Analyzing a χ Model of a Turntable System using Spin, CADP and Uppaal

E. Bortnik [a], N. Trčka [b], A.J. Wijs [c,1], S.P. Luttik [b,c],
J.M. van de Mortel-Fronczak [a], J.C.M. Baeten [b],
W.J. Fokkink [c,d], J.E. Rooda [a]

[a] *Department of Mechanical Engineering, Eindhoven University of Technology,P.O.Box 513, 5600 MB Eindhoven, The Netherlands*

[b] *Department of Mathematics and Computer Science, Eindhoven University of Technology,P.O.Box 513, 5600 MB Eindhoven, The Netherlands*

[c] *Department of Software Engineering, CWI,P.O.Box 94079, 1090 GB Amsterdam, The Netherlands*

[d] *Vrije Universiteit Amsterdam, Department of Theoretical Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

Abstract

Nowadays, due to increasing system complexity and growing competition and costs, industry makes high demands on powerful techniques used to design and analyze manufacturing systems. One of the most popular techniques to do performance analysis is simulation. However, simulation-based analysis becomes insufficient since it cannot guarantee the correctness of a system. Furthermore, it is not suitable for functional analysis. Our research focuses on examining other methods to do performance analysis and functional analysis, and trying to combine the two. One of the approaches is to translate a simulation model that is used for performance analysis to a model written in an input language of an existing verification tool. We translate a χ [1] simulation model of a turntable system into models written in the input languages of the tools CADP [2], Spin [3] and Uppaal [4] and do a functional analysis with each of them. This allows us to evaluate the usefulness of these tools for the functional analysis of χ models. We compare the input formalisms, the expressiveness of the temporal logics, and the algorithmic techniques for model checking, that are used in those tools.

[1] Corresponding author. Tel.: +31-20-592-4165; fax: +31-20-592-4199.
E-mail address: A.J.Wijs@cwi.nl (A.J.Wijs).

1 Introduction

The χ language is a modeling and simulation language for the specification of industrial systems. It can be used for creating discrete-event, continuous or combined, so-called hybrid, models. The language and simulator have been successfully applied to a large number of industrial cases, such as an integrated circuit manufacturing plant, a brewery and process industry plants [5]. Simulation is a powerful technique for performance analysis, like calculating throughput and cycle time, but for functional analysis (verification) it is less suitable. It can, for instance, show that (a model of) a system has a deadlock but it cannot show that the system is deadlock-free. For the purpose of verification first the discrete-event part of χ has been formalized [6]. The language was mapped onto the very expressive, process algebra like, language called χ_σ for which an operational semantics was defined and a state space generator has been built [6]. Recently, a new formalization of χ, including the hybrid part, resulted in a more elegant language [7]. The discrete-event part of this language is very similar to χ_σ [8].

The main goal of the TIPSy project [9] (Tools and Techniques for Integrating Performance Analysis and System Verification) is to combine performance analysis with verification, particularly in the χ environment. At the start of this project we are focusing on verification. There is no tool support for the new version of χ yet and the current toolset for χ_σ is a prototype, meant only for educational purposes. Therefore it is not comparable, when it comes to state space generation, to more developed toolsets. Since we do not expect that a dedicated tool for χ, that would be able to compete with existing optimized model checkers, could be built within reasonable time, our aim is to translate χ models to input languages of other existing tools. While doing this, we want to compare input formalisms of different tools and see which are best suited for translating χ models to. We also want to investigate the expressiveness of temporal logics and algorithmic techniques for model checking that are used in those tools.

For this paper, we choose the well-known specification and verification tools CADP [2], SPIN [3] and UPPAAL [4]. There are several reasons why we make this choice:

(1) The three tools are quite popular and have been used to detect design errors in applications from many different domains.

(2) Each tool has a different input language. We use μCRL [10] as the modeling language for CADP. It is an action-based, process algebraic (ACP[11,12,13]) language with excellent data support. SPIN's input language, PROMELA, is a state-based, imperative language. Finally, UPPAAL's input language is a specific class of timed automata, combining

both action-based and state-based features.

(3) Each tool handles time differently.

(4) Each tool has a different logic for expressing properties of a model. In CADP, regular alternation-free μ-calculus [14] is applied, while Spin and Uppaal use a temporal logic, LTL and TCTL [15] respectively.

(5) Each tool uses a different strategy for verification. In CADP (with μCRL as input) the whole state space must be built. Spin does model checking on-the-fly. Uppaal checks invariant and liveness properties by on-the-fly exploration of the state space of a system in terms of symbolic states represented by constraints.

Our case study is a turntable device, a rotating drilling machine. We choose this particular case study because:

(1) It is not too complex; otherwise it would take the emphasis away from translating and comparing and make the modeling unnecessarily difficult.

(2) It is complex enough in the sense that it contains many interesting features to model, such as parallelism and time.

(3) It is a case study that has been used before [6], making it possible for us to look at existing models and extend them.

(4) We have access to a physical turntable system and we can use it to perform physical experiments.

In this document, we show how the turntable model can be mapped to the input languages of the mentioned tools and how it can be verified in those environments. We do not cover translations of general χ models and rather focus on the turntable only, but it should be clear that the same story holds for a large class of χ specifications. Of course, models resulting from a translation of χ models might be very different from those made from scratch. Our aim is to have translations resemble the original χ model closely so that possible verification errors in these translations can be related back to the original model. We show that many interesting properties of the turntable can be verified but that none of the three tools can easily express all of them. We also compare experiences of working with the tools and results such as the number of states generated.

The structure of the document is as follows: First, the turntable device is explained. Then, we give an introduction to χ and present the model of the turntable. The next three sections are devoted to each tool. We give an overview of the input language and the verification mechanism, we explain how we deal with the translation problems and we present the verification of the turntable in detail. The last section gives some comparisons and conclusions.

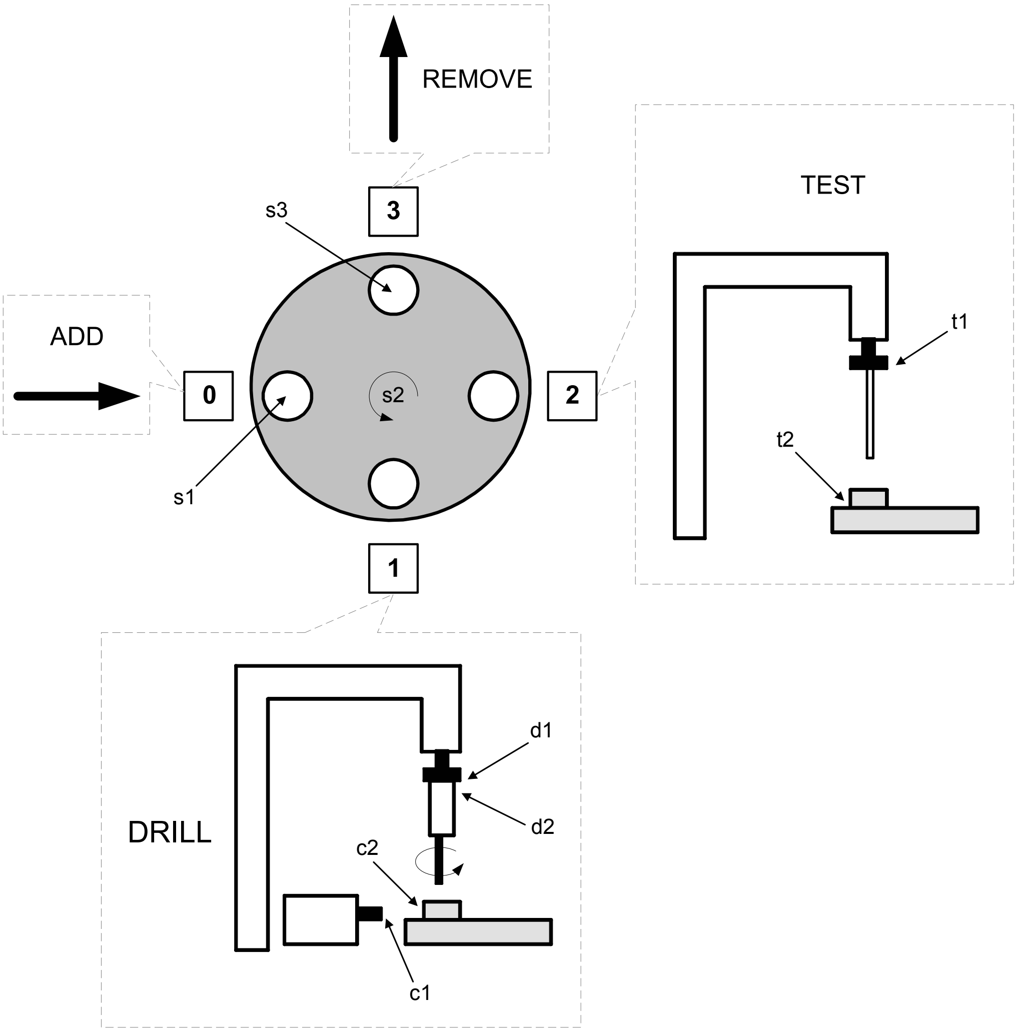


Fig. 1. The turntable system

2 Turntable description

The turntable system is an example of a real-life manufacturing system that is used for (real-time) control research [6,16,17].

The turntable system consists of a round turntable, a clamp, a drill and a testing device (Figure 1). The turntable transports products to the drill and the testing device. The drill drills holes in the products. After drilling a hole the products are delivered to the tester, where the depth of the hole is measured, since it is possible that drilling went wrong. To control the turntable system, sensors and actuators are used. A sensor detects a physical phenomenon, and changes its state. The controller reads the state of the sensor, and sends output to actuators. The actuators translate output from the controller to a physical change in the machine.

The turntable has four slots that can hold a product. Each slot can hold at most one product and can be in input, drill, test or output position. There are three sensors attached to the turntable: the sensor $s1$ at the input position (to detect if a product has been added by the environment), the sensor $s3$ in the output position (to detect if a product has been removed by the environment) and the sensor $s2$ that detects whether the turntable has completed the turn.

The drilling module consists of the drill and the clamp. Every product should be locked before drilling and unlocked afterwards. To detect whether the clamp is locked or not two sensors are used ($c1$ and $c2$ respectively). The drill also has two sensors to detect whether the drill is in its up ($d1$) or down ($d2$) position. These sensors are located above the surface of the turntable, so it is not possible to say whether the product has been drilled successfully or not.

In the testing position there are two sensors to detect whether the tester has reached its up ($t1$) or down ($t2$) position. If the tester has reached its down position the test result of the product is good and if the sensor at the down position did not send a signal during a certain amount of time the test result of the product is bad.

The turntable control system consists of the main controller, turntable controller, drill controller, and tester controller. The main controller supervises the other controllers and the environment. It stores current information about products and operations being performed and based on this information it issues commands to the other controllers and the environment to start operations. When operations are completed the main controller updates the information about the products.

The turntable controller gets signals from the turntable sensors and passes them to the main controller. It also starts rotation of the turntable at the command of the main controller.

The drill controller supervises the drill and the clamp. It switches the drill on/off and commands to lock/unlock the clamp or to start or stop drilling. The drill controller also gets signals from the drill and clamp sensors.

The test controller sends a signal to the tester to start the operation. Then it waits for a signal from the sensor at the down position. If the hole is not deep enough, the sensor is not activated and the current product should be rejected.

The operation-routing sequence of each product is following: add a product to the input position, make a turn (now product is in the drilling position), lock the clamp, switch on the drill, drill, switch off the drill, unlock the clamp, make another turn (now product is in the test position), test, and make a turn again (product is in the removing position).

No product can be added if the adding slot is not empty. No drilling, testing or removing can be performed if the corresponding slot is empty. The turntable can treat up to four products at the same time, that means that the operations can be done in parallel.

Design rules and assumptions Creating the model we consider only "good weather" behavior, i.e. the assumption is that the system works without faults and there is no product loss. The initial state is defined as follows: all slots are empty and no operation is started.

For reasons of simplicity, we decided to concentrate on the control system. That means that we do not model material flow as this information can be obtained from the information stored by the main controller.

We assume that the main controller sends messages to the environment to allow adding and removing of products and the environment informs the main controller when the operations are completed. The environment can skip the adding or removing operations. A product can be removed from the removing position only if it has been drilled properly. If a product has a good test result and it has not been removed, it should not be drilled and tested again. If a product has a bad test result it must be drilled and tested again. That means that the information whether product has been added or removed is necessary only after the rotation of the turntable.

When the other sensors change their states, the control system must be notified immediately. For instance, if the clamp sensor does not report that the clamp is locked, the drill cannot start drilling. This difference causes different implementation of the sensors. The turntable sensor states are checked by the control system just before a turn, while the other sensors inform the control system about their state changes immediately.

We also assume that the order of starting and ending of the adding, drilling, testing and removing operations is not known in advance.

The execution of each turntable operation requires a certain amount of time. Because the duration of the turntable operations has not been defined anywhere, we have decided to use the delays, that have been defined in other turntable models, like [6]. We assume that the environment needs 2 time units to perform adding or removing of a product. The clamp needs 2 time units to lock or unlock a product. The drilling operation takes 3 time units and returning the drill to its up position takes 2 time units. Testing and returning the tester to its initial (up) position require 2 time units each.

Verification properties Traditionally, verification properties have been classified into safety and liveness properties. Safety is usually defined as a set of properties that the system may not violate, while liveness is defined as the set of properties that the system must satisfy [3]. Safety, then, defines that something bad will never happen, and liveness defines that eventually something good will happen.

Given those assumptions we want to verify the following properties:

(1) The system does not contain a deadlock, i.e. it cannot come to a state from which it cannot continue operating (safety).
(2) If drilling (testing, adding or removing) is started then it is also finished and the turntable doesn't rotate in the meantime (liveness).
(3) If the product has a bad test result then the product remains on the table and is drilled again (liveness).
(4) If the product has a good test result then the remover will be called to remove the product (liveness).
(5) No drilling (testing or removing) takes place if there is no product in the slot and no adding is performed if there is a product in the slot (safety).
(6) Every added product is drilled in the next rotation (liveness).
(7) Every product eventually leaves the table (liveness).
(8) When a product is added it takes between 21 and 30 time units to get its test result (liveness).

The property 7 is a liveness property that requires a *fairness* principle, which makes this property the most complicated one.

First, a product can be removed only if it has a good test result. However, the remover can always decide not to remove and the tester can always generate bad test results. This can happen because the choices whether the product will be removed and whether the test result of the product is good or bad are non-deterministic. In order to verify this property we must put some notion of fairness to the verification process, i.e. exclude unfair paths, in which a product yields a bad test result infinitely often.

Second, since there are at most four products on the table it can happen that one of the products stays on the table while the other ones are drilled properly and removed. In order to verify that every product will eventually be removed we must identify them in some way. The most common solution is to give colors to the products, for instance, *red* and *white*, and change the adder so that it adds (non-deterministically) zero or more *white* products, then one *red*, and then again zero or more *white* ones. We want to make sure that if a *red* product is added then a *red* one will leave the table eventually. Another solution would be to assign unique identifiers to products or use some other way to distinguish them.

The fairness constraints can be expressed syntactically in linear temporal logic (like PLTL), but not in branching temporal logic (like CTL). In μ-calculus fairness properties can be expressed very efficiently [14].

The last property (so-called *bounded liveness*) also requires identification of the products. First we calculate manually the time interval within which a test result of a product is known based on the assumptions. After that we check this interval automatically.

3 The turntable model in χ

3.1 The χ language

The χ language was designed as a *hybrid*, modeling and *simulation* language. Since we are interested only in discrete-event models and verification, we present here just a part of the language, disregarding features that are used for simulation and to model hybrid behavior. For a complete reference of χ, see [1]. The discrete-event subset of χ is described in [18].

Data types The χ language is statically strongly typed. Every variable has a type which defines the allowed operations on that variable. The basic data types are boolean, natural, integer and real numbers and enumerations. The language provides a mechanism to build sets, lists, array tuples, record tuples, dictionaries, functions, and distributions (for stochastic models). Channels also have a type that indicates the type of data that is communicated via the channel.

Time model The time in χ is dense, i.e. timing is measured on a continuous time scale. The weak time determinism principle, or sometimes called the time factorization property, (time doesn't make a choice) and maximal progress (a process can delay only if it cannot do anything else) are implicit. The time additivity (if a process can delay first t_1 and then immediately following t_2 time units then it can delay $t_1 + t_2$ time units from the start) is not present. Delaying is enforced by the delay operator but some processes can also implicitly delay, e.g. send.

