Verifying χ Models of Industrial Systems with Spin

Nikola Trčka

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

Abstract

The language χ has been developed for modeling of industrial systems. To obtain performance measures, its simulator has been successfully used in many industrial areas. However, for functional analysis, simulation is less applicable. Such analysis can be done in other environments. In this paper we present guidelines and techniques for translating χ specifications to PROMELA, the input language of the well known model checker SPIN. We highlight the differences between the two languages and show, in a step by step manner, how some of them can be resolved. We conclude by giving the translation scheme and apply it to a small industrial case study.

Key words: industrial systems, verification, the modeling language χ, the model checker SPIN

1 Introduction

When designing industrial systems (machines, manufacturing lines, warehouses, factories, etc.) engineers make frequent use of simulation models to detect flaws and to optimize performance. The language χ has been developed especially for this purpose. It allows for the specifying of both discrete-event and continuous aspects of industrial systems and its simulator has been successfully applied to a large number of industrial cases, such as a car assembly line (NedCar [12]), a multi-product, multi-process water fab (Philips [7]), a brewery (Heineken), a fruit juice blending and packaging plant (Riedel [10]) and process industry plants ([1]). Simulation is a powerful technique for performance analysis, like calculating throughput and cycle time, but it is less suitable for functional analysis (sometimes called verification). It can for instance reveal that a system has a deadlock (it is unable to proceed) or that it sometimes has a certain behavior, but it cannot show that the system is deadlock-free nor that it always has a certain behavior.

A most widely used verification technique today is model checking. This technique performs an exhaustive search of the state space checking if a certain property of the system holds. The property is represented as a formula of some temporal logic, a logic that allows us to say things like: if a machine is given input then it will eventually produce a correct output. There are many variants of these logics (consult e.g. [22]). They can be linear (reasoning is about a single sequence of states) or branching (reasoning involves several different branches starting from a state), action based (reasoning is about what action can be performed in a state) or state based (reasoning about the value of variables in a state), etc. Once the property is stated, model checking becomes a completely automated process.

To facilitate model checking, χ has to be turned into a formal method. Indeed, recently, a big part of the discrete-event subset of the language was formalized (the formalization is called χ_σ); a structural operational semantics and a notion of equivalence were defined [5]. Moreover, a state space generator, a tool that generates all possible behaviors of the model, was built. Properties of the model, represented as alternation free μ-calculus formulas (see [22] again), can then be verified by the model checker of the CADP toolset [9]. However, there are some problems concerning this approach. First, the current toolset is a prototype; to be used in real-world applications, it would need to be optimized. Second, χ has (the notion of states of) variables but their values are not kept in the state space and therefore cannot be used in the temporal logic formulas, i.e. model checking must be action based. To a large extent, formulas referring to values of variables can be verified with action based model checking but this always requires big changes of the model, e.g. adding of actions that notify that a variable has a certain value. In addition, not all of the temporal logic formulas have equivalents in the alternation free μ-calculus [18]. The third (and probably the biggest) problem is that the (usually very large) state space must be generated completely before a verification within CADP can take place.

To solve some of these problems we consider verification of χ models in some other environments. Also, an extra motivation is that developers of χ are currently redesigning and extending the language [21] but have no intention to build any other tools besides a simulator. In particular, they do not intend to implement an optimized state space generator. Rather, their idea is to facilitate the verification of χ models by establishing a connection with other state-of-the-art verification tools and techniques.

The aim of this paper is to present techniques for translating general χ specifications to Promela, the input language of the popular model checker Spin. Aspects in which χ and Promela differ from each other are given in a step by step manner and treated in detail. For each aspect, difficulties of translation are discussed, pitfalls and solutions presented and explained. We cover many important features of χ such as time, nested parallelism, urgency etc. that are usually present in models of industrial systems. We also (syntactically) define a subset of χ models that can be translated to PROMELA and present a translation scheme.

Our work can be seen as an extension of that presented in [4] and [3]. In [4], the authors present a translation of a χ model of a turntable machine to PROMELA and verify properties, like the absence of deadlock and no product loss, with SPIN. The focus is on the verification and not on the translation; general guidelines for translating arbitrary χ models to PROMELA are not provided. In [3], a more detailed model of the same machine is translated to PROMELA, μCRL [2] and UPPAAL timed automata [19]. Even though this paper shows some techniques and difficulties of the translation to PROMELA, its aim is to compare the different approaches for the functional analysis of χ models, and to a lesser extent on the aspect of translation.

Note that here we take an informal approach to the problem. In [20] we support a part of our translation with a formal correctness proof. There we define a notion of equivalence for (a slightly different version of) χ, prove that it is a congruence and that it preserves validity of temporal logic formulas. Then, we identify a subset of χ that maps to PROMELA more straightforwardly and show how a bigger subset can be reduced, modulo the equivalence, to that form.

