴ transform(P1[|{cs}|](Q1[|{cs}|]R1))
33   = [
```

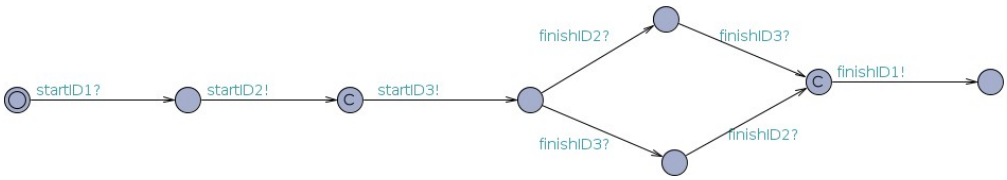



tock?
s0
startID4?
s1
cs!
ready_c=0
getGuard(cs)=getReady(cs)
s2
startID6!
s3
tock?
s0
startID6?
s1
tick!
s2
finishID2!
s3
finishID7?
s4
finishID8?
s0
startID3?
s1
startID7!
s2
startID8!
s3
s6
finishID3!
s7
finishID8?
finishID7?
s5
tock?
s0
startID7?
s1
c2!
s2
startID9!
s3
tock?
s0
startID9?
s1
ready_cs= ready_cs+1
s2
cs?
s3
startID11!
s4
tock?
s0
startID11?
s1
tick!
s2
finishID7!
s3
tock?
s0
startID8?
s1
c3!
s2
startID10!
s3
tock?
s0
startID10?
s1
ready_cs=ready_cs+1
s2
cs!
s3
startID12!
s4
tock?
s0
startID12?
s1
tick!
s2
finishID8!
s3

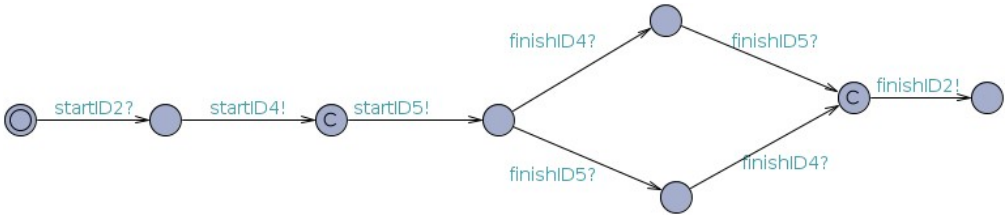
]

Example C.6. $(P1[|\{cs\}|]Q1)[|\{cs\}|]R1$

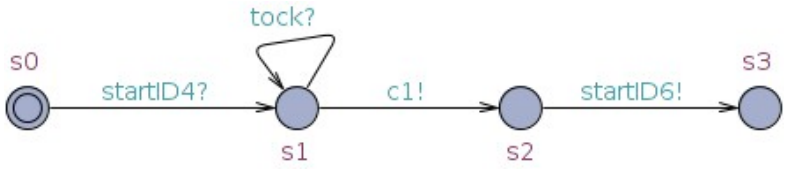
```
transform((P1[|{cs}|]Q1)[|{cs}|]R1)
 = transTA((P1[|{cs}|]Q1)[|{cs}|]R1), startID1, finishID1, ∅, ∅, ∅)
 = [
```



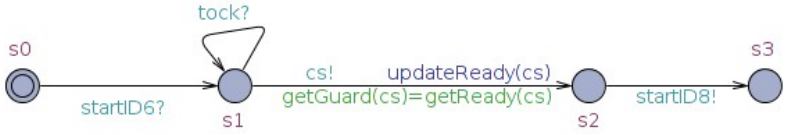

```
        ] ⌢ transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, {cs}, ∅)
        ⌢ transTA(R1, startID3, finishID3, {cs}, ∅, ∅)

transTA((P1[|{cs}|]Q1), startID2, finishID2, ∅, ∅)
 = [
```



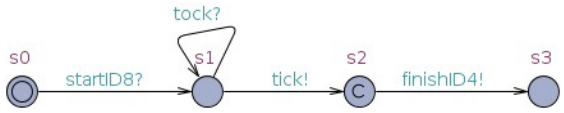

```
        ] ⌢ transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
        ⌢ transTA(Q1, startID5, finishID5, {cs}, {cs}, ∅)


transTA(P1, startID4, finishID4, ∅, {cs}, ∅)
 = transTA(c1->cs->SKIP, startID4, finishID4, ∅, {cs}, ∅)
 = [
```



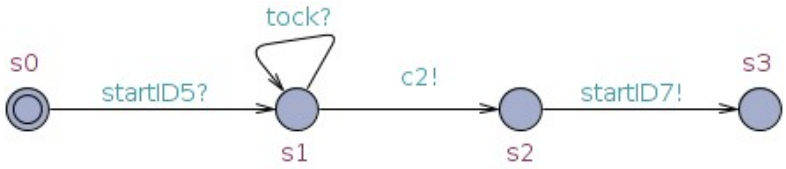

```
    ] ⌢ transTA(cs->SKIP, startID6, finishID4, ∅, {cs}, ∅)
      = [
```



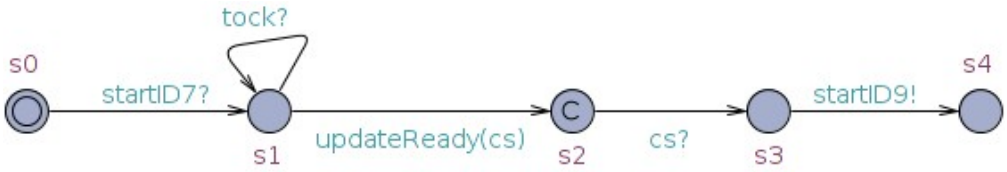
```
15      ] ⌢ transTA(SKIP, startID8, finishID4, ∅, {cs}, ∅)
16          = [
```



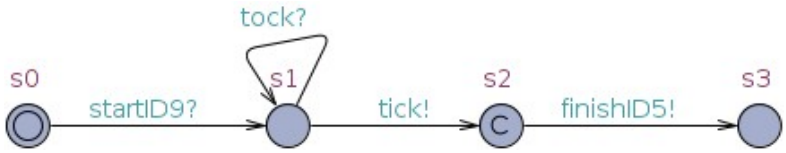

```
17 transTA(Q1, startID5, finishID5, {cs}, {cs}, ∅)
18  = transTA(c2->cs->SKIP, startID5, finishID5, {cs}, {cs}, ∅)
19      =[
```



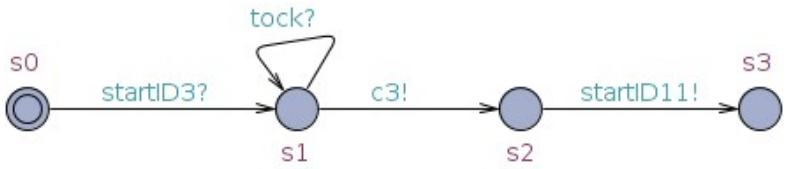