Atomic processes The atomic processes of χ are process constructors and they cannot be split into smaller processes. They are:

(1) The assignment process ($x := e$). It assigns the value (must be defined) of expression e to variable x. It doesn't have the possibility to delay.
(2) The skip process. It performs the internal action τ and cannot delay.
(3) The send process ($m\,!\,e$). It sends the value of the expression e via channel m. The value of e must be defined and of the right type. It is able to delay arbitrarily long.
(4) The receive process ($m\,?\,x$). It receives a value via the non-empty channel m and assigns it to the variable x which must be of the right type. It is also able to delay arbitrarily long.
(5) The delay process ($\Delta\, e$). It delays a number of time units equal to the value of the expression e or less. The value of e must be a positive real number.

Communication model Communication in χ is synchronous, meaning that a *send* and a *receive* action on a same channel cannot happen individually but only together, as a communication action.

Operators Atomic processes can be combined by means of the following operators. We present each one of them together with their (informal) semantics.

(1) The guard operator ($\rightarrow$). A process $b \rightarrow p$ behaves as p if the value of the boolean expression (guard) b is *true*, otherwise it deadlocks.
(2) The alternative composition operator ($[\!]$). A process $p \;[\!]\; q$ represents a non-deterministic choice between p and q.
(3) The sequential composition operator (;). A process $p;\, q$ behaves as p followed by the process q.
(4) The repetition operator ($*$). A process $*p$ behaves as p infinitely many times.
(5) The parallel operator ($\|$). A process $p \parallel q$ executes p and q concurrently in an interleaved fashion, i.e. the actions of p and q are executed in arbitrary order. If one of the processes can execute a *send* action and the other one can execute a *receive* action on the same channel then they communicate, in other words $p \parallel q$ executes the communication action on this channel.
(6) The scope operator ($[\![\ |\ ]\!]$). A process $[\![\, s \mid p \,]\!]$ behaves as p in a local state s. The state s is used to define local variables and channels visible only to the process p. It is recursively defined as the empty state or as dcl, s' where s' is a state and dcl is a variable declaration ($x : type[= val]$) or a channel declaration ($m : ?\,type$ for receiving and $m : !\,type$ for sending).

Process definitions The language χ provides the possibility to define processes. We don't give a syntax definition here but rather an example:

$$\mathsf{proc}\ p(c : ?\,\mathsf{nat}\ ,\ b : \mathsf{bool}\ ) = [\![\ x : \mathsf{nat}\ \mid b \rightarrow c\,?\,x\ ]\!]$$

The process p has two arguments, a channel c that can transport natural numbers and a boolean variable b. It has only one local variable, x. The process can now be instantiated (e.g. $p(m, y > 7)$) inside another process.

3.2 The turntable model

The turntable system architecture is depicted in Figure 2. The mechanical components are represented by means of the processes *tester*, *drill*, *clamp* and *turn_table*. These components are controlled by switching commands: *cDrillOnOff* switches the drill on/off, *cDrillUpDown* instructs the drill to start or stop drilling, *cClampOnOff* instructs the clamp to lock or unlock the product, and *cTesterUpDown* instructs the tester to start or stop testing. The other signals that are used are *cRotate* (commands the turntable to start turning), *cEnvCanAdd, cEnvCanRemove* (inform the environment that it can perform adding or removing operations respectively). As already mentioned, the sensors are implemented in several ways (more explanations are given in the descriptions of the corresponding processes).

The control system model consists of the main controller, drill and clamp controller, tester controller and turntable controller which are modeled by means of the processes *main_control*, *drill_control*, *tester_control* and *TTC* respectively. The processes *env_add* and *env_remove* represent the environment.

Below we explain all processes in detail. Of each process, a description is given followed by the χ code that models the component.

The *turn_table* process In the *turn_table* process we define three boolean variables representing the turntable sensors. The variables *bS1* and *bS3* correspond to the sensors at the adding and removing positions respectively. The variable *bS2* corresponds to the turntable sensor that detects whether the turntable is rotating or not. The current states of the sensors are sent via channels *cS1, cS2, cS3*. The sensor states are updated by the environment when a product is added or removed (*cEnvAdded, cEnvRemoved*). In the real system the states of these sensors are automatically updated while turning. To achieve this we add two more channels (*cUpdateS1, cUpdateS3*). A different way to model a change of the sensor states can be found later in the description of the *main_control* process. When the *turn_table* gets the signal *cRotate* it performs a delay. Reading and updating the sensors are 'atomic' and instantaneous actions. The control system is modeled in such a way that it is not possible to perform those actions in parallel. This allows us to use alternative composition instead of the parallel one and reduce the state space.

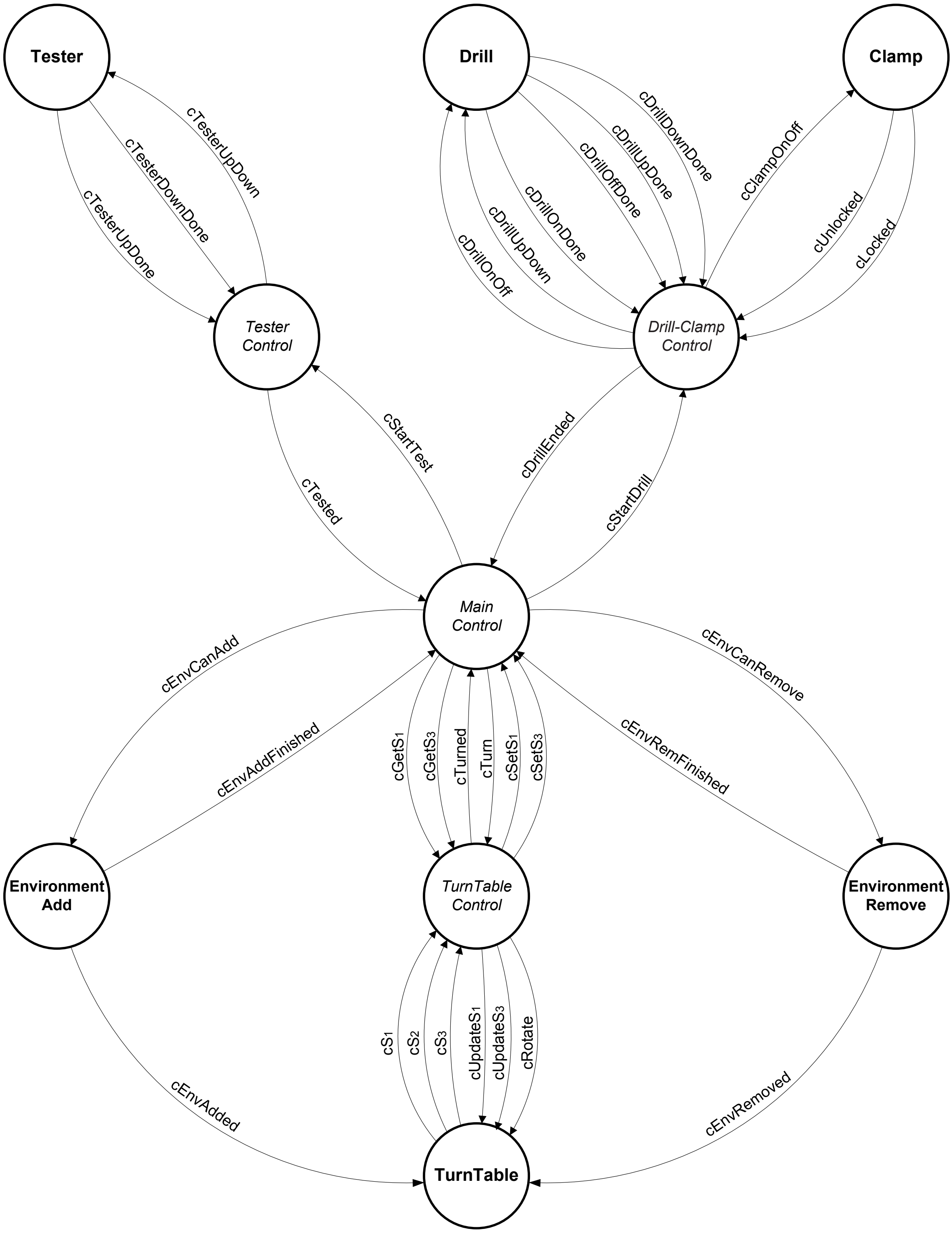


Fig. 2. The turntable model architecture

```
proc Turn_table( cEnvAdded, cEnvRemoved
               , cRotate, cUpdateS1, cUpdateS3 : ?bool
               , cS1, cS2, cS3 : !bool
               )=
|[ bS1 : bool = false, bS2 : bool = false, bS3 : bool = false, x : bool
| ∗( cS1 ! bS1
   [] cS2 ! bS2
   [] cS3 ! bS3
   [] cEnvAdded ? bS1
   [] cEnvRemoved ? x; bS3 := false
   [] cUpdateS1 ? bS1
   [] cUpdateS3 ? bS3
   [] cRotate ? bS2; Δ4.0; bS2 := false
   )
]|
```

The *clamp* process The clamp has one actuator that is used to switch it on/off (*cClampOnOff*). The clamp also has two sensors to detect if it is locked or unlocked. When the states of the sensors are changed, the process *clamp* reports to the control system via the channels *cLocked* and *cUnlocked.*

```
proc Clamp( cClampOnOff : ?bool
          , cLocked, cUnlocked : !bool
          )=
|[ x : bool
| ∗( cClampOnOff ? x; Δ2.0; cLocked ! true
   ; cClampOnOff ? x; Δ2.0; cUnlocked ! true
   )
]|
```

The *drill* process The drill is controlled by two independent actuators. One of the actuators is used to switch the drill on/off (*cDrillOnOff*). The other one (*cDrillUpDown*) instructs the drill to start drilling or to return in its initial (up) position. The states of the sensors are detected through the channels *cDrillDownDone*, *cDrillUpDone.* The commands are handled independently, that is why we use the parallel composition in the *drill* process. On the other hand, both actuators are the parts of the one physical component (the drill) and that is why we do not represent them by means of two separate χ processes, instead, we combine them into one process.

```
proc Drill( cDrillUpDown, cDrillOnOff : ? bool
          , cDrillUpDone, cDrillDownDone
          , cDrillOnDone, cDrillOffDone : ! bool
          )=
|[ x : bool
| ∗( cDrillUpDown ? x;  Δ3.0;  cDrillDownDone ! true
   ; cDrillUpDown ? x;  Δ2.0;  cDrillUpDone ! true
   )
|| ∗( cDrillOnOff ? x;  cDrillOnDone ! true
   ; cDrillOnOff ? x;  cDrillOffDone ! true
   )
]|
```

The *tester* process The tester is controlled by one actuator (*cTesterUpDown*) that is used to start or stop testing. It has two sensors as well. One of them is used to detect a test result of a product. The other one detects whether the tester is in its initial (up) position. Possible test results are implemented by non-deterministic choice. When the test result of a product is good the process *tester* sends a signal via the channel *cTesterDownDone*. Otherwise, the process *tester* waits for the command to move up to the initial position (*cTesterUpDown*) and, then, sends a signal through the channel *cTesterUpDone*.

```
proc Tester( cTesterUpDown : ? bool
           , cTesterUpDone, cTesterDownDone : ! bool
           )=
|[ x : bool
| ∗( cTesterUpDown ? x
   ; Δ2.0
   ; (cTesterDownDone ! true [] skip)
   ; cTesterUpDown ? x
   ; Δ2.0
   ; cTesterUpDone ! true
   )
]|
```

The *main_control* process The *main_control* process keeps track of the slot states and operates the other controllers. We use four integer variables (*p0, p1, p2, p3*) to describe the state of every slot. The variable values range from 0 to 4 (0 means that there is no product in the slot, 1 - there is a product in the slot and it is not drilled, 2 - a product has been drilled, 3 - a product has been tested and has a bad test result, and 4 - a product has been tested and has a good test result). First, the *main_control* process checks the states of the slots and starts corresponding processes (adding, drilling, testing and removing). As we assumed, the order of starting and finishing of these operations can vary and is not known a priori. In order to implement it, we use nested parallelism. The operations (*cEnvCanAdd*, *cStartDrill*, *cStartTest*, *cEnvCanRemove*) are started according to the following rules:

- The environment is allowed to add a product if there is no product in the slot.
- Drilling can be performed if there is a product in the slot and it has not been drilled yet or it has a bad test result.
- Testing is allowed if there is a product in the slot and it has been drilled.
- The environment is allowed to remove a product if there is a product in the slot and it has a good test result.

If these operations have been started the *main_control* process waits till they are completed (*cEnvAddFinished, cEnvRemFinished, cTested, cDrillEnded*). After that, it gives the command to the turntable controller (the process TTC) to read the states of the sensors at the adding and removing positions (*cGetS1, cGetS3*) and gets their current states (*cSetS1, cSetS3*). If their states have been changed (that means that the products have been added or removed), the *main_control* updates the information about current slot states. Then, it sends the command to the turntable controller to rotate the turntable (*cTurn*) and waits till the turn is completed (*cTurned*). Then, the loop is repeated. In the real system the states of the sensors at adding and removing positions are automatically updated during the turn. To achieve this in the model we send new states of the turntable sensors over the channel *cTurn.* In our model *main_control* sends the value of the sensors after the turn over the channel *cTurn* (the information is coded as an integer in following way: $p = 0$ means that there is no product in the adding and removing positions, $p = 1$ means that there is no product in the adding position and there is a product in the removing slot, $p = 2$ means that there is a product in the adding position and there is not product in the removing position, $p = 3$ means that there are products in both slots). Another approach to update the sensor states is to duplicate the information about all slots in the *turn_table* process [6]. This approach allows one to separate the physical and control systems easier and simpler but leads to a larger state space.

```
proc $Main\_control($ $cDrillEnded, \; cTested, \; cTurned, \; cSetS1$
                    $, \; cSetS3 : ?\,\mathsf{bool}$
                    $, \; cEnvCanAdd, \; cEnvAddFinished$
                    $, \; cStartDrill, \; cStartTest$
                    $, \; cEnvCanRemove, \; cEnvRemFinished$
                    $, \; cGetS1, \; cGetS3 : !\,\mathsf{bool} \; , \; cTurn : !\,nat$
                    $)=$
$[\![ \; x, \; y : \mathsf{bool} \; , \; p : nat = 0, \; pp : nat = 0,$
  $p0 : nat = 0, \; p1 : nat = 0, \; p2 : nat = 0, \; p3 : nat = 0$
$| \; *( \; ($
      $( \; p0 = 0 \rightarrow cEnvCanAdd \, ! \, \mathrm{true}; \; cEnvAddFinished \, ? \, x$
      $[\!] \; p0 \neq 0 \rightarrow \mathrm{skip}$
      $)$
    $\| \; ( \quad p1 = 1 \vee p1 = 3 \quad \rightarrow cStartDrill \, ! \, \mathrm{true}; \; cDrillEnded \, ? \, x; \; p1 := 2$
      $[\!] \; \neg(p1 = 1 \vee p1 = 3) \rightarrow \mathrm{skip}$
      $)$
    $\| \; ( \; p2 = 2 \rightarrow cStartTest \, ! \, \mathrm{true}; \; cTested \, ? \, y; \; (y \rightarrow p2 := 4 \; [\!] \; \neg y \rightarrow p2 := 3)$
      $[\!] \; p2 \neq 2 \rightarrow \mathrm{skip}$
      $)$
    $\| \; ( \; p3 = 4 \rightarrow cEnvCanRemove \, ! \, \mathrm{true}; \; cEnvRemFinished \, ? \, x$
      $[\!] \; p3 \neq 4 \rightarrow \mathrm{skip}$
      $)$
    $)$
    $; \; ( \; p0 = 0 \rightarrow cGetS1 \, ! \, \mathrm{true}; \; cSetS1 \, ? \, x$
                $; \; (x \rightarrow p0 := 1 \; [\!] \; \neg x \rightarrow \mathrm{skip})$
      $[\!] \; p0 \neq 0 \rightarrow \mathrm{skip}$
      $)$
    $; \; ( \; p3 = 4 \rightarrow cGetS3 \, ! \, \mathrm{true}; \; cSetS3 \, ? \, x; \; (\neg x \rightarrow p3 := 0 \; [\!] \; x \rightarrow \mathrm{skip})$
      $[\!] \; p3 \neq 4 \rightarrow \mathrm{skip}$
      $)$
    $; \; pp := p3; \; p3 := p2; \; p2 := p1; \; p1 := p0; \; p0 := pp$
    $; \; ( \; p0 = 0 \rightarrow (p3 = 0 \rightarrow p := 0 \; [\!] \; p3 \neq 0 \rightarrow p := 1)$
      $[\!] \; p0 \neq 0 \rightarrow (p3 = 0 \rightarrow p := 2 \; [\!] \; p3 \neq 0 \rightarrow p := 3)$
      $)$
    $; \; cTurn \, ! \, p; \; cTurned \, ? \, x$
  $)$
$]\!|$
```