The structure of the paper is as follows. In Section 2 we give an introduction to χ; its syntax and (informal) semantics. As an illustration we present a χ model of a small manufacturing line. In Section 3 we briefly introduce PROMELA, pointing out features that we need. Section 4 is the main section of this paper. There we explain how we deal with the aspects of χ that are uncommon to PROMELA: parallelism, scoping and complex data types, deadlock and immediate termination, guarded processes and time. For each feature we show what the problems of translating it to PROMELA are and how we can circumvent them. Then, we present a translatable subset of χ and a translation scheme. At the end of the section we illustrate the translation process by showing the PROMELA translation of the model of a manufacturing line introduced in Section 2. We also show the usefulness of our approach by proving a simple property of the line. In the last section we give some conclusions.

1.0.1 Acknowledgements

I thank Bas Luttik for useful comments.

2 The χ Language

The redesign of the χ language (including features to model hybrid behavior) is still work in progress [21]. Since a large part of the discrete-event subset of the new version of χ can be expressed in χ_σ, we take χ_σ as our starting point.

2.1 Syntax and semantics of χ_σ

Here we give a short and informal introduction to the language. We refer to [5] for a complete syntax definition and a formal semantics of the language constructs.

2.1.1 Data types

The basic data types of χ_σ are booleans, natural, integer and rational numbers and (typeless) channels. Real numbers are not supported. Most of the usual constants, operators and relations are defined for every data type and can be used together with variables to build expressions. Furthermore, χ_σ provides a mechanism to build tuples, lists and sets from the basic types (but not channels).

2.1.2 Time model

The time domain in χ_σ is dense, i.e. timing is measured on a continuous time scale. The weak time determinism principle, sometimes called the time factorization property (time does not make a choice), is implicitly adopted. Maximal progress (a process can delay only if it cannot do anything else) can be enforced by an operator. Time additivity (if a process can delay first t_1 and then immediately t_2 time units then it can delay $t_1 + t_2$ time units from the start) is not implemented. Delaying is enforced by the delay operator but some processes can also implicitly delay (see the next paragraph).

2.1.3 Atomic processes

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

(1) The empty process (ε), the process whose only option is to terminate successfully.

(2) The deadlock process (δ). It cannot execute any action, it cannot delay and it does not have the option to terminate successfully.
(3) The *skip* process. It performs the internal action τ. It cannot delay.
(4) The delay process (Δe). It delays any number of time units less or equal than the value of the expression e.
(5) The assignment process ($x := e$). It assigns the value of the expression e to the variable x. It does not have the possibility to delay.
(6) The send process ($m!e$). It stores the value of the expression e in the channel m (which behaves as one-place buffer). It is able to delay arbitrarily long.
(7) The receive process $m?x$. If the channel m is not empty, then it assigns the value in m to the variable x; otherwise it blocks. It is also delayable.

2.1.4 Operators

There are nine operators in χ_σ. We present each one of them together with their (informal) semantics.

(1) The guard operator ($:\rightarrow$). A process $b :\rightarrow p$ behaves as p only 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 zero or more times.
(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. If one of the processes can execute the *send* action and the other one can execute the *receive* action on the same channel then they can also communicate; in other words $p \| q$ can also execute a 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 (λ_s) or as $v : s'$ where s' is a state and v is a variable declaration ($x : type[\mapsto c]$) or a channel declaration ($\tilde{~}m$).
(7) The encapsulation operator (∂_A). A process $\partial_A(p)$ disables all actions of p that occur in the parameter set A. Most of the time, the set A contains only send and receive actions to enforce a rendez-vous (synchronous) communication.
(8) The maximal progress operator (π). A process $\pi(p)$ behaves as p only that it can delay only if p can delay and p cannot execute any action.
(9) The abstraction operator (τ_I). A process $\tau_I(p)$ 'hides' (renames to τ) all actions of p that occur in the parameter set I.

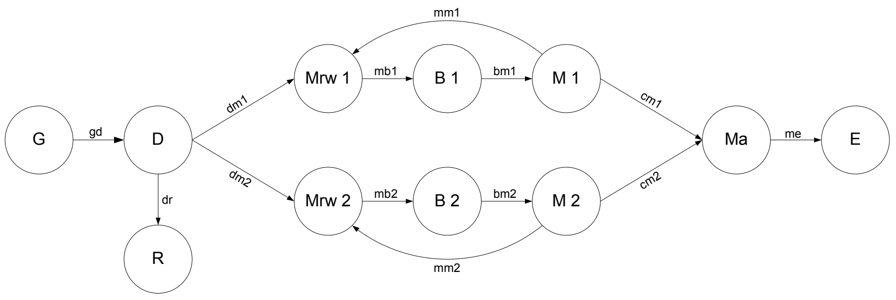

Fig. 1. Components of the manufacturing system

Note that χ_σ provides process definitions but this part of the language is not formalized so we do not consider it here.