```
20      ] ⌢ transTA(cs->SKIP, startID7, finishID5, {cs}, {cs}, ∅)
21          = [
```



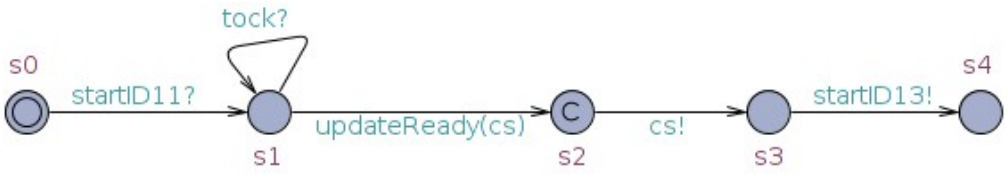

```
22      ] ⌢ transTA(SKIP, startID9, finishID5, {cs}, {cs}, ∅)
23          = [
```



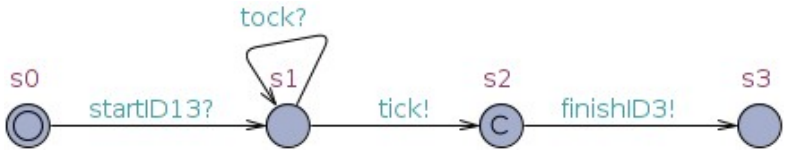

```
      ]
24 transTA(R1, startID3, finishID3, {cs}, ∅, ∅)
25  = transTA(c3->cs->SKIP, startID3, finishID3, {cs}, ∅, ∅)
26  = [
```



27] ⌢ transTA(cs->SKIP, startID11, finishID3, {cs}, ∅, ∅)
28 =[

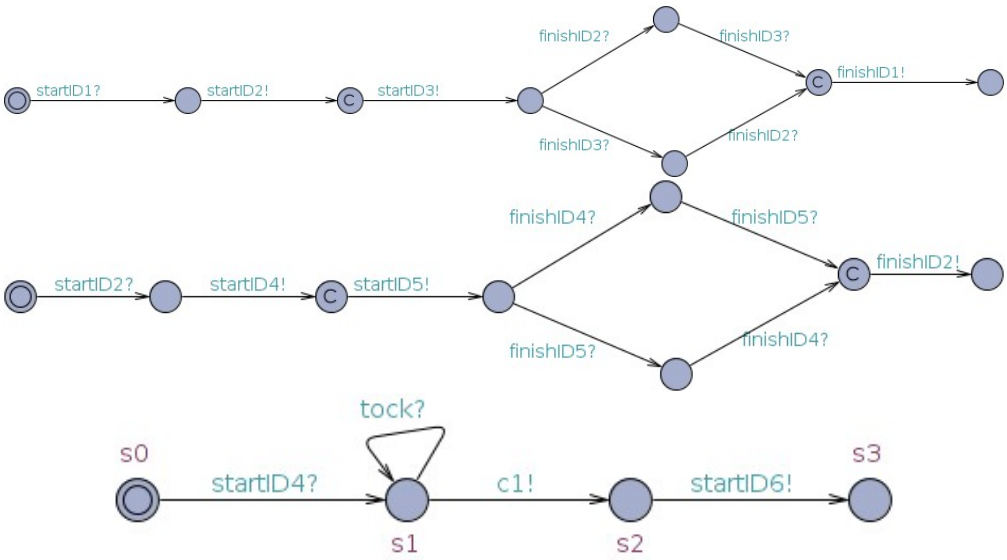


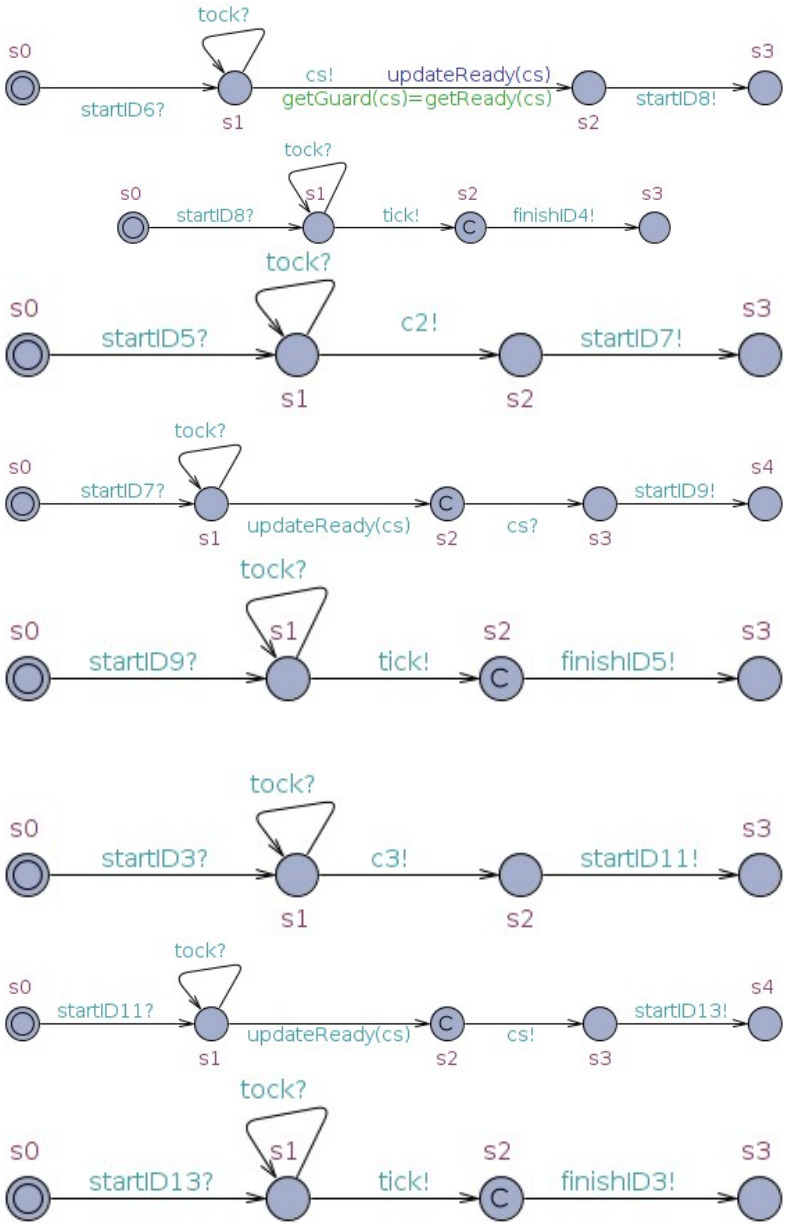
29] ⌢ transTA(SKIP, startID13, finishID3, {cs}, ∅, ∅)
30 =[



]

31 ∴ transform((P1[|{cs}|]Q1)[|{cs}|]R1)
32 = [




tock?
s0
cs!
updateReady(cs)
s3
startID6?
getGuard(cs)=getReady(cs)
s1
s2
startID8!
tock?
s0
s1
s2
s3
startID8?
tick!
C
finishID4!
tock?
s0
startID5?
c2!
startID7!
s3
s1
s2
tock?
s0
startID7?
s4
C
startID9!
s1
updateReady(cs)
s2
cs?
s3
tock?
s0
s1
s2
s3
startID9?
tick!
C
finishID5!
tock?
s0
startID3?
c3!
startID11!
s3
s1
s2
tock?
s0
startID11?
s4
C
startID13!
s1
updateReady(cs)
s2
cs!
s3
tock?
s0
s1
s2
s3
startID13?
tick!
C
finishID3!

]

D. A Basic Tool for Checking the Steps of the Proof

This is the source code of the basic tool for automatically checking the steps of the proof, mainly formulae, syntax and terms used in the proof [10].