The *drill_control* process The process *drill_control* gets the command to start drilling from the *main_control* over the channel *cStartDrill*. Then, it sends a signal to lock the clamp (*cClampOnOff*) and waits for the reply from the clamp sensor (*cLocked*). When the clamp is locked the *drill_control* uses the other switching command (*cDrillOnOff*) to start drilling and waits for the confirmation (*cDrillOnDone*). Then, it gives a signal to start drilling (*cDrillUpDown*), waits for confirmation from the sensor (*cDrillDownDone*), sends a signal to return the drill in its initial (up) position (*cDrillUpDown*), and waits for confirmation from the sensor (*cDrillUpDone*). Then, the *drill_control* switches the drill off (*cDrillOnOff*), and waits for confirmation (*cDrillOffDone*). After that, the *drill_control* switches the clamp on again (*cClampOnOff*), waits for the signal from the clamp sensor (*cUnlocked*) and reports to the *main_control* that drilling is completed (*cDrillEnded*).

```
proc Drill_control( cStartDrill, cLocked, cUnlocked
                  , cDrillUpDone, cDrillDownDone
                  , cDrillOnDone, cDrillOffDone
                  , cDrillEnded : ? bool
                  , cClampOnOff, cDrillUpDown
                  , cDrillOnOff : ! bool
                  )=
|[ x : bool
| *( cStartDrill ? x
   ; cClampOnOff ! true; cLocked ? x
   ; cDrillOnOff ! true; cDrillOnDone ? x
   ; cDrillUpDown ! true; cDrillDownDone ? x
   ; cDrillUpDown ! true; cDrillUpDone ? x
   ; cDrillOnOff ! true; cDrillOffDone ? x
   ; cClampOnOff ! true; cUnlocked ? x
   ; cDrillEnded ! true
   )
]|
```

The *tester_control* process *Tester_control* gets a command to perform testing from *main_control* (*cStartTest*) and switches *tester* on (*cTesterUpDown*). To perform the testing operation, *tester* needs 2 time units. If the tester has reached its down position within 2 time units, the test result of the product is good (*cTesterDownDone*) and if the sensor does not react in 2 time units, the test result of the product is bad. However, in our model *tester_control* waits for the signal from the *tester* for 4 time units instead of 2. The reason for this is that if *tester* and *tester_control* delay for the same amount of time, there is a possibility that *tester_control* would make its choice before *tester*. So, in order to ensure that *tester* always makes its choice before *tester_control* the latter delays longer. In that case, *tester* makes a choice in 2 time units and after that *tester_control* has no choice anymore. Then, *tester_control* stores the test result (*bTstRes*), switches *tester* off (*cTesterUpDown*), and sends the test result to *main_control* over the channel *cTested*.

```
proc Tester_control( cStartTest, cTesterDownDone
                   , cTesterUpDone : ? bool
                   , cTesterUpDown, cTested : ! bool
                   )=
|[ x, bTstRes : bool
| *( cStartTest ? x
   ; cTesterUpDown ! true
   ; (cTesterDownDone ? bTstRes
     [] Δ4.0; bTstRes := false
     )
   ; cTesterUpDown ! true
   ; cTesterUpDone ? x
   ; cTested ! bTstRes
   )
]|
```

The *TTC* process The process *TTC* (the turntable controller) gets commands from *main_control* to perform the turn or update sensor information. When the turn is completed *TTC* sends a signal to *main control* over the channel *cTurned*.

```
proc TTC( cTurn : ? nat, cS1, cS2, cS3
        , cGetS1, cGetS3 : ? bool
        , cSetS1, cSetS3, cUpdateS1, cUpdateS3
        , cRotate, cTurned : ! bool
        )=
|[ x : bool , bS1 : bool = false
, bS3 : bool = false, ss : nat = 0
| *( cTurn ? ss; cRotate ! true
     ; cUpdateS1 ! ss = 2 ∨ ss = 3
     ; cUpdateS3 ! ss = 1 ∨ ss = 3
     ; cS2 ? x; cTurned ! true
   [] cGetS1 ? x; cS1 ? bS1; cSetS1 ! bS1
   [] cGetS3 ? x; cS3 ? bS3; cSetS3 ! bS3
   )
]|
```

The environment processes There are two environment processes in the model: adding and removing. They get appropriate signals from *main_control* to add or remove a product (*cEnvCanAdd, cEnvCanRemove*). After performing (or skipping) the operations the environment processes notify *main_control* that they have finished (*cEnvAddFinished, cEnvRemFinished*). If a product is added or removed the environment processes send corresponding messages to the *turn_table* process through the channels (*cEnvAdded, cEnvRemoved*).

```
proc Env_add( cEnvCanAdd : ?bool , cEnvAdded
            , cEnvAddFinished : !bool
            )=
|[ x : bool
| *( cEnvCanAdd ? x
   ; (skip [] cEnvAdded ! true)
   ; cEnvAddFinished ! true
   )
]|

proc Env_remove( cEnvCanRemove : ?bool , cEnvRemoved
               , cEnvRemFinished : !bool
               )=
|[ x : bool
| *( cEnvCanRemove ? x
   ; (cEnvRemoved ! true [] skip)
   ; cEnvRemFinished ! true
   )
]|
```

The state space As already mentioned, there is no tool available for χ yet. Therefore we have generated the state space in the χ_σ toolset with the following results: the number of states is 32570 (6839 states after minimization under strong bisimulation).

4 Promela/Spin

4.1 Introduction to Promela/Spin

The full presentation of Promela, a very complex language, is beyond the scope of this paper. We give here only a brief overview mentioning only those parts of the language that we are interested in. For more information, see [3,19,20] or consult the Spin's web page `http://spinroot.com`.

Promela's syntax is derived from C [21], with communication primitives from CSP [22] and control flow statements based on the guarded command language [23]. It has many language constructs similar to χ_σ constructs.

A common specification consists of global channel declarations, variable declarations and process declarations with possibly one special `init` process. Process declarations specify behavior, channel and variable declarations define the environment in which the processes run. Promela has a rather limited set of data types, only bool, byte, short, int (all with the unsigned possibility) and channels. It also provides a way to build records and arrays and to define macros (processed by the C language preprocessor). Message channels are declared, for instance, as chan m = [2] of {int} meaning that the channel is buffered and it can store (at most) two values of type integer (has only one field of type int). Channels can also be of length 0, i.e. unbuffered, to model synchronous communication. They can also have more than one field, not necessarily of the same type.

Every variable must be declared before use. The exception is the special dummy variable '_' which is a predefined write-only variable, that can be used to store scratch values. The type of this global variable is int. It is an error to use or reference its value.

Process declarations are of this form:

```
proctype name(parameters) {
   local variables and channels;

   body
}
```

Local variables and channels specify the local state of the process and they are not visible to other processes. The same rules as for global variables apply here. The body is a list of statements, itself a statement. Any expression can be used as a statement, enabled precisely if it evaluates to a non-zero value. Assignments are also statements and have the usual semantics. The skip statement executes the action (1) and has no effect on variables. The send statement (m!e_1,...,e_n) sends a tuple of values of the expressions e_i to the channel m. The receive statement (m?E_1,...,E_n) retrieves a message from the non-empty channel m, for every E_i that is a variable assigns a value of e_i to it and for every other E_j makes sure that its value matches the value of the e_j. If the channel is buffered, a send is enabled if the buffer is not full; a receive is enabled if the buffer is non-empty. On an unbuffered channel, a send (receive) is enabled only if there is a corresponding receive (send) that can be executed simultaneously. There are also many variants of these statements (message can be left in or removed from a channel after receiving, send/receive can only be offered etc.)

There are several ways to combine statements. The alternative composition is defined by the selection statement:

```
if
   :: statements
   ...
   :: statements
```

```
fi.
```

It selects one among its options and executes it. An option can be selected if its first statement is executable. A selection blocks until there is at least one selectable option. If more than one option is selectable, one will be chosen non-deterministically. The repetition is achieved by the statement

```
do
   :: statements
   ...
   :: statements
od.
```

It is similar to the selection statement except that the choices are executed repeatedly, until control is explicitly transferred to outside the statement by `break` or the `goto` statement. The `break` terminates the innermost repetition statement in which it is executed and cannot be used outside a repetition.

Another way to combine statements is to use sequential composition denoted as `p;q` or `b -> p`. The latter is usually used to emphasize that a process `p` is guarded by the conditional expression/statement `b`.

The original version of PROMELA/SPIN is untimed but there is a discrete time extension, called DTPROMELA/DTSPIN [24]. The idea is to divide time into slices and then frame actions into these slices. The time between actions is measured in ticks of a global digital clock. By having a variable `t` declared as `timer`, setting its value to some expression that evaluates to a natural number (by doing `set(t,e)`) and waiting for `t` to expire (by stating `expire(t)`) a process can be enforced to postpone its execution for `n` time slices (where `n` is the value of `e`). When DTSPIN executes the `timeout` action, all timers synchronize and time progresses to a next slice. This action is executed only if no other actions can be executed, meaning that maximal progress is implicit. Deadlock is recognized when `timeout` is about to happen and all timers are off (not set or already expired).

PROMELA provides two constructs, `atomic{stmt_1;...;stmt_n}` and `d_step{stmt_1;...;stmt_n}` that can be used to model indivisible events and to reduce a state space. Their purpose is to forbid the statements from inside to interleave with other statements in the specifications. The difference is that additionally `d_step` executes all statements as one (one state in the state space). These constructs are very useful but have a limitation: statements other than the first may not block and the `d_step` cannot contain send/receive statements on unbuffered channels.

Once declared every process can be started by the PROMELA process creation mechanism, the `run` statement. The special `init` process, if present, is automatically instantiated once, and is often used to prepare the true initial state of a system by initializing variables and running the appropriate process-instances. Processes can be started with different parameters. Once started they execute in parallel with the interleaving semantics. This is the only way to achieve parallelism because there is no explicit parallel operator. Processes communicate with each other through global variables and channels.

4.2 The turntable model in Promela

Translation of χ constructs like assignments, skip statement, sequential and alternative composition and repetition is straight-forward since they have obvious equivalents in Promela.

The data types used in the turntable model are also present in Promela.

Both languages have a notion of channels. Communication in χ is synchronous and consequently all channels in the Promela translation are of length zero. For example, channel *cRotate* is declared as `chan cRotate = [0] of {bool}`.

In general **proc** definitions of χ are translated to `proctype` definitions of Promela and `init` process is used to run them all.

For example, the *drill_control* process is translated as

```
proctype drill_control() {

  do
  :: cStartDrill?_,1;
     cClampOnOff!1; cLocked?_;
     cDrillOnOff!1; cDrillOnDone?_;
     cDrillUpDown!1; cDrillDownDone?_;
     cDrillUpDown!1; cDrillUpDone?_;
     cDrillOnOff!1;  cDrillOffDone?_;
     cClampOnOff!1;  cUnlocked?_;
     cDrillEnded!1
  od;
}
```

Since in the *drill_control* process we use channels only for synchronization, after receiving we don't need the value of x so we replace it by the dummy variable _. The additional parameter ’`,1`’ in `cStartDrill?_,1` will be explained later.

We now present features of which translation requires a more careful consideration.

Guards Statements of type $b \rightarrow p$, in general cannot be just translated as `b -> p`. This is due to the fact that, since in PROMELA operator `->` is equivalent to the sequential operator and the boolean expression `b` is also a statement, if the value of `b` is `true`, SPIN will execute the action `(1)` (e.g. it will pass the guard) even though process `p` cannot execute anything. This is different from χ which looks for both b to be true and for p to be executable before taking the step.

However, if p in $b \rightarrow p$ is an atomic process there is a way to translate. The guarded assignment such as $b \rightarrow x := e$ is translated as `d_step{b; x = e}`. With the `d_step` operator we force the statement to be executed as one action like in χ. If the value of `b` is `false` the statement is blocked and if it is `true`, since an assignment is always executable, the statement will execute only one action. Translation is similar for a guarded skip.

In order to translate guarded send/receive actions we must apply a different trick because those actions can block and therefore cannot be put inside the `d_step` statement. For a channel that has send/receive actions involved in guarded statements we first change the declaration by adding another field argument to it, one of an integer type. We need the extra argument to synchronize on guards and we translate $b \rightarrow m\,!\,e$ to `m!e,b` and $B \rightarrow m\,?\,x$ to `m?x,eval(2-B)`. We use `2-B` instead of just `B` because the communication between a guarded send and a guarded receive should not take place if both guards evaluate to false (`2-B = b` is equivalent to `B=1` and `b=1`). The `eval` function is used to force the evaluation of the expression `2-B`. SPIN does not do this automatically in receive statements because the expression can be a variable in which case it should not serve as a match but instead it would be assigned the incoming value from the message field. If a communication action, for example $m\,?\,x$, is not used in the guarded context but its counterpart send is, then it should be translated to `m?x,1`. This goes similarly for $m\,!\,e$ when a corresponding receive is guarded. For example, in the *main_control* process a send action on the channel *cStartDrill* is guarded, $p_1 = 1 \vee p_1 = 3 \rightarrow \mathit{cStartDrill}\,!\,\mathit{true}$, and this statement is translated as `cStartDrill!1,(p1 == 1 || p1 == 3)`. The corresponding receive action, $\mathit{cStartDrill}\,?\,x$ in the *drill_control* process, is not guarded and therefore translated as `cStartDrill?_,1`.

In case of the *main_control* process not only atomic processes are guarded, but also, for example, we see

$$\begin{array}{ll} & p_0 = 0 \rightarrow (p_3 = 0 \rightarrow p := 0 \parallel p_3 \neq 0 \rightarrow p := 1) \\ \parallel & p_0 \neq 0 \rightarrow (p_3 = 0 \rightarrow p := 2 \parallel p_3 \neq 0 \rightarrow p := 3) \end{array}$$

To translate this fragment we use the fact that $b_1 \rightarrow (b_2 \rightarrow p)$ is equivalent to $(b_1 \wedge b_2) \rightarrow p$ and that $b \rightarrow (p \parallel q)$ is equivalent to $(b \rightarrow p) \parallel (b \rightarrow q)$. This assures that we can distribute guards over the operators and have the equivalent process with guarded atomic processes only. The PROMELA translation is therefore:

```
if
     :: d_step{(p0 == 0 && p3 == 0) -> p = 0}
     :: d_step{(p0 == 0 && p3 != 0) -> p = 1}
     :: d_step{(p0 != 0 && p3 == 0) -> p = 2}
     :: d_step{(p0 != 0 && p3 != 0) -> p = 3}
fi;
```

Time Note that in the turntable model all the delays are natural numbers so we don't think that much is lost when switching from continuous to discrete time. The $\Delta\, n$ statement is translated to the DTPROMELA statement `expire(t)`, where `t` is `timer`, previously set to the value of n. In cases where $\Delta\, n$ is not involved in a choice, `set(t,n)` can be present immediately before the `expire(t)`. This is, indeed, the case in the translations of the *clamp*, *turn_table*, *drill*, *tester*, *env_add* and the *end_remove* process. However, in the tester_control process there is an alternative composition of delaying and receiving:

$cTesterUpDown\,!\,\text{true};$
$(\;cTesterDownDone\,?;$
$\quad bTstRes := \text{true}$
$[\!]\;\Delta 2;$
$\quad bTstRes := \text{false}$
$);$
$cTesterUpDown\,!\,\text{true}$

In order to prevent time from making a choice, the `set(t,4)` must be moved to some place 'safe', i.e. outside of the alternative composition. That is because it is always executable and therefore always available as a choice, while `expire(t)` is a boolean expression/statement that is blocked until `4` time slices later. The discussed fragment of the *tester_control* process is translated as:

```
proctype tester_control(){
  bool bTstRes;
  timer t;

  do
  :: cStartTest?_,1; set(t,4);
     cTesterUpDown!1;
     if
     :: cTesterDownDone?_; bTstRes = 1
     :: expire(t); bTstRes = 0
```

```
      fi;

  . . .

  od
}
```

Parallel operator In Promela there is no explicit parallel operator. Since processes *drill* and *main_control* contain it, we encounter a problem when trying to translate them.

In the *drill* process no variables are shared (except the dummy x that is removed in the Promela translation anyway) and the parallel operator is not used in the context of other operators. This means that *drill* can be split into two smaller processes that can be translated separately:

```
proctype drill1() {
  timer t;

  do
  :: cDrillUpDown?_;
     set(t,3); expire(t);
     cDrillDownDone!1;
     cDrillUpDown?_;
     set(t,2); expire(t);
     cDrillUpDone!1
  od
}
```

```
proctype drill2() {

  do
  :: cDrillOnOff?_;
     cDrillOnDone!1;
     cDrillOnOff?_;
     cDrillOffDone!1
  od
}
```

The `drill1` and `drill2` are executing in parallel when started in the `init` process.