2.2 A manufacturing line in χ_σ

To give an impression of the language we give an example, a slight modification of the one given in [21]. Consider a manufacturing line that consists of a generator, a distributor, a rejector, two manufacturing cells, an assembly machine and an exit process. The generator (G) generates products every 7 time units and delivers them to the distributor. The distributor (D) waits 5 time units for one of the cells to be ready and then sends it a product. If in 5 time units none of the cells are ready the distributor sends a product to the rejector. Each manufacturing cell consists of two machines (Mrw and M) and a 3-place buffer in between (B). Every product is processed by the cell twice. After processing, products are sent to the assembly machine (Ma) where they are processed further, combined and sent to the exit process (E). Every machine takes 4 time units to perform its operation. The whole system is pictured in Fig. 1.

We now present a χ_σ model of the line:

$$
\begin{aligned}
&\pi(\partial_A(\llbracket {\sim}gd : {\sim}dc1 : {\sim}dc2 : {\sim}dr : {\sim}cm1 : {\sim}cm2 : {\sim}me : {\sim}mb1 : {\sim}bm1 : {\sim}mm1 : {\sim}mb2 : {\sim}bm2 : {\sim}mm2 : \lambda_s \mid \\
&\quad \llbracket x : bool \mapsto false \mid \ (\Delta 7 \,;\, gd!x)^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool \mid \ (gd?x; (dm1!x \,[]\, dm2!x \,[]\, (\Delta 5 \,;\, dr!x)))^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool \mid \ (dr?x)^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool \mid \ ((dm1?x \,[]\, (mm1?x \,;\, x := true)) \,;\, \Delta 4 \,;\, mb1!x)^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool \mid \ ((dm2?x \,[]\, (mm2?x \,;\, x := true)) \,;\, \Delta 4 \,;\, mb2!x)^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool, bf : list[bool] \mapsto [\,] \mid (len(bf) < 5 :\rightarrow (mb1?x \,;\, bf := bf +\!\!+ (x : [\,])) \\
&\qquad\qquad\qquad\qquad\qquad\qquad [] \ \ len(bf) > 0 :\rightarrow (bm1!hd(xs) \,;\, bf := tl(bf)))^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool, bf : list[bool] \mapsto [\,] \mid (len(bf) < 5 :\rightarrow (mb2?x \,;\, bf := bf +\!\!+ (x : [\,])) \\
&\qquad\qquad\qquad\qquad\qquad\qquad [] \ \ len(bf) > 0 :\rightarrow (bm2!hd(xs) \,;\, bf := tl(bf)))^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool \mid \ (bm1?x \,;\, \Delta 4 \,;\, (x :\rightarrow cm1!x \,[]\, \neg x :\rightarrow mm1!x))^* \,;\, \delta \rrbracket \\
&\parallel \llbracket x : bool \mid \ (bm2?x \,;\, \Delta 4 \,;\, (x :\rightarrow cm2!x \,[]\, \neg x :\rightarrow mm2!x))^* \,;\, \delta \rrbracket
\end{aligned}
$$

$\| \, [\![x, y : bool \mid \; ((cm1?x \parallel cm2?y) \,;\, \Delta 4 \,;\, me!(x \wedge y))^* \,;\, \delta]\!]$
$\| \, [\![x : bool \mid \; (me?x)^* \,;\, \delta]\!] \;]\!]))$

Every parallel component represents a part of the system. The order is: G, D, R, Mrw_1, Mrw_2, B_1, B_2, M_1, M_2, Ma and E. A global scope is used for the declaration of channels that represent connection between parts. The set A contains all send and receive actions. Variable x models products. It is a boolean variable so that we can distinguish cases when the product was not processed by the cells yet (x is *false*) from when it is already processed once (x is *true*). Symbol $[\,]$ denotes an empty list, $x : [\,]$ means that x is added to the empty list, and the operator $+\!\!+$ concatenates two lists.

3 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 [15,11,16] or consult SPIN's web page `http://spinroot.com`.

PROMELA's syntax is derived from C [17], with communication primitives from CSP [14] and control flow statements based on the guarded command language [8]. 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 C-like macros. Message channels are declared, for instance, as `chan m = [2] of {int}` meaning that the channel is buffered and that it can store (at most) two values of (its field's) type integer. Channels can 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, global, write-only variable of type integer, used to store some input values that are of no importance later. It is an error to use or reference its value.