```
1
2 {- The function pcpl is a quick-and-dirty way of revealing errors in
     a particular instance of the claim that every term in the proof is
      equal to every other term. If the claim is true, it reports "GOOD
      PROOF!", otherwise it reports that the proof is bad, and shows
     the values at each step. An incorrect step is revealed by two
     adjacent values being different. The code was developed by Jeremy
     Jacob.
3 -}
4
5 module ProofLayout where
6
7 import Data.List
8
9 infixr 0 :=:
10 data ProofLayout a = a :=: (ProofLayout a) | QED
11 instance Foldable ProofLayout where
12   foldr f z = ffz
13     where ffz QED = z
14           ffz (l :=: pl) = f l (ffz pl)
15
16 pcpl :: (Eq a, Show a) => ProofLayout a -> IO ()
17 pcpl = putStr . report . allOK . foldr (:) []
18   where
19     allOK pl = (and [ l == m | (l, m) <- zip pl (tail pl)], pl)
20     report (True, _ ) = "GOOD␣PROOF!\n"
21     report (_   , pl) = "BAD␣PROOF!\n" ++ foldr ((++) . ("\n␣␣"++) .
          (++"\n:=:") . show) "QED\n" pl
```

[10] Thanks to Jeremy Jacob for providing this code.

E. Details of the Proof

```
1  -- Contents
2  -- 1. Proof
3  -- 2. Definitions
4
5
6  module Proof_function where
7
8  import Data.List
9  import ProofLayout
10
11
12 {---------- Begining of the proof -------------------------
13 Proof by induction over the Natural numbers.
14 forall n:Nat, forall P::Proc, traces_tockCSP n P = traces_TA n .
       transTA P
15
16 BASE CASE for n=0 ----------------------------------------------
17  for n=0,
18      forall P::Proc, traces_tockCSP 0 P = traces_TA 0 (transTA P)
19 -}
20
21
22 proofTrans :: CSPproc -> Int -> ProofLayout [[Event]]
23 proofTrans    p          0   =
24     traces_tockCSP 0 p
25     :=: --{traces_tockCSP.0, l~>r, _0<~p}
26     [[]]
27     :=: --{traces_TA.0, r~>l, _0<~transTA p}
28     traces_TA 0 (transTA p  ""  0  0  0 [])
29     :=: QED
30 {- Because we have used no properties of P, other than P::CSPproc, we
       may
31    conclude, by the rule of generalisation:
32     forall P::Proc, traces_tockCSP 0 P = traces_TA 0 (transTA P)
33 -}
34
35
36 {- INDUCTION STEP for n>0
       ------------------------------------------
37  for n>0: forall P::Proc,
38    (traces_TA n (transTA P) = traces_tockCSP n P)
39    => (traces_TA (n+1) (transTA P) = traces_tockCSP (n+1) P)
40 -}
41
42
43
44
```

```
-- Base case: STOP
-- (traces_tockCSP n STOP = traces_TA n (transTA STOP))
--   => (traces_tockCSP (n+1) STOP = traces_TA (n+1) (transTA STOP))
proofTrans    STOP        n   =
    traces_tockCSP (n+1) STOP
    :=: --{traces_tockCSP.stop, n<~n+1}
    [(replicate l "tock") | l <- [0..n+1]]
    :=: --{List comprehensions }
    [(replicate l "tock") | l <- [0..n]] ++ [replicate (n+1) "tock"]
    :=: --{traces_tockCSP.stop}
    (traces_tockCSP n STOP) ++ [replicate (n+1) "tock"]
    :=: --{Induction hypothesis}
    nub ((traces_TA n (transTA STOP "ta" 0 0 0 [])) ++ [replicate (n
       +1) "tock"])
    :=: --{transTA.stop, l~>r}
    nub (traces_TA n taSTOP
         ++ [replicate (n+1) "tock"])
    :=: --{tracesTA.n, l~>r, tas<~STOP}
    nub ([t \\ ["startID0_0"] |
           t <- (traces'TA n taSTOP ) ]
         ++ [replicate (n+1) "tock"] )
    :=: --{traces'TA.stop, l~>r}
    [t \\ flowActions taSTOP |t <- ([("startID0_0"):s
                                |s <- [(replicate l "tock")|l <- [0..n
                                   ] ] ] ) ]
    ++ [replicate (n+1) "tock"]
    :=: --{Introducing the connection action}
    [t \\ flowActions taSTOP | t <- ([("startID0_0"):s
                                 |s <- [(replicate l "tock")|l <- [0..
                                    n] ] ])  ]
    ++ [ t \\ ["startID0_0"] | t <- [["startID0_0"] ++ (replicate (n
       +1) "tock")] ]
    :=: --{List comprehensions}
    [t \\ flowActions taSTOP | t <- ([("startID0_0"):s
                                   |s <- [(replicate l "tock")|l <-
                                      [0..n+1] ] ] ) ]
    :=: --{traces'TA.stop, r~>l}
    nub ([t \\ flowActions taSTOP
           | t <- (traces'TA (n+1) taSTOP ) ] )
    :=: --{tracesTA.n, r~>l, n<~n+1, tas<~taSTOP}
    nub (traces_TA  (n+1)  taSTOP )
    :=: --{trans.stop, r->l}
    nub (traces_TA   (n+1)  (transTA STOP "ta" 0 0 0 []))
    :=: QED
        where
           taSTOP = [([ "s0", "s1"], "s0",  ["ck"], ["startID0_0"], [
               "tock"],
                      [("s0", "startID0_0", [], [], "s1"), ("s1", "
                         tock", "ck<=1", "ck", "s1")], [] )]
```