On the other hand, in the *main_control* process, the parallel operator is used within a repetition and a sequential composition context. To solve this problem we use Promela's process creation mechanism. The parts of the *main_control* process that run in parallel are translated to separate process definitions, namely `MC1()`, `MC2()`, `MC3()` and `MC4()`. These processes should not be started in the `init` process since they are not available from the beginning. The part that comes after the parallel composition (together with the loop) is also translated to the new process but with the additional statement at the beginning of the loop whose role is to start the new processes. This process is called `main_control` and it must be started in the `init` process.

There is one more problem to solve. After the `main_control` process starts its subprocesses it should be waiting for them to finish, not run in parallel with them as would be the case now. Therefore, some synchronization is needed. We use a global variable `WAIT` of type integer, initially 0, which is incremented at the end of each subprocess, and for which the `main_control` waits to be equal to 4, the number of subprocesses it started. Then, it sets the variable back to 0 (for later use) and continues. Therefore, *main_control* is translated as:

```
proctype main_control(){
  bool x;
  int p = 0, pp = 0;

  do
  :: atomic{ run MC1(); run MC2();
             run MC3(); run MC4() };

     d_step{ (WAIT == 4) -> WAIT = 0 };

  . . .

  od
}
```

Note that since variables p_0, p_1, p_2 and p_3 are shared between parts that are now separate processes in PROMELA, they must be declared in the global scope.

Remark: Since parts of the *main_control* process that run in parallel don't communicate with each other the parallel operator here is just an interleaving operator. In some cases the interleaving of actions in PROMELA could also be achieved with one loop and few additional guards (boolean variables). The idea is to associate one guard to each action. If there is a choice between two actions they share the same guard. Only actions available from the start have there guards initially set to true. When an action is executed, its guard is put to false and the guard of the action that comes next is assigned true. This is done in a loop that is exited when all the guards are false. To illustrate the technique we give an example. The interleaving between $a; b$ and $c; (d \; [] \; e)$ can be expressed as:

```
bool b1,b2,b3,b4;

d_step{ b1=1; b2=0; b3=1; b4=0; }

do
   :: d_step{ b1->a; b1=0; b2=1 }
   :: d_step{ b2->b; b2=0 }
   :: d_step{ b3->c; b3=0; b4=1 }
   :: d_step{ b4->d; b4=0 }
   :: d_step{ b4->e; b4=0 }
   :: !(b1 || b2 || b3 || b4) -> break
od;
```

The d_step is used to prevent state space from growing when introducing extra actions.

However, this approach results in a PROMELA model that is not very similar to the original χ model so we use it only to compare the state spaces generated by the χ_σ toolset and SPIN.

4.3 Verification of the model in SPIN

In this section we first compare state spaces generated by SPIN and χ_σ and later show how we verified the properties of the turntable.

By performing an exhaustive search, SPIN's verifier, almost instantly, reported 100995 states, 188724 transitions and 5.8975MB of memory used (3.342MB for states). To compare this result to the size of the state space generated by χ_σ (32570 states) we switch off all the optimizations of SPIN: like partial order reduction, statement merging and state vector compression. Now the number of states increases to 157576, the number of transitions to 455580. This shows the importance of the optimization features.

The huge difference in the number of states generated by χ_σ and SPIN is mostly the result of the set actions and the statements used for process creation and synchronizing in the *main_control* process. To show this we first force set actions to be executed atomically with the action before. Since this action is always send or receive we can't use the d_step, only the atomic statement. For example, in the turn_table process

```
cRotate?bS2; set(t,4);expire(t)
```

is changed to

```
atomic{
   cRotate?bS2;
   set(t,4)
}; expire(t).
```

Similarly in other processes. The number of states drops to 119616 and the number of transitions to 212876. Note that the fact that delays are always one (special) action in χ but can be more (timeout) actions in PROMELA and the fact that we used atomic instead of d_step, also introduce 'extra' states but this is unavoidable.

Second, instead of using the process creation mechanism we use the other trick (see the remark on page 25) to achieve nested parallelism. This results in 48252 states with 114048 transitions (32768/59154 fully optimized), much closer to the χ_σ's result. In this case, 9.236MB (7.170MB for states) is needed; 3.132MB (1.026MB for states) when fully optimized.

There are several ways to perform verification of properties in Spin but we use only LTL formulae verification and trace-assertions. The LTL mechanism checks properties expressed as linear temporal logic formulas over the values of variables (state based). The trace-assertion mechanism assures that the behavior of the system matches the behavior expressed as a deterministic automaton (trace) with only send/receive actions on globally declared channels as labels. In a case where communication is synchronous to prevent Spin from checking the send offers together with regular sending we use only receive actions as labels.

Now we discuss how the eight properties from section 2.1 can be expressed in a way Spin understands them:

(1) **The system does not contain a deadlock.** Absence of deadlock is verified in Spin by performing an exhaustive search for invalid end states.

(2) **If drilling (testing, adding or removing) is started then it is also finished and the turntable doesn't rotate in the meantime.** To verify this property we introduce two new variables into the Promela model, `drilling` and `rotating`, both initially 0. The idea is to keep track of states in which the table is turning and the states in which the drilling is going on. We set the `drilling` to 1 when the master controller sends a message to the drill controller instructing it to start drilling (`d_step{cStartDrill!1; drilling = 1}`), and set it back to 0 when master controller is informed that the drilling is finished (`d_step{cDrillEnded!1; drilling = 0}`). We do a similar thing for `rotating`. The `d_step` is used to prevent the state space from growing after the additional statement is added.

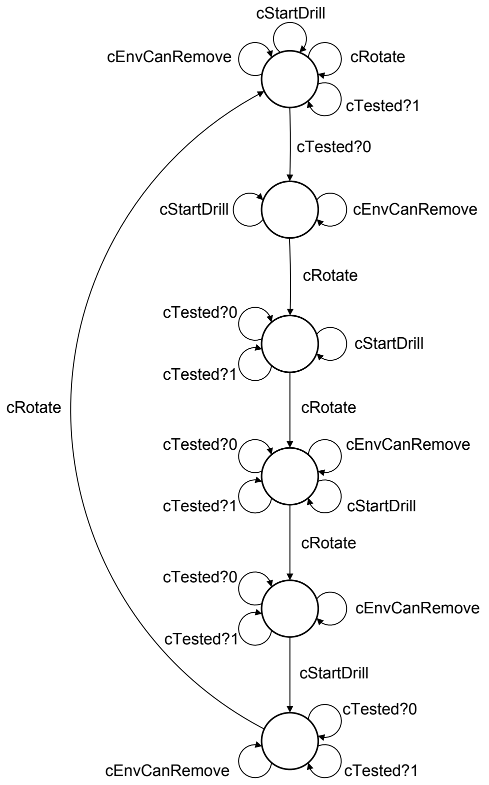
The property is now expressed as the LTL formula

```
[](drilling == 1 -> (rotating == 0 U drilling == 0))
```

Similarly, for testing, adding and removing.

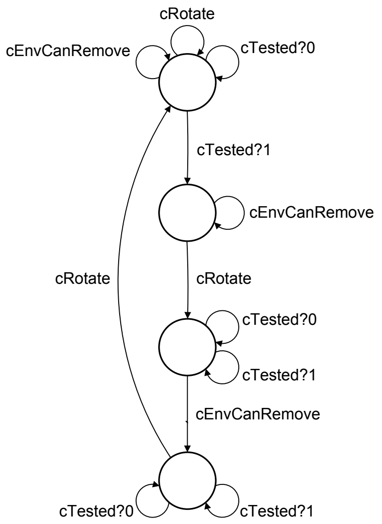
(3) **If the product has a bad test result the product remains on the table and is drilled again (when it comes to the drilling position).** Since the result of testing is communicated through the channel `cTested` and since it is easy to express the number of rotations we find the trace-assertion mechanism more suitable to verify this property then LTL. We must first rephrase this property so that it can be expressed with receive actions only: if a bad test result is received then in the next

rotation the master controller doesn't instruct the remover to remove and in the next two rotations (when we are back to the drilling position) the driller will drill the product again. Now we state this behavior as:



```
trace{ st1: if
    :: cEnvCanRemove?_,1 -> goto st1;
    :: cTested?1 -> goto st1;
    :: cTested?0 -> goto st2;
    :: cRotate?_ -> goto st1
    :: cStartDrill?_,1 -> goto st1
    fi;
st2: if
    :: cEnvCanRemove?_,1 -> goto st2;
    :: cRotate_1 -> goto st3
    :: cStartDrill?_,1 -> goto st2
    fi;
st3: if
    :: cTested?1 -> goto st3;
    :: cTested?0 -> goto st3;
    :: cRotate?_ -> goto st4
    :: cStartDrill?_,1 -> goto st3
    fi;
st4: if
    :: cTested?1 -> goto st4;
    :: cTested?0 -> goto st4;
    :: cEnvCanRemove?_,1 -> goto st4;
    :: cRotate?_ -> goto st5
    :: cStartDrill?_,1 -> goto st4
    fi;
st5: if
    :: cTested?1 -> goto st5;
    :: cTested?0 -> goto st5;
    :: cEnvCanRemove?_,1 -> goto st5;
    :: cStartDrill?_,1 -> goto st6
    fi;
st6: if
    :: cTested?1 -> goto st6;
    :: cTested?0 -> goto st6;
    :: cEnvCanRemove?_,1 -> goto st6;
    :: cRotate?_ -> goto st1
    fi;
}
```

(4) **If the product has a good test result the remover will be called to remove the product.** Similarly to the previous case we can rephrase the property and come out with the following trace:

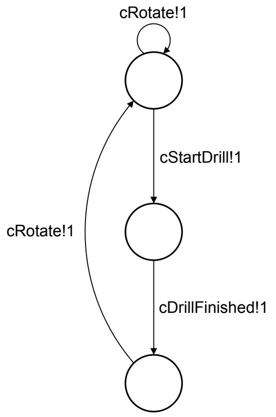


```
trace{ st1: if
   :: cEnvCanRemove?_,1 -> goto st1;
   :: cTested?0 -> goto st1;
   :: cTested?1 -> goto st2;
   :: cRotate_1 -> goto st1
   fi;
st2: if
   :: cEnvCanRemove?_,1 -> goto st2;
   :: cRotate?_ -> goto st3
   fi;
st3: if
   :: cTested?1 -> goto st3;
   :: cTested?0 -> goto st3;
   :: cEnvCanRemove?_,1 -> goto st4
   fi;
st4: if
   :: cTested?1 -> goto st4;
   :: cTested?0 -> goto st4;
   :: cRotate?_ -> goto st1
   fi;
}
```

(5) **No drilling (testing or removing) takes place if there is no product in the slot and no adding can be performed if there is a product in the slot).** The LTL formula that represents this property is:

```
[]!(p1 == 0 && drilling == 1)
```

but the following event-trace can be used as well:



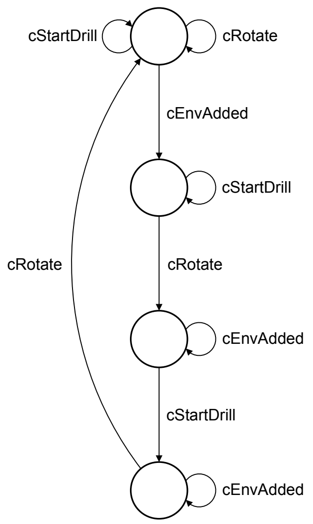
```
trace{ st1:  if
   :: cStartDrill?_,1 -> goto st2
   :: cRotate?_ -> goto st1
   fi;

st2:  if
   :: cDrillFinished?_ -> goto st2;
   fi;

st3:  if
   :: cRotate?_ -> goto st1
   fi
}
```

and similarly for other cases.

(6) **Every added product is drilled in the next rotation.** This property can be interpreted as: when you add and rotate afterwards then you must drill before you rotate again. The corresponding automata is:



```
trace{ s1: if
  :: cEnvAdded_1; goto s2;
  :: cRotate_1; goto s1;
  :: cStartDrill?_,1; goto s1;
  fi;

s2: if
  :: cRotate_1; goto s3;
  :: cStartDrill?_,1; goto s2;
  fi;

s3: if
  :: cEnvAdded_1; goto s3;
  :: cStartDrill?_,1; goto s4;
  fi;

s4: if
  :: cEnvAdded_1; goto s4;
  :: cRotate_1; goto s1;
  fi;
}
```

(7) **Every product eventually leaves the table.** To verify this property we first introduce a variable pstn that can have values 0, 1, 2 and 3. It represents the position of the turntable (or some mark on the rotating disk) with respect to the adding position. After every rotation the value of pstn is changed by the rule pstn = (pstn + 1) % 4. Second, variables removed and added are introduced. They keep track in which position the product was removed (added). They have two extra values: -1, meaning that the removing (adding) was skipped, and -2, a neutral value. The variable removed (added) is set to the neutral value after the remover (adder) has made a choice, to remove (add) or to skip. To verify the property we must now prove the following four LTL formulas:

```
[](added == 0 -> <> removed == 3)
[](added == 1 -> <> removed == 0)
[](added == 2 -> <> removed == 1)
[](added == 3 -> <> removed == 2)
```

We are sure that we are removing the same product we are adding since added cannot become 0 twice if removed doesn't become 3 in between (in the first formula, similarly for the other three). That is because if the product is not removed in $4k + 3$ rotations then in the next rotation we don't add because there is already a product in the slot. To add fairness we forbid the remover to always skip removing while in the position 3 (with a product to remove in place) and to always generate bad test results in the position 2 (with a product to test in place). The extended formula is:

```
(
 [](pstn==3 && p3==4 -> <> removed == 3) &&
 [](pstn==2 && p2==2 -> <> (bTstRes == 1 && pstn == 2)
) -> [](added == 0 -> <> removed == 3)
```

(8) **When a product is added it takes between 21 and 30 time units to get its test result.** For this property we must calculate the number of clock ticks (`timeout` actions) between adding a product and receiving its test result. To achieve this we must keep track of `timeout`'s somehow. Since they are not communication actions, trace assertion mechanism is ruled out. To use LTL mechanism we may try to add a special timer that counts ticks (it decreases accordingly) but because DTSpin does not allow values of timers in LTL formulas this would not be of much help. Another solution is to use `timeout` in formulas or to code the property directly as a never claim but at this moment it is hard to see how this can be done in an optimal and satisfactory way.

We here present a solution that is based on the fact that, if the product is added in some position that no product is added in the same position before the test result is known. Because we have to distinguish products again, we use the same idea (variable `pstn`) as before. Here we check that if adding happens in the position 0 then result is known in position 2 in 21-30 time units. Similarly for other positions.

We introduce a variable, called `added0`, which becomes 0 when adding does not happen (already a product in the slot) or is skipped (in the position 0), and 1 when product is added (in the position 0). When `added0` becomes 1, we also set a special timer variable, called `TT`, to 30. The idea is to check if `TT` has a value less or equal to $9(= 30 - 21)$ in a place where test result is obtained (in the position 2) and adding has previously happened (in the position 0). This is done by assertion mechanism of Spin, directly in code:

```
...
cTesterUpDone?_;
if
:: (added0 == 1 && pstn == 2) -> assert(TT.val < 10)
:: else
fi;
cTested!bTstRes;
...
```

5 μCRL/CADP

5.1 The language μCRL

Basically, μCRL is based on the process algebra ACP [11], extended with equational abstract data types [25]. In order to intertwine processes with data, actions and recursion variables can be parametrized with data types. Moreover, a conditional construct (if-then-else) can be used to have data elements influence the course of a process, and *alternative quantification* (also called *choice quantification*) is added to sum over possibly infinite data domains.

The language comes with a toolset [26] that can build a state space from a specification and store it in the `.aut` format, one of the input formats of the model checker CADP [2] (more on this model checker in paragraph 5.3). Next to that, in order to strive for precision in proofs, an important research area is to use theorem provers such as PVS [27] to help in finding and checking derivations in μCRL. A large number of distributed systems have been verified in μCRL, often with the help of a proof checker or theorem prover [28,29].

We will give a short overview of the language. For a complete reference, see [10].

Data types Initially there are no data types known in a μCRL specification. Therefore each specification should start by defining the necessary data types and the functions that work on them. In fact, it is mandatory to define the boolean type in each specification, since the conditional construct works with boolean expressions. In the case of the turntable model, the natural numbers were also defined. One can virtually define any data type.

In μCRL one can specify abstract data types [25] in an algebraic way, with an explicit recognition of so-called *constructor* function symbols, which intuitively cannot be eliminated from data terms [10]. In the case of natural numbers the zero (`0`) and the successor function `S` are constructors, while addition (`plus`) is not. For booleans we have the constructors true (`T`) and false (`F`). This explicit recognition of constructor symbols makes it possible to enumerate the elements of a data type.