Process declarations are of this form:

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

    statement
 }
```

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. Any expression is also a statement, executable 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.

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

```
 if
  :: statement_1
    ...
  :: statement_n
 fi .
```

It nondeterministically selects among its options an executable statement and executes it. A selection blocks until there is at least one selectable option.

The repetition is achieved by the statement:

```
 do
  :: statement_1
    ...
  :: statement_n
 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 a `break` or `goto` statement. The `break` statement 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 [6]. 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 limitations: statements other than the first may not block.

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. With the prefix `active`, a process definition is considered initially active and need not be started by the `init` process. 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.

Properties of a PROMELA model are expressed in a state-based linear temporal logic and verified by SPIN using the on-the-fly method of model checking (a property is checked while the state space is building, not after it is already built).

4 Translating χ_σ to Promela

First we introduce some mild assumptions about the χ processes we consider for translation. SPIN is a state based model checker and hiding of actions does not play a role so we assume that our models do not contain the τ_I operator. In addition, because the main form of communication in χ_σ is synchronous, we assume that the encapsulation operator (∂_A, with A the set of all send and receive actions) is applied to our process. Since there is no explicit encapsulation in PROMELA, we do not allow ∂_A to occur anywhere else. The last assumption concerns timing. Because maximal progress in SPIN is implicit, our process is prefixed also by the π operator. This is the only place where π is allowed (except for one special case, see Remark 1 below). To summarize, we consider only processes of the form $\pi\partial_A(p)$ where p does not contain abstraction, encapsulation nor the maximal progress operator (experience shows that most χ_σ specifications of industrial systems are of this shape). From now on when we refer to the process we translate, we mean p.

4.1 Techniques

Translation of some χ_σ constructs, like assignments, the *skip* statement, sequential and alternative composition is straightforward since they have direct equivalents in PROMELA, but there are several important aspects in which χ_σ and PROMELA differ from each other. Translation of such features requires a detailed explanation. This will be presented in the following paragraphs.

4.1.1 Parallelism

As said before, process definitions from a PROMELA specification are implicitly executed in parallel and there is no (explicit) parallel operator. This means that our χ_σ process can have parallelism only on top level, i.e. in general, we must disallow nested parallel operators. In some cases however, there are techniques to deal with nested parallelism and we discuss them now.

Note that a sequential composition can be simulated by a parallel composition at the expense of introducing an extra synchronization variable. Thus process $(p \parallel q) \,;\, r$ is equal to

$$[\![w : nat \mapsto 0 \mid p \,;\, w := w + 1 \parallel q \,;\, w := w + 1 \parallel w = 2 :\rightarrow r]\!]$$

and similarly $p \,;\, (q \parallel r)$ is equal to

$$[\![w : bool \mapsto false \mid p \,;\, w := true \parallel w :\rightarrow q \parallel w :\rightarrow r]\!] \,.$$

This technique can easily be extended from two to an arbitrary number of parallel components.

If parts of a process that run in parallel do not communicate with each other, the parallel operator is just an interleaving operator. In both χ_σ and PROMELA interleaving of atomic processes can sometimes be achieved with one loop and a few additional guards (boolean variables). The idea is to associate one guard to each atomic process. If there is a choice between two atomic processes then they share the same guard. Only atomic processes available from the start have their guards initially set to *true*. When an atomic process is executed, its guard is put to *false* and the guard of the atomic process that comes next is assigned *true*. This is done in a loop that is exited when all the guards are *false*. Note that this does not work when there is a * operator involved.

We illustrate the technique with an example. Suppose a,b,c,d and e are atomic processes. Then, process $a \,;\, b \parallel c \,;\, (d \,[]\, e)$ is transformed to:

$$
\begin{array}{l}
[\![\; b_1 : \mathit{bool} \mapsto \mathit{true}, b_2 : \mathit{bool} \mapsto \mathit{false}, b_4 : \mathit{bool} \mapsto \mathit{false}, b_3 : \mathit{bool} \mapsto \mathit{true} \mid \\
\quad (\; b_1 :\rightarrow a \,;\, b_1 := \mathit{false} \,;\, b_2 := \mathit{true} \\
\quad [] \; b_2 :\rightarrow b \,;\, b_2 := \mathit{false} \\
\quad [] \; b_3 :\rightarrow c \,;\, b_3 := \mathit{false} \,;\, b_4 := \mathit{true} \\
\quad [] \; b_4 :\rightarrow d \,;\, b_4 := \mathit{false} \\
\quad [] \; b_4 :\rightarrow e \,;\, b_4 := \mathit{false} \\
\quad) * ; \neg(b_1 \wedge b_2 \wedge b_3 \wedge b_4) :\rightarrow \varepsilon \;]\!].
\end{array}
$$

Note that this solution introduces many additional assignments and therefore enlarges the state space of a process. When translating the example to Promela one can put a guarded command and the assignments following in a `d_step` statement.