```
87
88 -- Therefore, traces_tockCSP (n+1) STOP = traces_TA  (n+1)  (transTA
       STOP)
89
90
91
92
93 --  Base case: SKIP
94 --  (traces_tockCSP n SKIP = traces_TA n (transTA SKIP))
95 --   => (traces_tockCSP (n+1) SKIP = traces_TA (n+1) (transTA SKIP))
96 proofTrans    SKIP        n   =
97     sort (traces_tockCSP (n+1) SKIP)
98     :=: --{traces_tockCSP.skip, n<~n+1}
99     sort ([(replicate l "tock") | l <- [0..n+1] ] ++
100    [(replicate l "tock")++["tick"] | l <- [0..n] ] )
101    :=: --{List comprehensions }
102    sort ([(replicate l "tock") | l <- [0..n] ]
103            ++ [(replicate l "tock")++["tick"] | l <- [0..(n-1)]]
104            ++ [replicate (n+1) "tock"] ++ [(replicate n "tock")++["
                  tick"]]  )
105    :=: --{traces_tockCSP.skip}
106    sort (traces_tockCSP n SKIP
107           ++ [ replicate (n+1) "tock"]
108           ++ [(replicate  n    "tock")++["tick"]]  )
109    :=: --{Induction hypothesis}
110    sort (nub (traces_TA n (transTA SKIP  "ta" 0 0 0 []))
111                ++ [ replicate (n+1) "tock"]
112                ++ [(replicate  n    "tock")++["tick"]] ) )
113    :=: --{transTA.skip, l~>r}
114    sort (nub (traces_TA n taSKIP
115                ++ [ replicate (n+1) "tock"]
116                ++ [(replicate  n    "tock")++["tick"]] ) )
117    :=: --{tracesTA.n, l~>r, tas<~SKIP}
118    sort (nub ([t \\ ["startID0_0"] | t <- (traces'TA n taSKIP ) ]
119                 ++ [ replicate (n+1) "tock"]
120                 ++ [(replicate  n    "tock")++["tick"]] ) )
121    :=: --{traces'TA.skip, l~>r}
122    sort ([t \\ flowActions taSKIP |t <- ([("startID0_0"):s | s <-
          ([(replicate l "tock")|l <- [0..n] ]
123                  ++ [(replicate l "tock") ++ ["tick"]|l <- [0..(n-1)
                        ] ] ) ] ) ]
124          ++ [replicate (n+1) "tock"] ++ [(replicate n "tock") ++ ["
                tick"]]   )
125    :=: --{Introducing the connection action }
126    sort ([t \\ flowActions taSKIP
127                |t <- ([("startID0_0"):s
128                    |s <- ([(replicate l "tock")|l <- [0..n] ]
129                ++ [(replicate l "tock") ++ ["tick"]|l <- [0..(n-1)]
                      ] ) ] ) ]
```

```
130             ++ [ t \\ ["startID0_0"]
131                       | t <- ([["startID0_0"] ++ (replicate (n+1) "tock
                              ")] ++
132                                 [["startID0_0"] ++ (replicate  n     "
                                       tock") ++ ["tick"]]) ] )
133     :=: --{List comprehensions}
134     sort [t \\ flowActions taSKIP
135               |t <- ([("startID0_0"):s
136                                 |s <- ([(replicate l "tock")|l <- [0..n
                                       +1] ]  ++
137                                     [(replicate l "tock") ++ ["tick"]|
                                           l <- [0..n] ] ) ] ) ]
138     :=: --{traces'TA.skip, r~>l}
139     sort (nub ([t \\ flowActions taSKIP
140            | t <- (traces'TA (n+1) taSKIP )  ]))
141     :=: --{tracesTA.n, r~>l, n<~n+1, tas<~taSKIP}
142     sort (nub (traces_TA  (n+1)  taSKIP   ))
143     :=: --{trans.skip, r->l}
144     sort (nub (traces_TA  (n+1)  (transTA SKIP "ta" 0 0 0 []))))
145     :=: QED
146         where taSKIP = ([["s0", "s1", "s2"], "s0",  ["ck"], ["
                startID0_0"], ["tock", "tick"],
147                           [("s0", "startID0_0", [], [], "s1"),
148                           ("s1", "tock", "ck<=1", "ck", "s1"),
149                           ("s1", "tick", [], [], "s2")], [] ) ]
150 -- Therefore, traces_tockCSP (n+1) SKIP = traces_TA  (n+1)  (transTA
        SKIP)
151
152
153
154
155 {-- Internal choice -------------------------------------------------
156 Induction steps, proof for internal choice is as follows:
157     traces_tockCSP (n+1) (P |~| Q) == traces_TA (n+1) (trans (P |~|Q)
            )
158 assuming
159     traces_tockCSP n (P |~| Q) == traces_TA n (trans (P |~|Q))
160     traces_tockCSP (n+1) P == traces_TA (n+1) (trans P)
161     and traces_tockCSP (n+1) Q == traces_TA (n+1) (trans Q)
162 -}
163
164 proofTrans (IntChoice p1 p2) n =
165     nub (traces_tockCSP (n+1) (IntChoice p1 p2) )
166     :=: --{traces_tockCSP.internalChoice, n<~n+1}
167     nub ((traces_tockCSP (n+1) p1) ++ (traces_tockCSP (n+1) p2))
168     :=: --{Induction hypothesis }
169     nub ((traces_TA  (n+1)  (transTA p1 "ta1" 0 0 0 []))   ++
170         (traces_TA (n+1) (transTA p2 "ta2" 0 0 0 [])))
171     :=: -- {tracesTA.n, l~>r}
```

```
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
        |t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
        |t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] )
    :=: -- {lemma1, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
         |t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
         |t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] ++
         [t \\ (flowActions taIntChoice )
             |t <- (traces'TA (n+1) taIntChoice ) ])
    :=: --{List comprehension}
    nub [t \\ (flowActions ((transTA p1 "ta1" 0 0 0 []) ++
                            (transTA p2 "ta2" 0 0 0 []) ++
                             taIntChoice ) )
                |t <- (traces'TA (n+1)
                           ((transTA p1 "ta1" 0 0 0 []) ++
                           ( transTA p2 "ta2" 0 0 0 []) ++
                             taIntChoice  ) )
        ]
    :=: --{tracesTA.n, r~>l}
    nub (traces_TA (n+1)
              ((transTA p1 "ta1" 0 0 0 []) ++
               (transTA p2 "ta2" 0 0 0 []) ++
              taIntChoice  ))
    :=: --{transTA.internalChoice, r~>l}
    nub ( traces_TA (n+1) (transTA  (IntChoice p1 p2) "ta" 0 0 0 [] )
        )
    :=: QED
        where
            taIntChoice = [(["s0", "s1", "s2", "s3"], "s0",  [],
                    ["startID0_0", "startID0_1", "startID1_1"], [],
                    [("s0", "startID0_0", [], [], "s1"),
                     ("s1", "startID0_1", [], [], "s2"),
                     ("s1", "startID1_1", [], [], "s3"),
                     ("s2", [], [], [], "s0"),
                     ("s3", [], [], [], "s0") ], [] ) ]
-- Therefore, traces_tockCSP n+1 (IntChoice p1 p2) = traces_TA n+1 (
    IntChoice p1 p2)
```

```
{-- External choice ------------------------------------------------
Induction steps, proof for internal choice is as follows:
    traces_tockCSP (n+1) (P [] Q) == traces_TA (n+1) (trans (P [] Q))
assuming
    traces_tockCSP n (P [] Q) == traces_TA n (trans (P [] Q))
    traces_tockCSP (n+1) P == traces_TA (n+1) (trans P)
    and traces_tockCSP (n+1) Q == traces_TA (n+1) (trans Q)
-}