To define a data type one uses the keyword `sort`. A sort represents a non-empty set of data elements. To declare the sort of booleans one can write:

```
sort  Bool
```

Now the elements of the data type can be declared. This is done with the keywords `func` and `map`. A constructor symbol, declared with `func`, has as target the data type in question. For the booleans we declare `T` and `F` both with `func`:

```
sort  Bool
func  T,F:→Bool
```

What we state here is that the elements of `Bool` are `T` and `F`.

Now that the structure of the data type is given, one can add additional functions. These can be defined with map. For the booleans we define the function and and the equality relation eq.

Using rewrite rules one can now define how the functions work. When defining, it's allowed to use variables, which have to be defined first using var. Having added the rewrite rules for and and eq, the declaration of Bool now looks like this:

```
sort  Bool
func  T,F:→Bool
map   eq:Bool#Bool→Bool
      and:Bool#Bool→Bool
var   x,y:Bool
rew   eq(x,x) = T
      eq(T,F) = F
      eq(F,T) = F
      and(T,x) = x
      and(x,T) = x
      and(F,x) = F
      and(x,F) = F
```

In a similar way one can define the sort of natural numbers with the equality relation eq and the addition function plus:

```
sort  Nat
func  0:→Nat
      S:Nat→Nat
map   plus:Nat#Nat→Nat
      eq:Nat#Nat→Bool
var   x,y:Nat
rew   plus(x,0) = x
      plus(x,S(y)) = S(plus(x,y))
      eq(x,x) = T
      eq(0,S(x)) = F
      eq(S(x),0) = F
      eq(S(x),S(y)) = eq(x,y)
```

Actions In μCRL one can declare actions in the `act` section of a specification. These actions may have zero, one or several data parameters. When parameters are used the data types of these parameters need to be given. In the next example an action `a` is defined without a parameter, an action `b` is defined with a parameter of type `Bool` and an action `c` is defined with two parameters of type `Nat` and `Bool` respectively:

```
act  a

     b:Bool

     c:Nat#Bool
```

One can allow processes `P` and `Q` to communicate in the parallel process `P`$\|$ `Q`. To do this it is possible to define which actions are able to synchronize with each other using the keyword `comm`. The following example states that the actions `d` and `e` can synchronize and form the action `f` together:

```
comm  d | e = f
```

Finally the process deadlock (δ), which cannot terminate successfully, and the internal action τ are predefined.

Operators There are eight operators in μCRL. We present each one of them with an informal semantics.

(1) The alternative composition operator (`+`). A process `p+q` proceeds (non-deterministically) as `p` or `q` (if they can proceed).
(2) The sum operator ($\sum_{\mathtt{d:D}}$`X(d)`), with `X(d)` a mapping from the data type `D` to processes, behaves as $\mathtt{X(d_1)} + \mathtt{X(d_2)} + \ldots$, i.e., as the possibly infinite choice between `X(d)` for any data term `d` taken from `D`. This operator is used to describe a process that is reading some input over a data type [30].
(3) The sequential composition operator (`.`). A process `p.q` proceeds as `p` followed by `q`.
(4) The process expression $\mathtt{p} \triangleleft \mathtt{b} \triangleright \mathtt{q}$ where `p` and `q` are processes, and `b` is a data term of data type `Bool`, behaves as `p` if `b` is equal to `T` (true) and behaves as `q` if `b` is equal to `F` (false). This operator is called the conditional operator, and operates as a then_if_else construct.
(5) The parallel operator ($\|$). A process `p` $\|$ `q` executes `p` and `q` concurrently in an interleaved fashion, i.e. the actions of `p` and `q` are executed in arbitrary order. For all actions `a` and `b` which can communicate with each other: If one process can execute `a` and the other one can execute `b` then `p` and `q` can communicate (`p` $\|$ `q` executes the communication action).
(6) The encapsulation operator ($\partial_{\mathtt{H}}$). A process $\partial_{\mathtt{H}}$`(p)` disables all actions of

p that occur in the set $\mathtt{H} \subseteq \mathtt{Act}$. Typically this operator is used to enforce that certain actions synchronize.

(7) The renaming operator ($\rho_{\mathtt{f}}$), with `f: Act`→`Act`, is suited for reusing a given specification with different action names. The subscript `f` signifies that the action `a` must be renamed to `f(a)`. The process $\rho_{\mathtt{f}}$`(p)` behaves as `p` with its action names renamed according to `f`.

(8) The abstraction operator ($\tau_{\mathtt{I}}$). A process $\tau_{\mathtt{I}}$`(p)` 'hides' (renames to τ) all actions of `p` that occur in the set $\mathtt{I} \subseteq \mathtt{Act}$.

Process definitions The heart of a μCRL specification is the `proc` section, where the behavior of the system is declared. This section consists of recursion equations of the following form, for $\mathtt{n} \geq 0$:

$$\mathtt{proc}\quad \mathtt{X}(\mathtt{x}_1\mathtt{:s}_1,\ \ldots,\ \mathtt{x}_\mathtt{n}\mathtt{:s}_\mathtt{n}) = \mathtt{t}$$

Here `X` is the process name, the $\mathtt{x}_\mathtt{i}$ are variables, not clashing with the name of a function symbol of arity zero nor with a parameterless process or action name, and the $\mathtt{s}_\mathtt{i}$ are sort names, expressing that the data parameters $\mathtt{x}_\mathtt{i}$ are of type $\mathtt{s}_\mathtt{i}$. Moreover, `t` is a process term possibly containing occurrences of expressions $\mathtt{Y}(\mathtt{d}_1,\ \ldots,\ \mathtt{d}_\mathtt{m})$, where `Y` is a process name and the $\mathtt{d}_\mathtt{i}$ are data terms that may contain occurrences of the variables $\mathtt{x}_1,\ \ldots,\ \mathtt{x}_\mathtt{n}$. In this rule, $\mathtt{X}(\mathtt{x}_1,\ \ldots,\ \mathtt{x}_\mathtt{n})$ is declared to have the same (potential) behavior as the process expression `t` [10].

The initial state of the specification is declared in a separate initial declaration `init` section, which is of the form

$$\mathtt{init}\quad \mathtt{X}(\mathtt{d}_1,\ \ldots,\ \mathtt{d}_\mathtt{n})$$

Here $(\mathtt{d}_1,\ \ldots,\ \mathtt{d}_\mathtt{n})$ represents the initial behavior of the system that is being described. In general, in μCRL specifications the `init` section is used to instantiate the data parameters of a process declaration, meaning that the $\mathtt{d}_\mathtt{i}$ are data terms that do not contain variables. The `init` section may be omitted, in which case the initial behavior of the system is left unspecified.

The time model Delaying for a certain amount of time is impossible in μCRL at first glance. This is because μCRL does not work with time. A later extension of μCRL to *timed* μCRL [31] introduced the notion of time. However, at present creating a timed μCRL specification is not very practical since the μCRL toolset can only parse timed μCRL code and cannot generate a state space from it.

There is another way however to simulate some notion of discrete time. In this paper we use a method based on the one from [32]. In short it works like this: first we define two actions: `tick` and `tick2`. The `tick` action represents the end of a time slice and the beginning of a new one. In order to share this notion of time all running processes need to synchronize their `tick` actions. If at least one of these processes is busy and therefore unable to perform a `tick` the `tick` action will not take place. This synchronization aspect is essential if one wants to use global timing. Note that, using this technique, we get discrete time in μCRL, since we represent a time period as a number of time units.

In most cases when using time in a model the modeler would like to give normal actions priority over `tick` actions. In order to realize this χ has implicit maximal progress, but in μCRL an operator for this does not exist. We can however get similar results by using the `tick2` action and post-processing the system after linearization (more on the latter in section 5.2).

The differences between `tick2` and `tick` are the following:

- The action `tick` is used for translating delays, while `tick2` is used to make an action delay-able (which means adding a `tick2` self-loop as an alternative to this action);
- A `tick` action can synchronize with any number of `tick` or `tick2` actions, but a `tick2` action cannot synchronize with only `tick2` actions (at least one `tick` action is needed for going from one time unit to the next).

Now, several delay-able processes can delay together if there is a `tick` action enabled in at least one process.

5.2 The turntable model in μCRL

In the next few paragraphs we will look at the μCRL model of the turntable which resulted from translating the original χ model. The μCRL language will be explained as far as needed. Translating the turntable model was done in an intuitive fashion in order to get some inspiration for developing a translation scheme for translating χ specifications to μCRL specifications. In this paper therefore the way in which we ended up with this μCRL model will not be discussed in detail. We restrict ourselves to highlighting the interesting parts. Translating χ constructs like an assignment, a *skip* statement, a sequential composition and a guard can be translated straightforwardly since there exist similar constructs in μCRL. Here we have to note that except for translating the atomic processes all other translations involve the usage of a program counter `n`. This counter is used to control the order of execution within a process. Initially, when a process starts its execution, the counter equals zero. By changing the value each time an action is executed and including the counter in the guards accompanying the actions one can specify the order in which these actions should be executed. Since we translated χ processes to *Linear Process Equations* (LPEs) we needed such a counter to translate (for instance) sequential compositions. Later in this section a definition of LPE can be found. How the program counter functions in practice will become apparent when looking at the translations in the next few paragraphs.

Following are remarks on the constructions that could not be translated in an obvious way:

- In the μCRL model we move from one time slice to the next by synchronizing `tick` actions of all the processes in the system. This means that we use discrete timing. Delays in χ should therefore always be 'discretizable'. A χ delay of n time units is then translated to n `tick` actions placed in sequence.
- The usage of time in combination with communication actions is tricky when translating. If you have a send action in one process and a corresponding receive action in another which (if any) of the actions can be delayed until the other action can be executed? In principle both send and receive actions are delay-able in χ. The maximal progress operator then ensures that possible communications get a higher execution priority than a delay. In other words: Both send and receive actions can delay if communication is not possible. Once communication is possible it will be executed immediately.

  In μCRL however, making all send and receive actions delay-able may result in communications being delay-able as well, since we cannot assign different priorities to actions (see the time model paragraph in section 5.1); there is no maximal progress operator. Communications that turn out to be delay-able in the μCRL specification are therefore bad translations. These can be fixed once the specification is linearized using the μCRL toolset though: The linearized version can be (automatically) manipulated in such a way that these communications become non-delay-able again. In short, to do this the guards of all `tick` actions have to be changed in such a manner, that `tick` actions only become enabled if no 'normal' (non-`tick`) action is.
- Another problem is translating an alternative composition: How to translate it depends on the number of alternatives that begin with a delay or are delay-able. This is due to the weak time determinism principle [18]: The passage of time cannot result in making a choice between the alternatives that can perform the time transition. In other words: Can both alternatives perform a delay, then they can delay together after which a choice can still be made. This principle led basically to two different possible translations of an alternative composition when translating the turntable model:

(1) None of the alternatives begin with a delay. We can translate this statement using the program counter in the right way; both alternatives are enabled at the start by extending the guards of both first actions of these alternatives with the same equation concerning the value of the program counter.

(2) The left alternative is delay-able and the right alternative begins with a

delay. This functions as a time-out: If you can start executing the left alternative within the time given by the delay of the right alternative, do that. Otherwise execute the right alternative. This can be translated by introducing a time counter initially having the value of the delay of the right alternative. Then we add an extra line to the process where the process can execute a `tick` and decrease the time counter by one as long as the time counter does not equal zero. Using some additional guards makes the translation complete. This situation can be found later on in this paragraph in the process `TESTER_CONTROL`. Note however that there will not be maximal progress in this situation, so processing the linearized version is necessary (see the paragraph on the time model in section 5.1).

- In LPEs we are not allowed to use parallelism, as can be seen by looking at its definition later on in this section. In the χ turntable specification however there are two instances where we find parallelism inside a process: In the *drill* process and in the *main_control* process.

  The nested parallelism in the *drill* process can be easily translated by really translating the two subprocesses of *drill* to separate processes and placing them in parallel in the `init` line.

  The second instance however poses a bigger problem since the subprocesses share variables. Since we cannot use shared variables in μCRL we have to find some other way to translate this construction. We do that by translating the subprocesses to individual processes and providing each process with local copies of the shared variables. Say two processes `A` and `B` share a variable named `x`. Both processes can read the value of `x` at all times, but if one of them changes the value the other one should be aware of this (and change the value of its own 'copy' of `x` likewise). To make this possible a new action `assignx` is introduced, which is called by a process if it has changed the value of `x`. As a parameter the new value should be given. This action communicates with another action `updatex`, which can be executed by the other process **at all times**. This last thing is very important, since an assignment should proceed as soon at it is invoked. Once the other process can communicate via `updatex` it receives the new value for `x` and assigns this to the local copy.

  This does not solve the problem yet though; in the *main_control* process we can identify four subprocesses running in parallel followed by a fifth subprocess placed sequentially behind these four. Since in the `init` line all processes need to be placed in parallel we have to force this order of execution upon the five processes which are translations of the five χ subprocesses. We can do that by introducing some extra actions which synchronize between all the processes (this is done in a way similar to how it's done for `tick`); all processes start their execution once they can synchronize on the `start` action. Then the four placed in parallel begin while the fifth process starts waiting for the other four to finish. Once the four processes in parallel are finished with their execution, they synchronize their `stop` actions with the $\texttt{start}_2$ action of the fifth process. Then this process starts the execution.

Once it's finished it synchronizes its $\mathtt{stop}_2$ action with those of the other four processes, after which all processes can start all over again. The actual processes (named `MAIN_CONTROL1` through `MAIN_CONTROL5`) will be treated later on.

The data types First of all a μCRL model starts by defining the data types used. As is the case with most models the data types `Bool` and `Nat` are used (representing booleans and natural numbers respectively). See section 5.1 for part of their definitions.

The μCRL turntable model uses a third data type though, which is used for readability and represents the state of a slot; a slot can be in one of the following states:

- There is no product in the slot (`NoProduct`);
- There is a product in the slot which is not drilled and not tested yet (`Product`);
- There is a product in the slot which is drilled but not tested yet (`Drilled`);
- There is a product in the slot which is drilled and had a bad test result (`TestedBad`);
- There is a product in the slot which is drilled and had a good test result (`TestedGood`).

This data type is called SlotState.

Linear process equations When translating we used *Linear Process Equations* (LPEs). The definition of an LPE is as stated in [33]. We chose to translate χ processes to LPEs instead of the more general μCRL processes since LPEs proved to be better suited for setting up a general translation scheme. An LPE is of the following form:

$$\begin{aligned}\mathtt{X(d{:}D)} = &\sum_{\mathtt{i}\in\mathtt{I}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i(f_i(d,e_i)).X(g_i(d,e_i))} \lhd \mathtt{h_i(d,e_i)} \rhd \delta \ \mathtt{+}\\ &\sum_{\mathtt{i}\in\mathtt{I'}}\sum_{\mathtt{e_i}\in\mathtt{D_i'}} \mathtt{a_i'(f_i'(d,e_i)} \lhd \mathtt{h_i'(d,e_i)} \rhd \delta\end{aligned}$$

where $\mathtt{I}$ and $\mathtt{I'}$ are finite index sets, $\mathtt{D}$, $\mathtt{D_i}$, $\mathtt{D_i'}$, $\mathtt{D_{a_i}}$ and $\mathtt{D_{a_i'}}$ are data types, $\mathtt{a_i}, \mathtt{a_i'} \in \mathbf{Act} \cup \{\tau\}$, $\mathtt{a_i} : \mathtt{D_{a_i}}$, $\mathtt{a_i'} : \mathtt{D_{a_i'}}$, $\mathtt{f_i} : \mathtt{D} \times \mathtt{D_i} \longrightarrow \mathtt{D_{a_i}}$, $\mathtt{f_i'} : \mathtt{D} \times \mathtt{D_i'} \longrightarrow \mathtt{D_{a_i'}}$, $\mathtt{g_i} : \mathtt{D} \times \mathtt{D_i} \longrightarrow \mathtt{D}$, $\mathtt{h_i} : \mathtt{D} \times \mathtt{D_i} \longrightarrow \mathtt{Bool}$ and $\mathtt{h_i'} : \mathtt{D} \times \mathtt{D_i'} \longrightarrow \mathtt{Bool}$.

Here the different states of the process are represented by the data parameter `d:D`. Type `D` may also be a Cartesian product of n data types. Besides that the data parameter $\mathtt{e_i}$ (either of type $\mathtt{D_i}$ or $\mathtt{D_i'}$) can influence the parameter of action $\texttt{a}_\texttt{i}$ (or $\texttt{a}_\texttt{i}$'), the condition $\texttt{h}_\texttt{i}$ (or $\texttt{h}_\texttt{i}$') and the resulting state $\texttt{g}_\texttt{i}$ (or $\texttt{g}_\texttt{i}$'), thereby giving LPEs a more general form. The data parameter $\texttt{e}_\texttt{i}$ is typically used to let a read action range over a data domain.