So far, when dealing with nested parallelism, we worked mostly within χ_σ, not considering features present only in Promela. A good candidate for dealing with nested parallelism is Promela's process creation mechanism, i.e. the `run` statement. The process $p \parallel q$ is translated to

```
atomic { run(p); run(q) }
```

and p and q become separate process definitions in Promela. However, this must be used with great care. If the parallel composition is in a choice context, e.g. $(p \parallel q) \,[]\, r$, then it cannot be translated as:

```
if
 :: atomic { run(p); run(q) }
 :: r
fi ,
```

For in this Promela specification the choice does not depend on the executability of $p \parallel q$; the `run` statement is always executable.

4.1.2 Data Types

From the set of basic data types the χ_σ specification can contain channels, booleans, natural numbers and integers. Rational numbers are not supported by Promela. Every undeclared variable must be translated to the dummy variable. Translation of *bool*, *nat* and *int* variable declarations is straightforward but translation of channel declarations requires more explanation. First, in Promela, forcing the communication on some channel to be handshake communication is automatically done if the channel is declared of zero length. Second, since all channels in χ_σ are typeless, in order to translate (declarations of) them to Promela we must first determine their type from the type of the receiving variable. This means, if we declare a channel as $\tilde{\ }m$ and use it, for example, as $m!1$ and $m?x$, where x is declared as integer, then in Promela m should be declared as

```
 chan m = [0] of {int}
```

and used as `m!1` and `m?x`. Using channels to transport data of different types is therefore not allowed.

Complex data types can be implemented using Promela's support for records, arrays and macro definitions. For example, a tuple of an integer and a boolean value (declared as $tuple[int, bool]$ in χ_σ) is represented as

```
 typedef TUPLE_INT_BOOL {
    int elem_1;
    bool elem_2;
} .
```

To model lists we can use buffered channels (a similar approach was taken in [13]). A list of length n is defined as a tuple of a channel `l` of capacity `n` (the actual list) and a variable `head` that holds the first element of the list. Adding an item to a list is represented as sending it to a channel that represents the list. Transforming a list into its tail is done by receiving an element from this channel. To keep the `head` variable up-to-date, we use a predefined Promela function `len` that returns the length of a channel and a variant of the send statement (`m?<x>`) that behaves as `m?x` only that the message from the channel `m` is not erased upon executing. A list of `n` integers is represented as:

```
 typedef LIST_INT {
    chan l = [n] of {int};
    type head = 0;
 } .
```

To make the usage of lists simpler and closer to χ_σ syntax we define four macros. They represent some usual functions on lists (note that χ_σ has more): `add(x,lst)` adds `x` to the list `lst`, `hd(lst)` returns a first element of `lst`, `tail(lst)` transforms the list `lst` into its tail and `length(lst)` gives the length of `lst`.

```
#define add(x,lst) d_step{ lst.l!x;
                           if
                            :: len(lst.l) == 1 -> lst.head = x
                            :: else
                           fi
                         }

#define hd(lst) (lst.head)

#define tail(lst) d_step{ lst.l?_;
                          if
                           :: len(lst.l) > 0 -> lst.l?<lst.head>
                           :: else
                          fi;
                        }

#define length(lst) (len(lst.l)) .
```

4.1.3 Scoping

In PROMELAthere are only two scope levels. Process local, in process declarations, and global, outside of them. It is not possible to introduce blocks with block-local variables inside the process declarations. This is not a serious limitation because for almost every process we can always find an equivalent one of the form $[\![s \mid [\![s_1 \mid p_1]\!] \mid\mid \ldots \mid\mid [\![s_n \mid p_n]\!]]\!]$ where the p_i's do not contain the scope operator.

First note that $[\![\lambda_s \mid p]\!]$ is equivalent to p and that $[\![s_1 \mid [\![s_2 \mid p]\!]]\!]$ is equivalent to $[\![set(s_1, s_2) \mid p]\!]$ (where `set` is a function that adds variables from `s_2` to `s_1`, overwriting those already present in `s_1`). This allows us to eliminate scope when its declaration section is empty or when it is immediately nested.

Further, it is not hard to prove that, when q does not contain free variables (a variable is free in q if it is not used within a scope that declares it) that are declared in s, then $[\![s \mid p]\!] \circ q$ is equivalent to $[\![s \mid p \circ q]\!]$ for all $\circ \in \{;, [\!], \mid\mid\}$. Similarly, $b :\rightarrow [\![s \mid p]\!]$ is the same as $[\![s \mid b :\rightarrow p]\!]$ when b does not contain variables also declared in s, and $p \,;\, [\![s \mid q]\!]$ is the same as $[\![s \mid p \,;\, q]\!]$ when the free variables of p are not declared in s.