proofTrans (ExtChoice p1 p2) n =
    nub (traces_tockCSP (n+1) (ExtChoice p1 p2) )
    :=: --{traces_tockCSP.externalChoice, n<~n+1}
    nub ((traces_tockCSP (n+1) p1) ++ (traces_tockCSP (n+1) p2))
    :=: --{Induction hypothesis }
    nub ((traces_TA  (n+1)  (transTA p1 "ta1" 0 0 0 []))   ++   (
       traces_TA (n+1) (transTA p2 "ta2" 0 0 0 [])))
    :=: --{tracesTA.n, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
           | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
           | t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] )
    :=: --{Adding empty list}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
        [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] ++ [] )
    :=: --{lemma2, l~>r}
    nub ([t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))] ++
         [t \\ (flowActions (transTA p2 "ta2" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p2 "ta2" 0 0 0 []))] ++
         [t \\ (flowActions taExtChoice )
             |t <- (traces'TA (n+1) taExtChoice ) ])
    :=: --{List comprehension}
    nub [t \\ (flowActions ((transTA p1 "ta1" 0 0 0 []) ++
                            (transTA p2 "ta2" 0 0 0 []) ++
                             taExtChoice ) )
                |t <- (traces'TA (n+1)
                           ((transTA p1 "ta1" 0 0 0 []) ++
                            (transTA p2 "ta2" 0 0 0 []) ++
                             taExtChoice ) ) ]
    :=: --{tracesTA.n, r~>l}
    nub (traces_TA (n+1)
              (taExtChoice ++ (transTA p1 "ta1" 0 0 0 []) ++
              (transTA p2 "ta2" 0 0 0 [])  ))
    :=: --{transTA.externalChoice, r~>l}
    nub ( traces_TA (n+1)
                    (transTA  (ExtChoice p1 p2) "ta3" 0 0 0 [] ) )
```

```
267     :=: QED
268         where
269           taExtChoice =
270                 [([ "s0", "s1", "s2", "s3"],    "s0",  [],
271                   ["startID0_0", "startID0_1", "startID1_1"], [],
272                   [("s0", "startID0_0", [], [], "s1"),
273                    ("s1", "startID0_1", [], [], "s2"),
274                    ("s2", "startID1_1", [], [], "s3"),
275                    ("s3",  [],          [], [], "s0") ], [] ) ]
276 -- Thus, traces_tockCSP n+1 (ExtChoice p1 p2) = traces_TA n+1 (
        ExtChoice p1 p2)
277
278
279
280
281 {-- Sequential composition  ----------------------------------
282 Induction steps, proof for sequential composition is as follows:
283     traces_tockCSP (n+1) (P ; Q) == traces_TA (n+1) (trans (P ; Q))
284 assuming
285     traces_tockCSP n (P ; Q) == traces_TA n (trans (P ; Q))
286     traces_tockCSP (n+1) P == traces_TA (n+1) (trans P)
287     and traces_tockCSP (n+1) Q == traces_TA (n+1) (trans Q)
288 -}
289
290 proofTrans (Seq p1 p2) n =
291     traces_tockCSP (n+1) (Seq p1 p2)
292     :=: --{traces_tockCSP.seq, n<~n+1}
293     (traces_tockCSP (n+1) p1)
294     ++ [(iinit s) ++ t
295         | s <- (traces_tockCSP (n+1) p1),  (ilast s) == "tick",
296           t <- (traces_tockCSP (n - size_p1) p2) ]
297     :=: --{Induction hypothesis }
298     nub ( (traces_TA (n+1) (transTA p1 "ta1" 0 0 0 []) )
299             ++ [(iinit t1) ++ t2
300                 | t1 <- (traces_TA
301                             (n+1)
302                             (transTA p1 "ta1" 0 0 0 []) ),
303                   t2 <- (traces_TA
304                             (n - size_p1)
305                             (transTA p2 "ta2" 0 0 0 []) ) ]  )
306     :=: --{tracesTA.n, 1~>r}
307     nub ( [t \\ (flowActions (transTA p1 "ta1" 0 0 0 []))
308             | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))]
309             ++ [t \\  (flowActions ((transTA p1 "ta1" 0 0 0 [])
310                     ++ (transTA p2 "ta2" 0 0 0 []) ))
311             |t <-  [(iinit t1) ++ t2
312                 | t1 <-  traces'TA
313                             (n+1)
314                             (transTA p1 "ta1" 0 0 0 [] ),
```

```
                              (ilast t1) == "tick",
                    t2 <-  traces'TA
                              (n - size_p1)
                              (transTA p2 "ta2" 0 0 0 [] ) ] ]    )
    :=: --{lemma3, l~>r}
    nub ( [t \\ (flowActions ( transTA p1 "ta1" 0 0 0 []))
        | t <- (traces'TA (n+1) (transTA p1 "ta1" 0 0 0 []))]
         ++ [t \\ (flowActions ((transTA p1 "ta1" 0 0 0 [])
         ++ (transTA p2 "ta2" 0 0 0 [])) )
               |t <- [(iinit t1) ++ t2
                       | t1 <- (traces'TA (n+1)
                              (transTA p1 "ta1" 0 0 0 [] )),
                              (ilast t1) == "tick",
                          t2 <- (traces'TA (n - size_p1)
                              (transTA p2 "ta1" 0 0 0 [] ) ) ]   ]
      ++ [t \\ (flowActions taSeq ) |t <- (traces'TA (n+1) taSeq) ])
    :=: --{List comprehension}
    nub [t \\ (flowActions ((transTA p1 [] 0 0 0 []) ++
                              (transTA p2 [] 0 0 0 []) ++
                               taSeq ) )
                |t <- (traces'TA (n+1)
                             ((transTA p1 [] 0 0 0 []) ++
                              (transTA p2 [] 0 0 0 []) ++
                               taSeq
                               )
                  )
        ]
    :=: --{tracesTA.n, r~>l}
    nub (traces_TA (n+1)
              ((transTA p1 [] 0 0 0 []) ++
               (transTA p2 [] 0 0 0 []) ++
               taSeq  ))
    :=: --{transTA.seq, r~>l}
    nub ( traces_TA (n+1) (transTA  (Seq p1 p2) "ta" 0 0 0 [] ) )
    :=: QED
        where
            taSeq = [(["s0", "s1", "s2", "s3"], "s0",  [],
                          ["startID0_0", "startID0_1", "startID0_2", "
                             FinishID0", "FinishID1"], [],
                          [("s0", "startID0_0", [], [], "s1"), ("s1", "
                             startID0_1", [], [], "s2"),
                           ("s2", "FinishID1", [], [], "s3"),  ("s3", "
                              startID0_2", [], [], "s0") ], [] ) ]
            size_p1 = length ( maximum (traces_tockCSP (n+1) p1) )
{- Therefore,
   traces_tockCSP n+1 (Seq p1 p2) = traces_TA n+1 (Seq p1 p2)
-}
```