The LPE expresses that in state `d` it can do two things:

(1) One can perform actions $\texttt{a}_\texttt{i}$, carrying a data parameter $\texttt{f}_\texttt{i}\texttt{(d,e}_\texttt{i}\texttt{)}$, under the condition that $\texttt{h}_\texttt{i}\texttt{(d,e}_\texttt{i}\texttt{)}$ is true; in such a case the resulting state is $\texttt{g}_\texttt{i}\texttt{(d,e}_\texttt{i}\texttt{)}$ [10].
(2) One can perform actions $\texttt{a}_\texttt{i}$', carrying a data parameter $\texttt{f}_\texttt{i}\texttt{'(d,e}_\texttt{i}\texttt{)}$, under the condition that $\texttt{h}_\texttt{i}\texttt{'(d,e}_\texttt{i}\texttt{)}$ is true; after executing one of these actions the process terminates.

In general, when translating a χ process to an LPE the locally defined variables s_i in the scope operator of χ should be translated to parameters of the LPE (in other words, should become part of the data parameter `d:D`). The initial values of these variables can be set at the initialization line. Channels in χ that a process works with are mentioned as parameters of that process. These should not be included in the LPE. An LPE can work with communication actions which are defined globally. More on communication actions in the next paragraph.

The actions and communication rules The turntable model contains a lot of different channels. We will not provide a full list here; that can be obtained from looking at the paragraph about the original χ model. Here we only give a guideline how to translate channels to pairs of actions. In general the usage of a channel a in the χ model will be translated as follows:

(1) Sending a message over channel a will appear in the LPEs as the action `sa` with possibly a value as a parameter being the message sent; if no parameter is provided the send functions as a trigger for a certain event.
(2) Receiving a message over channel a will appear in the LPEs as the action `ra` or $\sum_{\texttt{y:Type}}\texttt{(ra(y).X(y))}$ depending again on the fact whether the receive action functions as a trigger for an event or really for receiving a value (of type `Type`) respectively.
(3) For these two actions a communication rule must be specified which looks like this: `sa | ra = ca`. This together with the encapsulation operator ($\partial_{\{\texttt{sa,ra}\}}$) forces the actions `sa` and `ra` to only execute if they can synchronize.

Next we will look at interesting parts of some of the processes to give an idea of how the translation is done.

The *turn_table* process The first process we look at is the *turn_table* process. Here is part of the μCRL specification:

```
proc TURN_TABLE(n : Nat, bS1 : Bool, bS2 : Bool, bS3 : Bool) =
 sS1(bS1).TURN_TABLE(0, bS1, bS2, bS3) ◁ n = 0 ▷ δ +
 ...
 rEnvAdded.TURN_TABLE(0, T, bS2, bS3) ◁ n = 0 ▷ δ +
 ...
 τ.TURN_TABLE(0, bS1, bS2, F) ◁ n = 1 ▷ δ +
 ∑        rUpdateS1(b).TURN_TABLE(0, b, bS2, bS3) ◁ n = 0 ▷ δ +
 b:Bool
 ...
 ∑        rRotate(b).TURN_TABLE(2, bS1, b, bS3) ◁ n = 0 ▷ δ +
 b:Bool
 tick.TURN_TABLE(3, bS1, bS2, bS3) ◁ n = 2 ▷ δ +
 ...
 τ.TURN_TABLE(0, bS1, F, bS3) ◁ n = 6 ▷ δ +
 tick2.TURN_TABLE(n, bS1, bS2, bS3) ◁ n = 0 ▷ δ
```

In every μCRL process of the turntable specification the program counter `n` is used to control the order of action execution. Initially this `n` equals `0`. Therefore the program can start execution only by performing one of the actions for which the accompanying guard includes the conjunct `n=0`. The turntable for instance initially has several alternative options:

- It can send the value of any of the three sensors $S1$, $S2$ and $S3$. The state will not change after this.
- It can receive the message that the environment has received a new product or that a product has been removed. This will change the state in the obvious way.
- It can get new values for the sensors $S1$ and $S3$. These will be set.
- It can receive the request to rotate. Rotating takes four time units.
- The lines beginning with τ are in fact translations of χ assign actions.
- The line beginning with `tick2` is there to make some actions delay-able. The guard tells us which actions they are (see the paragraph on the time model in section 5.1).

The *tester_control* process The *tester_control* process controls the testing procedure. The following is part of the translation of the original process:

$$
\begin{array}{l}
\texttt{proc TESTER\_CONTROL}(\texttt{t}:\texttt{Nat},\texttt{n}:\texttt{Nat},\texttt{bTstRes}:\texttt{Bool}) = \\
\quad \ldots \\
\quad \texttt{rTesterDownDone.TESTER\_CONTROL}(0,3,\texttt{T}) \lhd \texttt{n} = 2 \rhd \delta \; + \\
\quad \texttt{tick.TESTER\_CONTROL}(\texttt{t}-1,\texttt{n},\texttt{bTstRes}) \lhd \texttt{n} = 2 \land \texttt{t} \neq 0 \rhd \delta \; + \\
\quad \tau.\texttt{TESTER\_CONTROL}(0,3,\texttt{F}) \lhd \texttt{n} = 2 \land \texttt{t} = 0 \rhd \delta \; + \\
\quad \ldots
\end{array}
$$

The sequence of actions starts when the command is received to start testing. Then TESTER is given the command to move the testing device. Should the message return within 4 time units after the testing device has been lowered, then bTstRes is set to true, otherwise to false. Note here the usage of counter t, which indicates the number of time units left before the time out ends. Next the testing device is raised and the result (the value of bTstRes) is sent to the MAIN_CONTROL process.

The *main_control* subprocesses Finally the most important and complicated process is the *main_control* process. As can be seen in the χ specification this process really consists of five subprocesses, with at first four of them running in parallel after which a fifth starts execution. Once the fifth subprocess has finished the four others restart etc. Since the subprocesses share some variables this requires special attention when translating. Basically each of the subprocesses is translated to an LPE using several special actions to synchronize the values of the local copies of the originally shared variables.

The first subprocess is called MAIN_CONTROL1. Part of it is given here:

$$
\begin{array}{l}
\texttt{proc MAIN\_CONTROL1}(\texttt{n}:\texttt{Nat},\texttt{p}_0,\ldots,\texttt{p}_3:\texttt{SlotState}) = \\
\quad \sum_{\texttt{s:SlotState}} \texttt{updatep}_0(\texttt{s}).\texttt{MAIN\_CONTROL1}(\texttt{n},\texttt{s},\texttt{p}_1,\texttt{p}_2,\texttt{p}_3) \; + \\
\quad \ldots \\
\quad \texttt{start.MAIN\_CONTROL1}(1,\texttt{p}_0,\texttt{p}_1,\texttt{p}_2,\texttt{p}_3) \lhd \texttt{n} = 0 \rhd \delta \; + \\
\quad \texttt{sEnvCanAdd.MAIN\_CONTROL1}(2,\texttt{p}_0,\texttt{p}_1,\texttt{p}_2,\texttt{p}_3) \lhd \texttt{n} = 1 \land \texttt{p}_0 = \texttt{NoProduct} \rhd \delta \; + \\
\quad \texttt{stop.MAIN\_CONTROL1}(4,\texttt{p}_0,\texttt{p}_1,\texttt{p}_2,\texttt{p}_3) \lhd \texttt{n} = 1 \land \texttt{p}_0 \neq \texttt{NoProduct} \rhd \delta \; + \\
\quad \texttt{rEnvAddFinished.MAIN\_CONTROL1}(3,\texttt{p}_0,\texttt{p}_1,\texttt{p}_2,\texttt{p}_3) \lhd \texttt{n} = 2 \rhd \delta \; + \\
\quad \texttt{stop.MAIN\_CONTROL1}(4,\texttt{p}_0,\texttt{p}_1,\texttt{p}_2,\texttt{p}_3) \lhd \texttt{n} = 3 \rhd \delta \; + \\
\quad \texttt{stop}_2\texttt{.MAIN\_CONTROL1}(0,\texttt{p}_3,\texttt{p}_0,\texttt{p}_1,\texttt{p}_2) \lhd \texttt{n} = 4 \rhd \delta \; + \\
\quad \texttt{tick2.MAIN\_CONTROL1}(\texttt{n},\texttt{p}_0,\texttt{p}_1,\texttt{p}_2,\texttt{p}_3)
\end{array}
$$

The first four lines show update actions which can be executed at any time due to the fact that there are no guards. The encapsulation operator together with the synchronization rules provided in the μCRL specification however force the update action only to take place if another process executes the corresponding assign action. These update actions make sure that the process always works with the latest values of the variables $\mathtt{p_0}$, $\mathtt{p_1}$, $\mathtt{p_2}$ and $\mathtt{p_3}$.

The process itself ensures that products are added to the turntable. It starts executing once it can synchronize its `start` action with the `start` actions of the other subprocesses. Then if there is no product in slot $\mathtt{p_0}$ available it sends the command to add a product. Once it has received the message that the product adding has finished it stops execution. Notice (this remark also holds for the other four subprocesses) that after executing $\mathtt{stop_2}$ the values of the slots are moved to the next slots of the turntable. In other words, the effect of a rotation is here expressed; each product moves to the next position of the turntable. In the original specification this rotation cannot be found at this place, but it is needed in the translation in order to prepare for the rotation happening in the fifth subprocess.

The last subprocess which runs after the others have stopped executing is called `MAIN_CONTROL5`. Part of its translation is the following:

$$
\begin{array}{l}
\mathtt{proc\ MAIN\_CONTROL5(n : Nat, x : Bool, p : Nat, p_0, \ldots, p_3 : SlotState)} =\\
\quad \ldots\\
\quad \tau.\mathtt{MAIN\_CONTROL5(9, x, p, p_3, p_0, p_1, p_2)} \lhd \mathtt{n} = 8 \rhd \delta\ +\\
\quad \ldots
\end{array}
$$

Rotating the turntable in a 90 degrees turn can be done with one τ action compared to the four actions in the original χ specification. To rotate the table in the χ specification four assignments need to be done to update the slot states. In μCRL it is possible to provide the new values of all four states at once as parameters of a recursive call.

The initialization line Now that all the processes have been translated all that remains is writing the right initialization line. In this line two special operators are used:

- The first one is $|\ \{\mathtt{tick}\}\ |$, which is basically a parallel composition operator which forces `tick` and `tick2` actions of the two processes to synchronize. For a better understanding of this usage of time see the paragraph on the time model in section 5.1.
- The second one is $|\ \mathtt{SVTPC}\ |$ which functions as a parallel composition operator that synchronizes both time (like the operator stated above) and assign / update actions that are needed to use shared variables (for the usage of shared variables see the remark at the beginning of this section).

From the μCRL specification a state space in the .aut format was generated by the μCRL toolset [26]. This state space could then be used by the verification tool CÆSAR ALDÉBARAN Development Package (CADP) [2]. Using this tool one can express properties in the regular alternation-free logic μ-calculus [14]. These properties can then be verified on the state space generated from the model. The CADP tool, together with μ-calculus, was used for verifying the properties of the turntable model.

The state space After having translated the χ specification, the state space of the μCRL specification was generated using the μCRL toolset. This took about 8 seconds, resulting in a state space of 25926 states and 50835 transitions (12957 of them were τ-transitions). After reduction modulo branching bisimulation [34] we ended up with a state space consisting of 4687 states and 7579 transitions, of which 931 were τ-transitions.

When looking at the state space itself, we noticed the following differences:

- While the state space resulting from the χ specification shows assignments as actions, these cannot be found in the state space generated from the μCRL specification. Instead one finds τ actions in these places. In short, μCRL does not have assignment actions. Instead one assigns new values to parameters of a recursion variable. Because by the definition of LPE a recursion variable has to be preceded by an action an assignment is translated to τ.
- In the fifth main controller subprocess of the χ specification multiple assignments were used for rotating the turntable 90°. These assignments together were translated to a single τ action followed by a recursive call containing the new values of the parameters, thus resulting in a smaller state space.
- Special actions that were added to the translation, such as `start` and `stop`, didn't show up in the state space due to the fact that they were hidden. It was chosen to do this because then the state space would become smaller (after reducing modulo branching bisimulation) while these actions are not important for verifying properties of the system anyway. By hiding these actions we also accomplished that the state space generated from the μCRL specification more resembled the state space generated from the χ specification.
- Delays need more states and transitions in the state space generated from the LPEs than in the state space generated from the original χ specification. This is because a delay of n time units takes up n states and n transitions in the state space generated from the LPEs.

The regular alternation-free μ-calculus To express the turntable properties we used the regular alternation-free μ-calculus. In this section a brief introduction to the syntax, that is relevant for understanding the formulas in this chapter, will be given. For a more detailed description see [14].

The regular alternation-free μ-calculus can only be used to express action-based formulas. There exists a more elaborate version of this logic which can express temporal properties involving data values, but that one is not supported (yet) by CADP.

A formula is built by providing a sequence of actions. We can place actions in sequence using the "`.`" operator. Names of actions are placed between double quotes (`"`). At places where the name of the action is not important we can write `true`. So a statement like `"a".true."b"` means "first we encounter an action `a`, then some other action followed by an action `b`".

We can use repetition by using the operator "`*`", which denotes that the statement preceding it can be executed zero or more times.

Finally we have a possibility and a necessity modal operator. The possibility modal operator ("`< >`") is used to express that there exists an execution path in the state space for which the formula in between holds (or does not, depending on the boolean expression following it). The necessity modal operator ("`[ ]`") is used to express that for all execution paths in the state space the given formula holds (or, again, does not, depending on the following boolean expression).

A formula is usually completed by adding a boolean value at the end. This value states that the preceding statement holds or does not hold.

Finally an example: The formula `["a"*.true*."b"]false` expresses that it's not true that in all execution paths in the state space we first find zero or more actions `a`, followed by zero or more other actions, followed again by one action `b`.

Verifying properties Now, using the state space, one can start verifying system properties. Here we look again at the properties initially given in the chapter on the turntable description.

(1) **The system does not contain a deadlock.** The absence of deadlock was verified in the CADP tool, which has this functionality built-in.
(2) **If drilling (testing, adding or removing) is started then it's also finished. The turntable does not rotate in the meantime.** This property is checked using the following μ-calculus formula:

```
[true*."cStartDrill".(not("cStartDrill"
     or"cRotate"))*."cDrillEnded"]true
```

It states that after every `cStartDrill` action eventually `cDrillEnded` can be found without another `cStartDrill` or `cRotate` being called before it.

(3) **If the product has a bad test result it remains on the table and is drilled again (when it comes to the drilling position).** This property can be expressed in μ-calculus as the following formula:

```
                 [true*."cTested(F)".
           (not"cRotate")*."cRotate".
     (not("cRotate"or"cEnvRemoved"))*.
  "cRotate".(not"cRotate")*."cRotate".
    (not("cRotate"or"cStartDrill"))*.
                    "cStartDrill"]true
```

It expresses that in every possible execution path of the state space where you encounter a failed test (`cTested(F)`) you will find later at some point three rotations without the product being removed in between. After that a `cStartDrill` can be found because the product needs to be drilled again.

(4) **If the product has a good test result then the remover will be called to remove the product.** This is represented in μ-calculus by the formula:

```
[true*."cTested(T)".(not"cRotate")*.
         "cRotate".(not"cRotate")*.
              "cEnvCanRemove"]true
```

(5) **No drilling (testing or removing) takes place if there is no product in the slot and no adding can be performed if there is a product in the slot.** Because of the usage of the regular alternation-free μ-calculus one can only verify action-based properties, but state-based properties can in general be checked in an action-based way by first changing the model slightly [34]. In essence what happens is that if you want to check the value of parameter `x` of process `X` you extend the specification of `X` so that it can perform an action having `x` as a parameter that does not have a real effect on the system. For this one can define a special action. Moreover, after executing this action the process returns to the state where it was when starting the action. In other words, the process performs a self-loop. Although this does not have an effect on the behavior of the system, it does provide the ability to see the value of `x` at any given time in the state space, since all states are now equipped with a self-loop of this action with the value of `x` visible as a parameter.

In this case one of the subprocesses of the main control has been equipped with a self-loop executing the action `inslot1`, which tells whether a product is currently in slot 1 or not. Now we can state in μ-calculus:

```
[true*."inslot1(F)"."cStartDrill"]false
```

This expresses that you can never encounter an `inslot(F)` action (there is no product in slot 1) just before `cStartDrill`.