Elimination of a scope in the context of a repetition is more complicated. Note that the process $[\![s \mid p]\!]^*$ has different behavior than $[\![s \mid p^*]\!]$. This is because p in $[\![s \mid p]\!]^*$, when it has finished executing, starts again in the 'fresh' state s while p in $[\![s \mid p^*]\!]$ starts from a possibly changed state. A solution is to

Table 1
Elimination of scopes

$[\![\lambda_s \mid p]\!]$	p
$b :\longrightarrow [\![s \mid p]\!]$	$[\![s \mid b :\longrightarrow p]\!]$
$[\![s \mid p]\!] ; q$	$[\![s \mid p ; q]\!]$
$p ; [\![s \mid q]\!]$	$[\![s \mid p ; q]\!]$
$[\![s \mid p]\!] [\,] q$	$[\![s \mid p [\,] q]\!]$
$[\![s \mid p]\!] \parallel q$	$[\![s \mid p \parallel q]\!]$
$[\![s_1 \mid [\![s_2 \mid p]\!]]\!]$	$[\![set(s_1, s_2) \mid p]\!]$
$[\![s \mid p]\!]^*$	$[\![s \mid (p ; x_1 := c_1 ; \ldots ; x_n := c_n)^*]\!]$

make p restore the old state when it is done. In other words, if s is of the form $x_1 : type_1 \mapsto c_1, \ldots, x_n : type_n \mapsto c_n, \tilde{}m_1, \ldots, \tilde{}m_k$, we transform $[\![s \mid p]\!]^*$ to $[\![s \mid (p ; x_1 := c_1 ; \ldots ; x_n := c_n)^*]\!]$. If some of the x_i's are not initialized (i.e. the part $\mapsto c_i$ is missing) we simply omit $x_i := c_i$.

The summary of all the transformations that (after adequately renaming variables) can be used for nested scopes elimination is given in Table 1.

4.1.4 Deadlock and Immediate Termination

The deadlock process δ has an equivalent in PROMELA, the boolean expression/statement `false`. The only difference is that, in PROMELA, deadlock is delayable until all timers are off. On the other hand, there is no notation to express (successful) immediate termination in PROMELA. Our process, therefore, cannot contain ε. To illustrate that ε in general cannot simply be translated to the process *skip* we give an example. The χ_σ process $skip [\,] \varepsilon ; \delta$ must always execute the *skip* action while the PROMELA statement

```
if
 :: skip
 :: skip;false
fi
```

can also execute the second *skip* and deadlock afterwards.

In some cases it is possible to remove the ε from a specification; using for instance that $\varepsilon ; p$, $p ; \varepsilon$, $p \parallel \varepsilon$ and $p \parallel \varepsilon$ are all equivalent to p and that $(p [\,] q) ; r$ is the same as $p ; r [\,] q ; r$.

The problem with immediate termination is also hidden in the repetition. In χ_σ, the behavior of the process p^* is 'execute p *zero* or more times' which means that it has an option to terminate immediately. Since p^* is equivalent to $p;p^* \,[]\, \varepsilon$, we can think of it as it has a hidden ε. In PROMELA, the repetition operator `do :: od` cannot terminate without executing an action (`break` or `goto`). Therefore, it is not possible to translate p^*. However, process $p^* \,;\, q$ (meaning q or p zero or more times and then q), if q does not terminate immediately, has an equivalent in PROMELA:

```
do
 :: p
 :: q; break
od .
```

Hence, in the process we can translate, every occurrence of * must be followed by the sequential operator.

Note that an infinite repetition of a process p in χ_σ is expressed as $p^* \,;\, \delta$. Its translation to PROMELA is

```
do
 :: p
 :: false; break
od .
```

Because the expression/statement `false` is never executable the latter statement is equivalent to

```
do
 :: p
od ,
```

which is a usual way of presenting an infinite behavior in PROMELA.

Remark 1 *Although* `break` *and* `goto` *statements should not be considered as actions but only as control flow mechanisms,*

```
do
 :: p
 :: break
od
```

(similarly with `goto`*) is not a right translation of the process p^*. This is because the* `break` *statement is still involved in a choice. The previous example is an illustration of this. If $p^* \,;\, \delta$ were translated as*

```
do
 :: p
 :: break
od;
false ,
```

SPIN *could always decide to change the control to the outside of the loop and deadlock.*

4.1.5 Delays

DTPROMELA is a discrete time extension so we require delays to be natural numbers. For rational ones there is always a number we can multiply all of them by, and obtain natural delays of the same ratio. The Δe statement is translated to the DTPROMELA statement `expire(t)`, where `t` is of type `timer` and is previously set to the value of e. For each Δ statement a new timer should be introduced. In cases where Δe is not involved in a choice, `set(t,e)` can be present immediately before the `expire(t)` (there is a PROMELA macro `delay(t,e)` defined as `set(t,e); expire(t)` that can be used instead). However, when there is a choice of Δe and another process we have to be more careful. If, for example, we translate $\Delta e \ [] \ p$ as

```
if
 :: set(t,e); expire(t)
 :: p
fi ,
```

then, because `set(t,e)` is always executable, SPIN can choose to execute it. For example, if `p` can do a send or receive action then we lose an option to communicate which contradicts the fact that send and receive processes are delayable and that alternatives delay together. Also, if `p` can do an assignment action, SPIN should not execute `set(t,e)` because of the maximal progress. Similar problem appears in processes of the form $\Delta e^* \ ; p$.