```
361
362
363 -- ERROR handling ------------------------------
364 proofTrans    _        _   =  error "to␣be␣completed..."
365
366
367
368 ------ We use the following definitions in the proof -----------
369 --1. Definition of transTA
370 transTA :: CSPproc -> ProcName ->  BranchID -> StartID -> FinishID ->
        UsedNames -> [TA]   -- (node, branch, branch)
371 --transTA :: CSPproc -> [TA]
372 transTA    STOP  procName  bid  sid  fid  usedNames
373            = [([ "s0", "s1"], "s0",  ["ck"], [coodAction], ["tock"],
                    [("s0", coodAction, [], [], "s1"), ("s1", "tock", "ck
                   <=1", "ck", "s1")], [] )]        --trans.stop
374            where
375             coodAction = concat ["startID", show bid, "_", show sid]
376
377 transTA  SKIP  procName  bid  sid  fid  usedNames
378            = [([ "s0", "s1", "s2"], "s0",  ["ck"], [coodAction], ["
                   tock", "tick"],
379               [("s0", coodAction, [], [], "s1"),
380                ("s1", "tock", "ck<=1", "ck", "s1"),
381                ("s1", "tick", [], [], "s2")], [] ) ]   s--trans.skip
382            where
383             coodAction = concat ["startID", show bid, "_", show sid]
384
385 transTA    (IntChoice p1 p2)  procName  bid  sid  fid  usedNames
386            =  [(["s0", "s1", "s2", "s3"], "s0",  [],
387                 ["startID0_0", "startID0_1", "startID1_1", "FinishID0
                      ", "FinishID1"], [],
388                 [("s0", "startID0_0", [], [], "s1"), ("s1", "
                      startID0_1", [], [], "s2"),
389                  ("s2", "FinishID_1", [], [], "s3"), ("s3", "
                       startID1_3", [], [], "s0") ], [] ) ]
390               ++ (transTA  p1  procName  bid      (sid+1)  fid
                      usedNames )
391               ++ (transTA  p2  procName  (bid+1)  (sid+1)  fid
                      usedNames )
392              where
393                coodAction1 = concat ["startID", show bid, "_", show
                      sid]
394                coodAction2 = concat ["startID", show bid, "_", show (
                      sid+1)]
395                coodAction3 = concat ["startID", show (bid+1), "_",
                      show (sid+1)]                    --trans.intChoice
396
397
```

```
transTA  (ExtChoice p1 p2)  procName  bid  sid  fid  usedNames
    =  [([ "s0", "s1", "s2", "s3"],    "s0",  [],
          [ coodAction1, coodAction2, coodAction3], [],
           [("s0", coodAction1, [], [], "s1"),
            ("s1", coodAction2, [], [], "s2"),
            ("s2", coodAction3, [], [], "s3"),
            ("s3", [],          [], [], "s0") ], [] ) ]
        ++ (transTA  p1  procName  bid      (sid+1)  fid  usedNames )
        ++ (transTA  p2  procName  (bid+1)  (sid+1)  fid  usedNames )
  where
    coodAction1 = concat ["startID", show bid,     "_", show sid]
    coodAction2 = concat ["startID", show bid,     "_", show (sid+1)]
    coodAction3 = concat ["startID", show (bid+1), "_", show (sid+1)]
                                                        --trans.extChoice

transTA  (Seq p1 p2) procName  bid  sid  fid  usedNames
    =  [(["s0", "s1", "s2", "s3"], "s0",  [],
          [coodAction0, coodAction1, coodAction2, coodAction3], [],
                [("s0", coodAction0, [], [], "s1"),
                 ("s1", coodAction1, [], [], "s2"),
                 ("s2", coodAction2, [], [], "s3"),
                 ("s3", coodAction3, [], [], "s0") ], [] ) ]
      ++ (transTA  p1  procName  bid  (sid+1)  (fid+1)  usedNames )
      ++ (transTA  p2  procName  bid  (sid+2)   fid     usedNames )
where
    coodAction0 = concat ["startID",  show bid,  "_", show  sid]
    coodAction1 = concat ["startID",  show bid,  "_", show (sid+1)]
    coodAction2 = concat ["finishID", show (fid+1)]
    coodAction3 = concat ["startID",  show bid,  "_", show (sid+2)]
                                                      --trans.seq


--2. Definition of traces_TA
traces_TA :: Int -> [TA] -> [[Event]]
traces_TA    0       _   =  [[]]            -- tracesTA.0
traces_TA    n      tas  =
                    [t \\ (flowActions tas) | t <- (traces'TA n tas)]
                                                    -- tracesTA.n


--3. Definition of traces'TA
traces'TA :: Int -> [TA]    -> [[Event]]
traces'TA    0       _       =  [[]]
traces'TA    n       []      =  [[]]
traces'TA    n    ((["s0", "s1"],  "s0",  ["ck"], [ coodAction ],
                    ["tock"], [("s0",  st, [], [], "s1"),
                    ("s1", "tock", "ck<=1", "ck", "s1")], []):tas)
            = [coodAction:s|s<-[(replicate l "tock")|l <- [0..n] ] ]
                ++ (traces'TA n tas)               -- traces'TA.stop
```

```
445
446 traces'TA    n    (((["s0", "s1", "s2"], "s0",  ["ck"], _,
447                     ["tock", "tick"],
448                     [("s0",  st,    [],      [],   "s1"),
449                      ("s1", "tock", "ck<=1", "ck", "s1"),
450                      ("s1", "tick", [],      [],   "s2")], []):tas)
451            = [(replicate l "tock")|l <- [0..n] ] ++
452                [(st:s ++ ["tick"])|s<-[(replicate l "tock")
453                    |l <- [0..(n-1)] ] ]
454                ++ (traces'TA n tas)           --traces'TA.skip
455
456 traces'TA    n    ( (["s0", "s1", "s2", "s3"], "s0",  [], _, [],
457                      [("s0", coodAction1, [], [], "s1"),
458                       ("s1", coodAction2, [], [], "s2"),
459                       ("s1", coodAction3, [], [], "s3"),
460                       ("s2", [], [], [], "s0"),
461                       ("s3", [], [], [], "s0") ], [] ):tas)
462            = [[coodAction1, coodAction2, coodAction3]]
463              ++ (traces'TA n tas)       -- traces'TA.internalChoice
464
465 traces'TA    n    ( ([ "s0", "s1", "s2", "s3"],    "s0",  [], _, [],
466                      [("s0", coodAction1, [], [], "s1"),
467                       ("s1", coodAction2, [], [], "s2"),
468                       ("s2", coodAction3, [], [], "s3"),
469                       ("s3", [],          [], [], "s0") ], [] ):tas)
470                = [[coodAction1, coodAction2, coodAction3]]
471                    ++ (traces'TA n tas)
472                                          --traces'TA.externalChoice
473
474 traces'TA    n    ( (["s0", "s1", "s2", "s3"], "s0",  [], _, [],
475                     [("s0", coodAction0, [], [], "s1"),
476                      ("s1", coodAction1, [], [], "s2"),
477                      ("s2", coodAction2, [], [], "s3"),
478                      ("s3", coodAction3, [], [], "s0") ], []):tas)
479            = ([coodAction1, coodAction2, coodAction3])
480                :(traces'TA n tas)                 -- traces'TA.seq
481
482
483
484 -- A structure for extending the proof to cover the remaining
       constructs
485 traces'TA    _   _  = [["~~Error~~"]] -- error "pending proof"
486
487 --4. Definition of traces_tockCSP
488 traces_tockCSP :: Int -> CSPproc -> [[Event]]
489 traces_tockCSP    0      _       =  [[]]       -- traces_tockCSP.0
490 traces_tockCSP    n      STOP
491               =  [(replicate l "tock") | l <- [0..n] ]
492                                              -- traces_tockCSP.stop
```