In a similar way one can equip one of the subprocesses of the main control with a self-loop containing the action `inslot0`, which provides info on whether slot 0 contains a product or not. Then we can express in μ-calculus the property that no adding can be performed if there is a product in the slot:

```
[true*."inslot0(T)"."cEnvCanAdd"]false
```

(6) **Every added product is drilled in the next rotation.** This property just means that after every call of `cEnvAdded` (a product has just been added to the table) and one rotation one can always find a `cStartDrill` call. In μ-calculus this leads to:

```
[true*."cEnvAdded".(not"cRotate")*.
        "cRotate".(not"cRotate")*.
               "cStartDrill"]true
```

(7) **Every product eventually leaves the table.** It is a fairness property in the sense that the reachability of the action `cEnvRemoved` is checked in *fair* execution sequences. In μ-calculus the notion of "fair reachability of predicates" is used for fairness, as already stated in chapter 1. To prove this property the model needs to be slightly changed so it supports colored products; a product can be either red or white. This is done by extending the sort `SlotState`. Next the *env_add* process is extended so that it ensures that it will constantly add white products except in at most one instance: Then it adds a red product. This red product can now easily be tracked at any time. Using this changed model it is possible to prove the fairness property.

As an intermediate property, first we prove that in an execution sequence one can never encounter more than one red product:

```
[true*."cEnvAdded(Red)".true*.
      "cEnvAdded(Red)"]false
```

Next using this transformed model one could express the property as follows:

```
    [true*."cEnvAdded(Red)".
  (not"cEnvRemoved(Red)")*]
 <(not"cEnvRemoved(Red)")*.
    "cEnvRemoved(Red)">true
```

This states that once a `cEnvAdded(Red)` action is encountered it is always possible to execute the action `cEnvRemoved(Red)` down the line.

(8) **When a product is added it takes between 21 and 30 time units to get its test result.** This property is checked by using the model with colored products, since we need to track a product to know how many time units it takes to get from having been added to having been tested. This model is changed so that the actions `cEnvAdded` and `cTested` provide us the information (as an argument) what the color is of the product that has been added or tested respectively. In μ-calculus we can then write a number of formulas:

- No paths exist with 20 `tick` action or less between `cEnvAdded(Red)` and `cTested(Red)`;
- There exists a path with exactly 21 `tick` actions between `cEnvAdded(Red)` and `cTested(Red)`;
- There exists a path with exactly 30 `tick` actions between `cEnvAdded(Red)` and `cTested(Red)`;
- No paths exist with 31 `tick` action or more between `cEnvAdded(Red)` and `cTested(Red)`.

The actual formulas will not be presented here as they are rather large and really not that interesting, since they can be created straightforwardly. When verifying these formulas we get the information from CADP that there exists a path with exactly 21 `tick` actions but not one with less. Besides that we also find a path with 30 `tick` actions while we cannot find one with a larger number.

6 Uppaal

6.1 Introduction to Uppaal

Uppaal is a tool for modeling, simulation and verification of real-time systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks. The Uppaal tool creators had two main aims in mind: efficiency and ease-of-use. For the latter reason Uppaal has a well-developed and documented GUI and all formal definitions and semantics are hidden from the user. The formal syntax and semantics are described in [35].

Uppaal allows to make random or directed simulation and verify the models on-the-fly. Different methods can be used to reduce the amount of the memory used during verification. The state space reduction options (conservative or aggressive) allow to generate a smaller state space but require more time. It is also possible to choose different kinds of the state space representation, like difference bound matrices, the compact data structure, under approximation (by bit-state hashing), and over approximation (by convex-hull approximation) [36]. More detailed information about Uppaal can be found in [4,35]. In this document we give a short description of the Uppaal timed automata.

A model in Uppaal consists of a network of timed automata with clocks, invariants, variables over basic data types, guards, handshake synchronization, urgency, and committed locations. Uppaal has a continuous time model (real-valued clocks). However, bounds of clocks and clock reset values must belong to the set of nonnegative integers.

A timed automaton consists of a directed control graph with labels on locations and transitions; every timed automata must have one location marked as *initial*. Locations can be equipped with an invariant. Transitions can carry guards, assignments, and synchronization signals. It is supposed that the invariants, guards, synchronization parts and assignments are always given, in case of absence the invariants and guards can be represented by constant *true*; synchronization and assignment parts can be empty.

All Uppaal timed automata work in parallel. Nested parallelism is not allowed. Uppaal allows rendezvous and broadcast synchronization via the unidirectional channels. Shared variables are used to communicate values from one process to another. Channels declared with prefix "urgent" perform synchronization without delay if synchronization is possible. On synchronizing transition, the assignments on the sender side are evaluated before the assignments on the receiver side.

The scope of the elements (variables, constants, clocks and channels) can be either global, i.e. visible in the whole system, or local, i.e. visible only in the process in which the element was declared. Global variables are declared in the global declaration part and local ones are declared in the declaration part of the corresponding process.

There are four predefined types *(bool, int, clock, chan)* in Uppaal. It is possible to use arrays (including channel arrays and multidimensional ones) and to define a range of integers (ranges reduce the state space).

In this chapter, we explain gradually how the χ processes can be represented as Uppaal timed automata. First, we mention the main problems of the translation. Then, we describe the declaration of the channels, clocks and variables. After that, we give the description of all Uppaal processes explaining how χ process terms and operators were translated. When an element (process or operator) first occurs its translation is explained in detail. We also mention how the resulting timed automata can be simplified.

Translation problems As it was already mentioned, the translation of the turntable model from χ into Uppaal timed automata given in this paper is not formal. In some cases, the ways of translating different χ processes or operators are specific for the turntable model and cannot be considered as generic.

The main problem is the translation of the nested parallelism in the *main_control* process. The nested parallelism can be translated by dividing the process into several parallel ones. The difficulty consists in the fact that *main_control* can continue working only when the parallel processes have finished. That means that we have to use additional shared variables as flags and additional communication. The more detailed description can be found in the *main_control* translation part.

The other difficulty lies in the translation of the maximal progress. In Uppaal a delay can be performed only in locations (it is not a transition). That means that all actions in Uppaal are delayable. To implement maximal progress every transition must satisfy one of the following conditions:

- the transition contains urgent synchronization
- the transition has a clock guard with equality
- the transition has urgent or committed input location

Note that urgent locations and transitions cannot carry clock invariants or guards.

Declaration As it was already mentioned, in Uppaal values cannot be transmitted through channels. That is why we have to declare additional global variables for every channel through which the value is passed. For instance, the global boolean variable *b_UpdateS1* is used to transmit the values through the channel *cUpdateS1*. Note, that in the χ model some channels are used only for synchronization and the values that are transmitted through these channels are not actually used. To improve the readability and reduce the state space these variables are not translated. All the channels are declared as the urgent ones in order to implement maximal progress.

In the χ model we use the variables $p0, p1, p2, p3$ to store the information about the slots and the current product states. To improve the readability of the model we declare the variables as an array ($int[0,4]\ p[4] := \{0,0,0,0\}$). Note, that we define the range of the values that can be stored in the array (from 0 to 4) because it also reduces the state space. This array is declared globally, as well as four additional variables (*bStartP0*, *bStartP1*, *bStartP2*, *bStartP3*). This is done in order to translate nested parallelism; more details can be found in the description of the *main_control* process.

The clocks and local variables are declared locally in the declaration part of the corresponding processes.

The *turn_table* process In the *turn_table* process, the channels are mostly used to transmit values. In that case, the corresponding transition consists of the synchronization part with send or receive signal (for instance, *cS1* ! or *cUpdateS1* ?) and the assignment part (for instance, $b\_S1 := bS1$ or $bS1 := b\_S1$ respectively). In the assignment part the new value is assigned to (or read from) the corresponding global variable. In Uppaal the assignment part of the sender is always executed before the assignment part of the receiver. Note, that the value that is sent through the channels *cEnvAdded* and *cRotate* is always *true*. This allows us to avoid the usage of additional global variables while keeping the behavior of the model unaltered.

After the signal *cRotate* the process must delay. In Uppaal, a delay can be performed only in a location. To translate the delay we have to declare the clock (locally). Before the delay the clock must be reset to *0* in the assignment part of the ingoing transition ($clTurning := 0$). Then, the process is allowed to delay in the location. During the delay the value of the clock is increased and the invariant on the location makes sure that the value of the clock will not exceed the value of the timeout ($clTurning \leqslant 4$). The guard ($clTurning == 4$) on the outgoing transition ensures that the process delays in the location for the exact number of time units and will not leave the location earlier (Figure 3, a).

In order to translate the sequential composition ($cRotate\,?\,bS2; \Delta 4$), the end location of the timed automaton that corresponds to $cRotate\,?\,bS2$ should be merged with the input location of the timed automaton that corresponds to the delay. The resulting timed automaton is depicted in Figure 3, b (note, that the initial location is marked with double circle). As one can see, the merged location is defined as a committed one, this is done to make sure that the

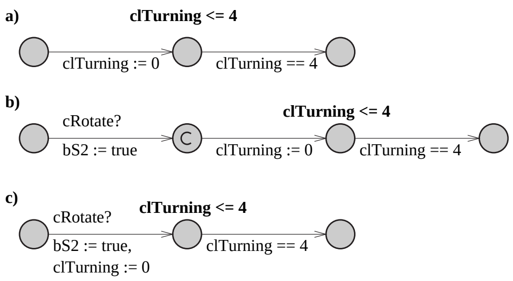


Fig. 3. Delay and sequential composition

process will not delay in this location. In Uppaal several assignments can be combined in the assignment part of the same transition. In this case, they are performed sequentially. This possibility allows us to simplify the automaton (Figure 3, c).

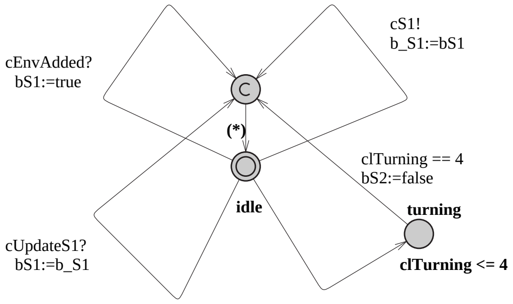


Fig. 4. Alternative composition and repetition

In order to translate the alternative composition of the created timed automata we merged their input locations into the one and did the same with their end locations (Figure 4). From the united input location the process can perform transitions synchronizing with other processes. If there is no synchronization available the process delays in the input location. If several of them are available the choice is made in a non-deterministic way. This behavior corresponds to the alternative composition in χ.

To translate the repetition operator, we need to add a transition that moves the process from its end location to its input location. The transition is marked with $(*)$ in Figure 4. Again, in order not to allow the process to delay in its former end location this location is labeled as a committed one. Knowing that the process must leave this location immediately we can get rid of it. The translation of the part of the *turn_table* process into the Uppaal timed automaton is depicted in Figure 5.

The *clamp* process The translation of the *clamp* process is similar to the translation of the *turn_table* process. Note, that in the case of the sequential

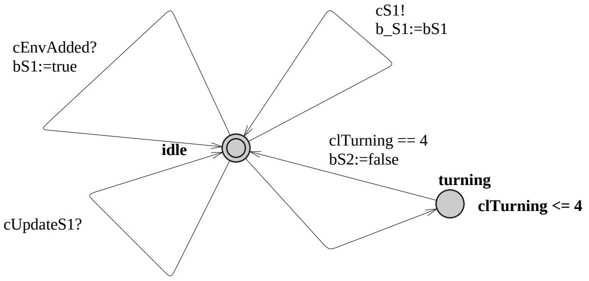


Fig. 5. The translated part of the *turn_table* process

composition of the delay and synchronization ($\Delta 2$; *cLocked* ! true) we cannot combine the clock guard ($clClamp == 2$) with the synchronization *cLocked* ! into one transition. The reason for it is that according to the χ semantics, the process must perform synchronization after the delay if it is possible, otherwise it can delay. On the other hand, in Uppaal it is not possible to use clock guards on the urgent transitions and all channels in the model are declared as urgent to comply with the maximal progress behavior (Figure 6).

Fig. 6. Sequential composition in the *clamp* process

The *drill* process The *drill* process actually consists of two independent processes working in parallel. The first process is responsible for switching the drill on/off and the second one performs drilling. Uppaal does not support nested processes that is why we have changed the structure of the model. There is no variable that is shared by the nested processes in the χ *drill* process (the value of variable x has never been used) and the *clamp* process does not perform any other action. For this reason we can safely divide this process into two processes. The more complicated case of the nested processes and its translation is explained in the description of the *main_control* process.

The *tester* process In the χ *tester* process the atomic process skip is used. In this case skip can be translated as a simple transition without any guards, synchronization or assignment labels. In χ skip cannot delay. To make it non-delayable in Uppaal the input location of the transition should be marked as urgent.

The *main_control* process The *main_control* starts up the parallel adding, drilling, testing and removing operations and waits for their results. This is another case of the nested parallel processes. We cannot translate this process in the same way as we have translated the *drill* process because those four processes are not truly independent. First, they share variables that store the information about the slot states. Second, *main_control* starts them up and after their completion it performs other operations sequentially. To translate this process we have to define four additional processes (*MC_p0* - adding, *MC_p1* - drilling, *MC_p2* - testing, *MC_p3* - removing) that will work in parallel (Figure 7). To start them up from the *main_control* process we use the flags (*bStartP0*, *bStartP1*, *bStartP2*, *bStartP3*) that are declared globally as well as the array where the current slot states are stored.

Main_control checks if the adding, drilling, testing and removing operations can be performed and if they can, it sets the corresponding flags to *true*. The flags are used as guards in the additional processes and as soon as they become *true*, the corresponding synchronization via the channels *cEnvCanAdd, cStartTest, cStartDrill, cEnvCanRemove* is performed. When the additional processes get the signals that the corresponding operations are completed (via the channels *cEnvAddFinished, cDrillEnded, cTested, cEnvRemFinished*), they update the slot states if necessary (Figure 7).

After finishing the additional processes set the flags to *false* and *main_control* can continue the execution of the sequential part. Note, that after starting up the nested processes the main process delays in the location till their completion. For this reason this location cannot be marked as urgent. To implement maximal progress we add synchronization over the urgent channel *cDummy* to the outgoing transition and a "dummy" process that can perform only synchronization on the channel *cDummy*. Note, that in general case the nested processes can be synchronized with the main process by means of additional channels instead of flags.

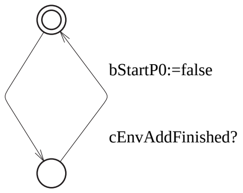


Fig. 7. The example of the additional processes of *main_control*

According to the χ semantics of the guard operator, the guarded process can perform an action if the guard is *true* and the process can perform the action. In order to translate this behavior we need to combine the guard and the guarded process in one transition (see the translation of $p0 = 0 \rightarrow cGetS1\,!\,\text{true}$ in Figure 8).

After finishing the adding and removing operation *main_control* requests the current slot (sensors) states from the turntable controller (*TTC*). *TTC* passes

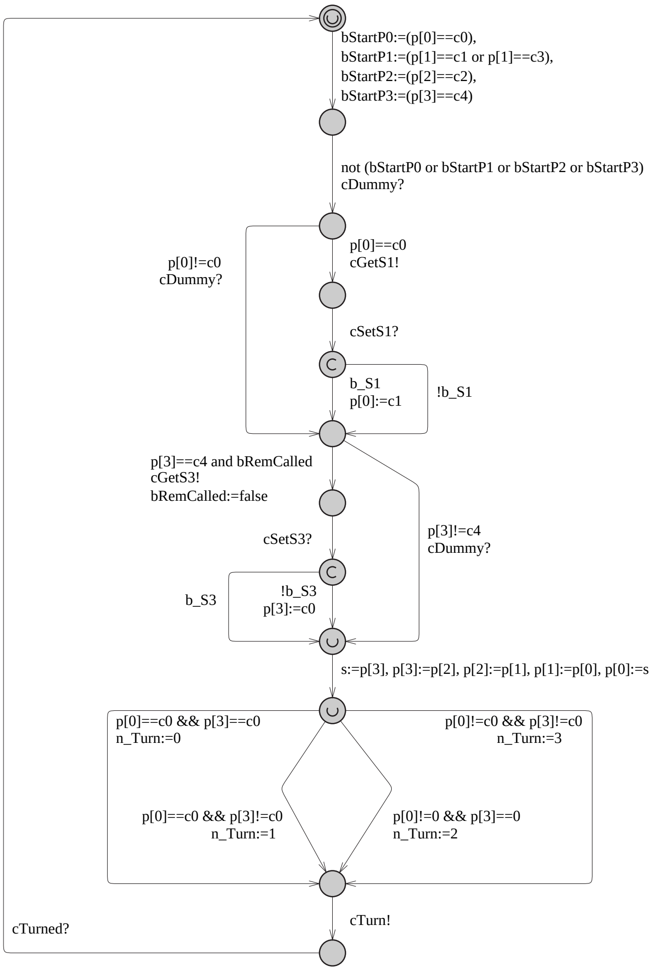


Fig. 8. The *main_control* process

the request to the *turn_table* process and gets the current states ($cS1\,!$, $b\_S1 := bS1$, $cS3\,!$, $b\_S3 := bS3$). After that, it passes those slot states to *main_control* ($cSetS1\,!$, $cSetS3\,!$). Note, that we did not define additional shared variables to pass the current slot states from TTC to *main_control*. Instead, *main_control* reads the shared variables that have been updated by the *turn_table* process. We can do this safely because it is only the *main_control* process that can request to update the variables *b_S1* and *b_S3*, and it can read them only after they have been updated by the *turn_table* process. That means that the situation, when one process wants to read from and another process wants to write into the same shared variable, is not possible. Re-using the same variable for several sequential communications allows us to reduce the state space by decreasing the number of the shared variables and assignments. As it has been explained before, the committed and urgent locations are used in order not to allow the process to perform a delay.