To prevent time from making a choice `set(t,e)` must be moved before the alternative composition (or repetition in the latter case). This is enough to assure the right behavior since `expire(t)` is a boolean expression/statement that is blocked until (the value of) `e` time slices later. Therefore, the right translation of $\Delta e \ [] \ p$ is:

```
set(t,e);
if
 :: expire(t)
 :: p
fi
```

and similarly for `do :: od` statement.

Note that delaying zero time units in χ_σ is equivalent to ε, while in PROMELA it is equivalent to `skip`, i.e. it does an action. This causes similar problems as the translation of ε to `skip` would. For example, the process $\Delta 2 \ ; p \ [] \ \Delta 2 \ ; q$

waits two time units and then behaves as $p \parallel q$. If we translate this process as

```
set(t1,2); set(t2,2);
if
 :: expire(t1); p
 :: expire(t2); q
fi ,
```

then after two clock ticks the values of `t1` and `t2` become 0, the statements `expire(t1)` and `expire(t2)` become executable, and choice is made regardless of the executability of `p` and `q`. Therefore, we can only translate χ_σ specifications in which all delay statements are immediately followed by the *skip* statement.

Remark 2 *We have said already that send and receive processes can implicitly delay and that we assume that the maximal progress operator is only on top of the specification. However, when send or receive should be urgent, i.e. not delayable, we need to use* $\pi(m!e)$ *instead of just* $m!e$ *(same for receive). We can achieve the same in* PROMELA *with:*

```
if
 :: m!e
 :: atomic { timeout; false }
fi .
```

This statement says that the send is available but the passage of time leads to an immediate deadlock.

4.1.6 Guards

Statements of type $b{:}{\rightarrow}p$, in general cannot be just translated as `b -> p`. This is because in PROMELA operator `->` is equivalent to the sequential operator and the boolean expression `b` is also a statement. This means, 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_σ which looks for both b to be *true* and for p to be executable before taking the step. For example, in χ_σ, the process $(true :\rightarrow \delta \parallel true :\rightarrow skip)$ will execute *skip* because δ is never executable. In PROMELA however, process

```
if
 :: true -> false
 :: true -> skip
fi ,
```

since it does not look 'behind' guards, can pick the first `true`, execute it and deadlock afterwards. Thus, the PROMELA statement `b -> p` actually corresponds to the process $(b :\rightarrow \mathit{skip})\ ;p$ in χ_σ.

In the special case when p is an atomic process it is always possible to translate process $b :\rightarrow p$. A guarded *skip* is translated to a PROMELA expression/statement `b`. Guarded delays $(b :\rightarrow \Delta e)$ are simply translated as `b && expire(t)` provided that `t` is previously set to the value of `e`. A guarded assignment, $b :\rightarrow x := e$, we translate as `d_step{b; x = e}`. With the `d_step` operator we force the statement to be executed as one action, like in χ_σ. 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.

In order to translate guarded send/receive actions we must apply a different trick because these actions can block. 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 type integer. 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 we want to avoid the communication between a guarded send and a guarded receive to happen when 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. 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.

Remark 3 *There is another possibility to translate guarded send and receive statements on unbuffered channels (see page 398 in [15]). Instead of adding a boolean argument to a channel declaration we could declare an array of channels of size 3 and then translate* $b{:}\rightarrow m!e$ *to* `m[b]!e` *and* $B{:}\rightarrow m?x$ *to* `m[2-B]?x`. *Since* `b`, `B` $\in \{$`0`, `1`$\}$ *we force all the communication to happen on channels with index* `1`. *As an array index, expression* `2-B` *evaluates automatically so there is no need for the* `eval` *function.*

Like in the case of the scope operator, the restriction to have only guarded atomic processes is not so serious since most processes have equivalents in that form. Transformations that simplify guards in the context of other guards, alternative and sequential composition, and repetition are shown in Table 2 (how guarded scopes are simplified we have shown in the paragraph on scoping). Note that the solution for a guarded repetition introduces ε, but if p^* is followed by the sequential operator the ε is redundant.