```
493
494 traces_tockCSP    n      SKIP
495                =  [(replicate l "tock") | l <- [0..n] ] ++
496                   [(replicate l "tock") ++ ["tick"]
497                       | l <- [0..(n-1)] ]    --traces_tockCSP.skip
498
499 traces_tockCSP    n      (IntChoice p1 p2 )
500               =  (traces_tockCSP n p1) ++
501                  (traces_tockCSP n p2)
502                               -- traces_tockCSP.internalChoice
503
504 traces_tockCSP    n      (ExtChoice p1 p2 )
505               =  (traces_tockCSP n p1) ++ (traces_tockCSP n p2)
506                               -- traces_tockCSP.externalChoice
507
508 traces_tockCSP    n      (Seq p1 p2 )
509               =  if (n > size_p1)
510                  then (traces_tockCSP n p1) ++
511                     [(iinit t1) ++ t2
512                         | t1 <- (traces_tockCSP n p1),
513                                 (ilast t1) == "tick",
514                          t2 <- (traces_tockCSP (n - size_p1)  p2) ]
515                  else (traces_tockCSP n p1)
516         where
517             size_p1 = length (maximum (traces_tockCSP (n+1) p1) )
518                                                 -- traces_tockCSP.seq
519
520
521 --5. Definition of flowActions
522 flowActions :: [TA]                          -> [String]
523 flowActions    []                            =  []
                                                  -- flowActions.0
524 flowActions    [(_, _, _, x, _, _, _)]       =  x      -- flowActions.1
525 flowActions    ((_, _, _, x, _, _, _):ts)
526         = concat [x, (flowActions ts)]                -- flowActions.n
527
528
529 {- A lemma for the binary constructors to show that the traces of the
       connecting TA are empty after removing the connections actions.
      the lemma established that if the traces ts contains a trace (like
       ts is empty in this case), so adding the same trace will not
      change the value of the traces -}
530
531 -- A lemma for the connection TA of the internal choice
532 lemma1 n = nub ts                                        -- lemma1
533         :=:
534         nub (ts ++ [t \\ (flowActions [(["s0", "s1", "s2", "s3"],
535              "s0", [], ["startID0_0", "startID0_1", "startID1_1"],
                    [], [("s0", "startID0_0", [], [], "s1"),
```

```
              ("s1", "startID0_1", [], [], "s2"),
              ("s1", "startID1_1", [], [], "s3"),
              ("s2", [],           [], [], "s0"),
              ("s3", [], [], [], "s0") ], [] ) ] )
              |t <- (traces'TA n
                      [(["s0", "s1", "s2", "s3"], "s0",  [],
                      ["startID0_0", "startID0_1", "startID1_1"], [],
                      [("s0", "startID0_0", [], [], "s1"),
                       ("s1", "startID0_1", [], [], "s2"),
                       ("s1", "startID1_1", [], [], "s3"),
                       ("s2", [],           [], [], "s0"),
                       ("s3", [],           [], [], "s0") ], [] ) ]
                    ) ]
           )
        :=:
         [[]]
        :=:
         QED
            where
                ts = [[]]


-- A lemma for the connection TA of the external choice
lemma2 n = nub ts                                          -- lemma2
    :=:
    nub (ts ++
        [t \\ (flowActions
                [([ "s0", "s1", "s2", "s3"], "s0",  [],
                  ["startID0_0", "startID0_1", "startID1_1"], [],
                  [("s0", "startID0_0", [], [], "s1"),
                   ("s1", "startID0_1", [], [], "s2"),
                   ("s2", "startID1_1", [], [], "s3"),
                   ("s3", [], [], [], "s0") ], [] ) ] )
         |t <- (traces'TA n [([ "s0", "s1", "s2", "s3"],"s0",  [],
                   ["startID0_0", "startID0_1", "startID1_1"], [],
                   [("s0", "startID0_0", [], [], "s1"),
                    ("s1", "startID0_1", [], [], "s2"),
                    ("s2", "startID1_1", [], [], "s3"),
                    ("s3", [],           [], [], "s0") ], []) ]
                ) ]
        )
    :=:
    [[]]
    :=:
     QED
    where
        ts = [[]]
```

```
585
586 -- A lemma for the connection TA of the sequential composition
587 lemma3 n = nub ts                                              -- lemma3
588         :=:
589         nub (ts ++
590         [t \\ (flowActions
591                  [(["s0", "s1", "s2", "s3"], "s0", [],
592                    ["startID0_0", "startID0_1", "startID0_2", "
                          FinishID0", "FinishID1"], [],
593                     [("s0", "startID0_0", [], [], "s1"),
594                      ("s1", "startID0_1", [], [], "s2"),
595                      ("s2", "FinishID1",  [], [], "s3"),
596                      ("s3", "startID0_2", [], [], "s0") ], [] )])
597                       |t <- (traces'TA n
598                                [(["s0", "s1", "s2", "s3"], "s0", [],
599                                ["startID0_0", "startID0_1",
600                                 "startID0_2", "FinishID0",
601                                 "FinishID1"], [],
602                                [("s0", "startID0_0", [], [], "s1"),
603                                 ("s1", "startID0_1", [], [], "s2"),
604                                 ("s2", "FinishID1",  [], [], "s3"),
605                                 ("s3", "startID0_2", [], [], "s0")],
606                                [] ) ] ) ] )
607         :=:
608          [[]]
609         :=:
610          QED
611             where
612                ts = [[]]
613
614
615 -- ilast is similar to the function last with the capability of
       handling empty trace
616 ilast :: [Event] -> Event
617 ilast    []      =  []          -- ilast.0
618 ilast    xs      =  last xs     -- ilast.1
619
620
621 -- iinit is similar to the function init with the capability of
       handling empty trace
622 iinit :: [Event] -> [Event]
623 iinit    []      =  []          -- iinit.0
624 iinit    xs      =  init xs     -- iinit.1
625
626
627 -- list_interleave
628 list_interleave :: [Event] -> [Event] -> [Event]
629 list_interleave    []         ys      =  ys          -- interleave.0
630 list_interleave    (x:xs)     ys      =  (x:(list_interleave ys xs))
```