The *drill_control*, *tester_control*, *TTC* and the environment processes The rest of the processes have been translated according to the described rules, so they do not require any additional explanation.

6.3 Verification of the model in Uppaal

The Uppaal model checking engine allows to automatically establish or refute properties that are expressed in the Uppaal Requirement Specification Language. This language is a subset of timed computation tree logic (TCTL), where primitive expressions are location names, variables, and clocks from the modeled system [35]. Uppaal performs verification on-the-fly and it is not possible to learn how many states were generated. The Uppaal developers created a tool named *memtime* to know how much time and how much memory the Uppaal model checker *verifyta* needs. Time and memory consumption depends on the options of the model checker (search order, state space reduction technique, state space representation, state space re-use). Verification of all properties of the turntable system requires 7.988 MB and 56.34 sec, if aggressive state space reduction and breadth-first search order options are used without re-using of the state space. Verification with conservative state space reduction requires 20.527 MB and 55.29 sec. Verification without any state space reduction requires 21.004 MB, 54.42 sec.

The state based properties like "If drilling, testing, adding or removing operation are started the turntable does not rotate in the meanwhile" or "No drilling, testing or removing operation take place if there is no product in the slot and no adding operation can be performed if there is a product in the corresponding slot" can be easily expressed as simple TCTL formulas.

The model checking of properties other than plain reachability (for instance, bounded liveness) might be carried out in Uppaal by means of the test automata [37] or the "decoration" method [38]. The latter can also be used to prove unbounded liveness properties.

(1) **The system does not contain a deadlock.** The absence of a deadlock can be easily proved in Uppaal using the TCTL formula
`A[] not deadlock`

(2) **If drilling (testing, adding or removing) is started then it's also finished and the turntable doesn't rotate in the meantime.** Auxiliary processes *MC_p0*, *MC_p1*, *MC_p2*, *MC_p3*, that we used to translate nested parallelism in the *main_control* process, can be in the *idle* or in progress (*adding*, *drilling*, *testing* or *removing*) locations. When the process is in its progress location that means that corresponding action has been started but is not finished yet. If the process is in its *idle* location that means that it has not been started or it has been already finished.

This property we verify with the formula:

```
A[] ( MC_p0.adding or MC_p1.drilling
      or MC_p2.testing or MC_p3.removing )
    imply not turn_table.turning
```

(3) **If the product has a bad test result it remains on the table and is drilled again.** We can rephrase this property in the following way: a product with a bad test result will be drilled again if not any product with a bad test result can be removed or reach the testing position without being drilled (considering that there is no product loss). First, knowing that product with a bad test result is indicated by the constant 3, we verify that it will not be removed:

```
A[] p[3] == 3 imply not MC_p3.removing
```

Then, we verify that the product with a bad test result will never reach the testing position without being drilled again:

```
A[] MC_p2.testing imply p[2]!= 3
```

(4) **If the product has a good test result then the remover will be called to remove the product.** In order to verify this property we introduce a new boolean variable *bRemCalled*. This variable is set to *false* before removing started and its value is changed to *true* when the remover is called. Then, the value of the guard on the transition before the turn has been changed in a way that if the product in the removing position has a good test result and remover has not been called, the system will be deadlocked. After that, we verify the first property again.

(5) **No drilling (testing or removing) takes place if there is no product in the slot and no adding can be performed if there is a product in the slot.** Knowing that the constant 0 indicates that there is no product in the slot we verify this property using simple formulas:

```
A[] p[1] == 0 imply not MC_p1.drilling
A[] p[0] != 0 imply not MC_p0.adding
```

(6) **Every added product is drilled in the next rotation.** This property can be proved in the way similar to the third property. Knowing that every added product is indicated by the constant 1 and considering that there is no product loss we express and verify this property using the formula

```
A[] MC_p2.testing imply p[2] != 1
```

(7) **Every product eventually leaves the table** Properties with fairness cannot be verified in Uppaal.

(8) **When a product is added it takes between 21 and 30 time units to get its test result.** This is so-called ”bounded liveness” property and it can be verified in Uppaal by the means of test automata or decoration method [38,37]. We have decided to use the decoration method because it requires less changes of the model.

We need to identify the products in order to verify that the product that has been added is tested. We know that there are four slots on the turntable and each of them can contain no more than one product. We also know that the products stay in the same slot till they are removed.

That means that we can use 4 integers to identify slots and if there is a product in the slot it can be identified by the identifier of this slot.

To identify slots we use four integers 0 through 3 that are stored in the array $int[0,3]\ id[4] := 0, 1, 2, 3$. The values stored in the array are the identifiers and the indexes of the array are the positions of the turntable, i.e. $id[2] == 3$ means that the slot (product) with $id == 3$ is in the testing position. Every time the turntable rotates, the slots move as well and the values stored in the array are updated. It is sufficient to verify the property for a product with one particular identifier.

The property "When a product is added it takes between 21 and 30 time units to get its test result" actually consist of two properties: "When a product is added it takes 30 or less time units to get its test result" and "When a product is added it takes 21 or more time units to get its test result".

To verify the first property additional global clock *clDec* and boolean variable *bDec1* with initial value *false* have been declared. Then, we duplicate the transition with communication over the channel *cEnvAdded* in the *turn_table* process. We add the guard $id[0] == 0$ and the assignment $bDec1 := true, clDec := 0$ to one of the duplicated transitions and the guard $id[0]! = 0$ to the other one (the part of the modified *turn_table* process is shown in Figure 9). That means that when the product is added to the slot with $id == 0$, the flag $bDec1$ is set to *true* and the clock $clDec$ is set to 0.

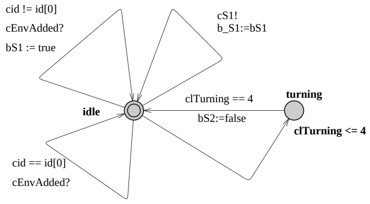


Fig. 9. The part of the modified *turn_table* process

Then, we duplicate the transition with the communication over the channel *cTesterUpDone* in the tester controller, and add the guard $id[2] == 0$ and the assignment part $bDec1 := false$ to one of them. We also add the guard $id[2]! = 0$ to the other one (the part of the modified TC process is shown in Figure 10). That means that the flag $bDec1$ is set to *false* when the test of the product in the slot with the $id == 0$ is completed.

Now we can verify that if the flag $bDec1$ is *true* the clock $clDec$ does not exceed 30 time units.

```
A[] bDec1 imply clDec <= 30
```

The property "When a product is added it takes 21 or more time units to get its test result" is verified using the same approach.

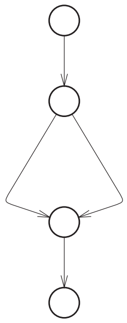


Fig. 10. The part of the modified *TC* process

7 Comparisons and conclusions

In this section we present three tables to give an impression on the level of difficulty concerning different aspects on translating the χ model and verifying its properties. In the first table the number of states and transitions is shown. Note that UPPAAL does not generate the entire state space. The UPPAAL verifier required at most 7.988 MB during verification of all properties. Comparing the sizes of the state spaces generated by SPIN and μCRL we can conclude that the translation to μCRL provides us with a state space which is smaller than the one provided by SPIN, though the state space generated by μCRL requires more memory to be stored.

Table 1
State space comparison

Tool \ **State space**	# states	# transitions	MB used
SPIN	100995	188724	5.8975
CADP(μCRL)	25926	50835 (τ: 12957)	7.332
UPPAAL	-	-	7.988

The second and third table use a grading system which should be read as follows:

- 0: Impossible. Due to differences between the two modeling languages or the limitations of the temporal logic it is impossible to do this.
- 1: Difficult. For Table 2, translation is not straightforward but can be done using special techniques; for Table 3, verification cannot be done straightaway, involves changing the model a lot.
- 2: Needs some work. For Table 2, translation is not completely straightforward, but it does not require special techniques; for Table 3, only slight changes in the model are needed to verify this property.
- 3: Easy. Translation or verification can be done easily.

The second table tells us how difficult it is to translate certain χ constructions in our case study (possibilities not mentioned here do not pose problems for any of the translations). Both translating to Promela and μCRL can be difficult under some circumstances, but translating to Uppaal on the other hand never gets really difficult in our case. These results tell us that, at least concerning the turntable model, Uppaal is the best choice when selecting a language based on the difficulty to translate to this language.

Table 2
Comparison of translation problems

Problems **Language**	Assignments	Delays	Guards	Nested parallelism	Shared variables
Promela	3	2	1	1	3
μCRL	2	1	3	2	1
Uppaal t.a.	3	3	3	2	3

The third table finally shows how difficult it is to express and verify the properties of the turntable using the tools. In this table for each property (using the numbering as in previous parts of this article) the type (**s**afety, **l**iveness, **l**iveness + **f**airness, **l**iveness + **t**ime) is given. What stands out is that property 7 is difficult or even impossible to verify in either tool. Property 8 is very hard to verify in Spin due to the fact that time is hard to be referred to when expressing properties. In Uppaal and CADP verifying this property still needs some work. This is not due to problems with time, but because in order to prove the property at least some products need to be uniquely identifiable. Overall we conclude that CADP provides the least number of problems concerning the verification of the properties.

Table 3
Comparison of the verification

Property **Tool**	1(s)	2(l)	3(l)	4(l)	5(s)	6(l)	7(l+f)	8(l+t)
Spin	3	2	2	2	3	2	1	1
CADP(μCRL)	3	3	3	3	2	3	1	2
Uppaal	3	3	3	2	3	3	0	2

All three formalisms are suitable for the analysis of systems that are originally modeled in χ. When translating we discovered that certain statements have straightforward translations in one language while they have not in another. For example, assignments exist in the same way in χ, Promela and Uppaal, while in μCRL they are represented differently. Additionally, some constructs like, for example, the parallel operator with shared variables inside the process definition, we find very hard to achieve in each language, for different reasons.

To reason about the values of variables (state values) in CADP (using regular, alternation-free μ-calculus) one must extend the model with additional actions that make these values visible. This con of CADP makes the linear temporal logic, built in Spin, and the timed computation tree logic, built in Uppaal, more appropriate when reasoning about states than the regular, alternation-free μ-calculus used in CADP, even though the latter is more powerful. Action-based properties could be verified in Spin as well, by the trace-assertion mechanism. In Uppaal proving such properties can be done using test automata or a decoration technique. When it comes to the fairness principle, μ-calculus and Spin can express it in a very effective way while in Uppaal the use of the fairness principle is impossible.

Finally, the graphical user interface makes modeling and verifying in Uppaal more comfortable than in μCRL. We also find XSpin, a graphical user interface for Spin, very useful.

Related work

- In the article [17] one of the first attempts to verify a χ model by manual translation to DTPromela and applying the model checker DTSpin is described. It dealt with a model of an industrial system and had three objectives. The first objective was to investigate the ability to verify translated χ models with the model checker DTSpin. The second objective was to find out whether there are opportunities to automatically translate χ models into DTPromela. And the third objective was to verify formally some properties of a manufacturing system model.
- In an article by Y.S. Usenko [39] Spin and μCRL are compared using the HAVi leader election protocol. Concerning the generation of a state space for a specification of this protocol, it was concluded that Spin generates states faster, but the resulting state space has more states. On the other hand, according to the article, the state space generation capabilities of Spin and the μCRL toolset cannot be compared due to the differences in the underlying languages. Furthermore, the results may be misleading, so the author tells us, due to the fact that the Promela code was derived from the μCRL code instead of from the informal description. The article ends by saying that a better comparison may be achieved using much smaller case studies.
- The article by H.E. Jensen, K.G. Larsen and A. Skou [40] is on comparing Spin and Uppaal using a collision avoidance protocol as a case study. In the paper it is indicated that it is possible to model real-time systems and their broadcast behavior in Uppaal and it cannot be done in Spin. The kind of properties expressible in the Uppaal requirement specification language are restricted to invariance and possibility properties. It is possible to verify bounded liveness properties in Uppaal, though they need to be expressed as separate test automata.

Future work

- Formulating general translation schemes from χ to input languages for model checkers;
- Investigating how current stochastic tools can be used to determine performance characteristics of χ models and comparing the results with those coming from the χ simulator;
- Investigating theorem proving capabilities of different verification tools, with respect to χ.

References

[1] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Syntax and Formal Semantics of Hybrid Chi. Technical report, Eindhoven University of Technology, Department of Computer Science, 2004. To be published.

[2] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *Proceedings 8th Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440, 1996.

[3] G.J. Holzmann. *The SPIN model checker.* Addison-Wesley, 2003.

[4] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[5] D.A. van Beek, A. van der Ham, and J.E. Rooda. Modelling and Control of Process Industry Batch Production Systems. In *15th Triennial World Congress of the International Federation of Automatic Control*, Barcelona, Spain, CD-ROM, 2002.

[6] V. Bos and J.J.T. Klein. *Formal Specification and Analysis of Industrial Systems.* PhD thesis, Eindhoven University of Technology, 2002.

[7] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. A Hybrid Language for Modeling, Simulation and Verification. In *IFAC Conference on Analysis and Design of Hybrid Systems*, Saint-Malo Brittany, France, 2003.

[8] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Formal semantics of hybrid chi. In *First International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science, Marseille, France, 2003.

[9] TIPSy Project Website. http://se.wtb.tue.nl/~awijs.

[10] W.J. Fokkink, J.F. Groote, and M. Reniers. Modelling Distributed Systems. Unpublished manuscript, 2002.

[11] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[12] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[13] W.J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2000.

[14] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[15] R. Alur, C. Courcoubetis, and D. Dill. Model Checking in Dense Real-Time. In *Proc. of the 5th IEEE Symposium on Logic in Computer Science*. LICS, 1990.

[16] A.T. Hofkamp and H.W.A.M. van Rooy. *Embedded Systems Laboratory Exercises Manual*, 2003.

[17] V. Bos and J.J.T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.

[18] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Discrete-event subset of χ_{σ_h}. Eindhoven University of Technology.

[19] R. Gerth. Concise Promela Reference. Online document: http://spinroot.com/spin/Man/Quick.html.

[20] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[21] B.W. Kernighan and D.M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[22] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[23] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[24] D. Bošnački. *Enhancing State Space Reduction Techniques for Model Checking*. PhD thesis, Eindhoven University of Technology, 2001.

[25] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, Chichester, Stuttgart, 1996.

[26] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In *Proceedings 13th Conference on Computer Aided Verification (CAV2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254, 2001.

[27] S. Owre, J.M. Rushby, and N. Shankar. PVS: a Prototype Verification System. In *Proceedings 11th Conference on Automated Deduction (CADE'92)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, 1992.

[28] W.J. Fokkink, J.F. Groote, J. Pang, and B. Badban. Verifying a Sliding Window Protocol in μCRL. Technical Report SEN-R0308, CWI, 2003.

[29] J.F. Groote, F. Monin, and J.C. van de Pol. Checking verifications of protocols and distributed systems by computer. In *Proceedings 9th Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 629–655, 1998.

[30] B. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002.

[31] J.F. Groote. The Syntax and Semantics of timed μCRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.

[32] S.C.C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In *Andrei Ershov Fifth International Conference Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 178–192, 2003.

[33] M. Bezem and J.F. Groote. Invariants in Process Algebra with Data. In *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416, 1994.

[34] R. De Nicola and F. Vaandrager. Three Logics for Branching Bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

[35] M.O. Möller. *Structure and Hierarchy in Real-Time Systems.* PhD thesis, University of Aarhus, 2002.

[36] F. Larsson, K.G. Larsen, P. Pettersson, and W. Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997.

[37] L. Aceto, P. Bouyer, A. Burgueño, and K.G. Larsen. The Power of Reachability Testing for Timed Automata. *Theor. Comput. Sci.*, 300(1-3):411–475, 2003.

[38] M. Lindahl, P. Pettersson, and W. Yi. Formal Design and Analysis of a Gear Controller. *IEEE Transactions on Software Engineering*, 3:353–368, 2001.

[39] Y.S. Usenko. State space generation for the HAVi leader election protocol. *Sci. Comput. Program*, 43(1):1–33, 2002.

[40] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and Uppaal. In *Proceedings of the 2nd SPIN Workshop*, Rutgers University, New Jersey, USA, 1996.