```
                                                        -- interleave.1
631
632 tracesInterleave t1 t2
633   = nub (concat [[x++y, (list_interleave x y)] | x <- t1, y <- t2 ] )
                                                        -- traceInterleave.1
634
635 -- Example
636 traceInterleave_eg
637   = tracesInterleave [[], ["a"], ["a", "b"]]  [[], ["x"], ["x", "y"]]
638
639
640 -- Data type for the CSP process ------------------------------------
641 data CSPproc  =  STOP
642                | SKIP
643                | WAIT         Int
644                | Prefix       Event     CSPproc
645                | IntChoice    CSPproc   CSPproc
646                | ExtChoice    CSPproc   CSPproc
647                | Seq          CSPproc   CSPproc
648                | Interleave   CSPproc   CSPproc
649                | GenPar       CSPproc   CSPproc     [Event]
650                | Interrupt    CSPproc   CSPproc
651                | Exception    CSPproc   CSPproc     [Event]
652                | Timeout      CSPproc   CSPproc Int
653                | Hiding       CSPproc   [Event]
654                | Rename       CSPproc   [(Event, Event)]
655                | EDeadline    Event     Int
656                | ProcID       String
657
658
659 type TA = ([State], State, [Clock], [Action], [Action] , [(State,
       Action, Clock, Invariant, State)], [Invariant] )
660
661 type ProcName    =  String               -- An identifier for each TA
662 type BranchID    =  Int                  -- An index for the braches
663 type StartID     =  Int                  -- An index for the start
       event
664 type FinishID    =  Int                  -- An index for the finish
       event
665 type SyncPoint   = (Event, String)       -- Assign an identifier for
       a sync point
666 type UsedNames   = [String]              -- List of the names used in
        developing the translation rules.
667 type State       =  String
668 type Clock       =  String
669 type Action      =  String
670 type Invariant   =  String
671 type Event       =  String
672 type NamedProc   =  String
```

References

[1] Andreas Angerer, Remi Smirra, Alwin Hoffmann, Andreas Schierl, Michael Vistein, and Wolfgang Reif. A Graphical Language for Real-Time Critical Robot Commands. *Proc. Third Int. Work. Domain-Specific Lang. Model. Robot. Syst. (DSLRob 2012)*, 2012.

[2] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.

[3] Lucas Liebenwein, Cenk Baykal, Igor Gilitschenski, Sertac Karaman, and Daniela Rus. Sampling-based approximation algorithms for reachability analysis with provable guarantees. *Proceedings of Robotics: Science and Systems 2018*, 2018.

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

[5] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL Over 15 Years. *Software: Practice and Experience*, 41(2):133–142, 2011.

[6] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2016.

[7] Shimon Y Nof. *Handbook of industrial robotics*. John Wiley & Sons, 1999.

[8] Robert Bogue. Strong prospects for robots in retail. *Industrial Robot: the international journal of robotics research and application*, 2019.

[9] E. Demaitre. Five robotics predictions for 2016. Robotics Business Review." (2016)., 2016. Available at `https://tinyurl.com/y3odb9zq` (Accessed: 19th October, 2020).

[10] Selma Kchir, Saadia Dhouib, Jeremie Tatibouet, Baptiste Gradoussoff, and Max Da Silva Simoes. RobotML for industrial robots: Design and simulation of manipulation scenarios. In *IEEE Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, 2016.

[11] Alvaro Miyazawa, Ana Cavalcanti, Pedro Ribeiro Wei Li, Jim Woodcock, and Jon Timmis. RoboChart Reference Manual. Technical report, University of York, 2016.

[12] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

[13] S Alexandrova, Z Tatlock, and M Cakmak. RoboFlow: A flow-based visual programming language for mobile manipulation tasks. *2015 IEEE Int. Conf. Robot. Autom.*, pages 5537–5544, 2015.

[14] Hoang Pham. *Software reliability*. Springer Science & Business Media, 2000.

[15] Christian Schlegel, Andreas Steck, and Alex Lotz. Robotic software systems: From code-driven to model-driven software development. *Robotic Systems-Applications, Control and Programming*, pages 473–502, 2012.

[16] John Peterson, Paul Hudak, and Conal M. Elliott. Lambda in motion: Controlling robots with Haskell. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 1999.

[17] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Oreback. Towards component-based robotics. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 163–168. IEEE, 2005.

[18] Issa Nesnas, Richard Volpe, Tara Estlin, Hari Das, Richard Petras, and D Mutz. Toward developing reusable software components for robotic applications. *Proc. 2001 IEEE/RSJ Int. Conf. Intell. Robot. Syst. Expand. Soc. Role Robot. Next Millenn. (Cat. No.01CH37180)*, 2001.

[19] Saddek Bensalem, Matthieu Gallien, Félix Ingrand, Imen Kahloul, and Nguyen Thanh-Hung. Designing autonomous robots. *IEEE Robotics & Automation Magazine*, 16(1):67–77, 2009.

[20] Geoffrey Biggs and Bruce Macdonald. A Survey of Robot Programming Systems. *Proc. Australas. Conf. Robot. Autom.*, pages 1–3, 2003.

[21] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[22] Arne Nordmann, Nico Hochgeschwender, and Sebastian Wrede. A Survey on Domain-Specific Languages in Robotics. *J. Softw. Eng. Robot.*, 1(July):195–206, 2016.

[23] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic Property Checking of Robotic Applications. *IEEE/RSJ Int. Conf. Intell. Robot. System*, 2017.

[24] Anthony M. Sloane. *Software Abstractions: Logic, Language, and Analysis by Daniel Jackson, The MIT Press, 2006, 366pp, ISBN 978-0262101141.* 2009.

[25] Anthony Mallet and Matthieu Herrb. Recent developments of the genom robotic component generator. In *6th National Conference on Control Architectures of Robots*, pages 4–p, 2011.

[26] F Fleurey and A Solberg. A Domain Specific Modeling Language supporting Specification, Simulationand Execution of Dynamic Adaptive Systems. *Model. ACM/IEEE 12th Int. Conf. Model. Eng. Lang. Syst.*, pages 606–621, 2009.

[27] Paul Black, Mark Badger, Barbara Guttman, and Elizabeth Fong. Dramatically reducing software vulnerabilities: Report to the white house office of science and

technology policy. Technical report, National Institute of Standards and Technology, 2016.

[28] John Fitzgerald, Juan Bicarregui, Peter Gorm Larsen, and Jim Woodcock. Industrial deployment of formal methods: Trends and challenges. In *Industrial Deployment of System Engineering Methods*, pages 123–143. Springer, 2013.

[29] Paul Black, Larry Feldman, Gregory Witte, et al. Dramatically reducing software vulnerabilities. Technical report, National Institute of Standards and Technology, 2017.

[30] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Félix Ingrand, and Anthony Mallet. Model checking real-time properties on the functional layer of autonomous robots. In *International Conference on Formal Engineering Methods*, pages 383–399. Springer, 2016.

[31] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

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

[33] Jim Woodcock and Jim Davies. *Using Z: Specification Refinement and Proof*. Prentice Hall International, 1996.

[34] Daniel Jackson. *Software abstractions*, volume 2. MIT press Cambridge, 2006.

[35] Rajeev Alur. *TECHNIQUES FOR AUTOMATIC VERIFICATION OF REAL-TIME SYSTEMS*. PhD thesis, 1991.

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

[37] Kun Wei, Jim Woodcock, and Ana Cavalcanti. Circus time with reactive designs. In *International Symposium on Unifying Theories of Programming*, pages 68–87. Springer, 2012.

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

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

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

[41] J.A Brzozowski and C.J.H. Seger. Advances in asynchronous circuit theory. Part I : gate and unbounded inertial delay. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS, Vol. 42(1990), No. October, p. 198-249*, 1990.

[42] Patricia Bouyer. An introduction to timed automata. 2011. A lecture note that is available at `www.lsv.fr/~bouyer/files/mpri1112.pdf` (Last accessed 21 October 2020).

[43] 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.

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

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

[46] GHC Team. Ghc user's guide documentation, 2018.

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

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

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

[50] 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.

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

[52] 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.

[53] 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.

[54] Anders Hessel, Kim G Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Formal methods and testing*, pages 77–117. Springer, 2008.

[55] Craig Schlenoff, E Prestes, PJ Sequeira Gonçalves, Mathieu Abel, Yacine Amirat, S Balakirsky, ME Barreto, JL Carbonera, A Chibani, S Rama Fiorini, et al. Ieee standard ontologies for robotics and automation. 2015.

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

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

[58] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.