In general, $b :\rightarrow (p \parallel q)$ is not equivalent to $(b :\rightarrow p) \parallel (b :\rightarrow q)$. This is because when the value of b is *true*, after executing an action (for example from p)

Table 2
Simplification of guards

$b_1 :\rightarrow b_2 :\rightarrow p$	$(b_1 \wedge b_2) :\rightarrow p$
$b :\rightarrow (p \, [\!] \, q)$	$(b :\rightarrow p) \, [\!] \, (b :\rightarrow q)$
$b :\rightarrow (p; q)$	$(b :\rightarrow p) \, ; q$
$b :\rightarrow p^*$	$(b :\rightarrow p) \, ; p^* \, [\!] \, b :\rightarrow \varepsilon$

process $b{:}\!\rightarrow (p \,\|\, q)$ proceeds as $p' \,\|\, q$ and process $(b{:}\!\rightarrow p) \,\|\, (b{:}\!\rightarrow q)$ as $p' \,\|\, (b{:}\!\rightarrow q)$ and the action might have changed the value of b to $false$. Only when p and q do not change the value of b, e.g. when they do not contain atomic processes that influence variables present in b, we can distribute the guard over the parallel operator.

4.2 Translation scheme

In this section we present the subset of χ_σ models that can be translated to Promela code and present the translation scheme. The subset is generated by the following grammar:

$$A ::= skip \mid x := e \mid m!e \mid m?x \mid \pi(m!e) \mid \pi(m?x) \mid \Delta e \, ; skip$$

$$BP ::= \delta \mid b :\rightarrow BP_1 \mid BP_1 \, ; BP_2 \mid BP_1 \, [\!] \, BP_2 \mid BP_1^* \, ; BP_2 \mid [\![S \mid BP_1]\!]$$

$$P ::= \pi(\partial_A([\![S \mid BP_1 \,\|\, \ldots \,\|\, BP_n]\!])), \; n \geq 1$$

Here S is a state, of the same form as in the explanation of the scope operator of χ_σ but with the restricted number of data types that variables can be declared of. The symbol e represents any expression and the symbol b represents a boolean expression.

The grammar above allows nested scopes and arbitrary guarded processes. These statements do not have direct translations to Promela code, but, to eliminate them, we can perform the transformations from Tables 2 and 1. In this (preprocessing) step we can also introduce the new symbol $\star$ that abbreviates the combination $^*;$. The reduced processes are generated by the following grammar (A stays the same):

$$BP ::= \delta \mid b :\rightarrow A \mid BP_1 \, ; BP_2 \mid BP_1 \, [\!] \, BP_2 \mid BP_1 \star BP_2$$

$$P ::= \pi(\partial_A([\![S \mid [\![S_1 \mid BP_1]\!] \,\|\, \ldots \,\|\, [\![S_n \mid BP_n]\!]]\!])), \; n \geq 1$$

We now present the translation scheme:

$\delta \quad \longmapsto \quad$ `false`

$$b :\to A \quad \longmapsto \quad \begin{cases} \texttt{d\_step\{b; x := e\}} & \text{if } A \equiv x := e \\ \texttt{b} & \text{if } A \equiv \mathit{skip} \\ \texttt{m!e,b} & \text{if } A \equiv m!e \\ \texttt{m?x,eval(2 - b)} & \text{if } A \equiv m?x \\ \begin{array}{l} \texttt{set(t,e);} \\ \texttt{b \&\& expire(t)} \end{array} & \text{if } A \equiv \Delta e \,; \mathit{skip} \\ \begin{array}{l} \texttt{if} \\ \texttt{ :: m!e,b} \\ \texttt{ :: atomic \{timeout; false\}} \\ \texttt{fi} \end{array} & \text{if } A \equiv \pi(m!e) \\ \begin{array}{l} \texttt{if} \\ \texttt{ :: m?x,eval(2 - b)} \\ \texttt{ :: atomic \{timeout; false\}} \\ \texttt{fi} \end{array} & \text{if } A \equiv \pi(m?x) \end{cases}$$

$BP_1 \,;\, BP_2 \quad \longmapsto$

```
BP1; BP2
```

$BP_1 \,[\!]\, BP_2 \quad \longmapsto$

```
if
 :: BP1
 :: BP2
fi
```

$BP_1 \star BP_2 \quad \longmapsto$

```
do
 :: BP1
 :: BP2; break
od
```

$P \quad \longmapsto$

```
S;
active proctype P1() {
   S_1
   BP_1
}
 ...
active proctype Pn() {
   S_n
   BP_n
}
```

$$S \text{ (and } S_i) \quad \longmapsto \quad \begin{cases} & \text{if } S \equiv \lambda_s \\ \begin{array}{l}\texttt{type x [= val];} \\ \texttt{S'}\end{array} & \text{if } S \equiv (x : type[\mapsto val]) : S' \\ \begin{array}{l}\texttt{chan m[0] of \{type,int\};} \\ \texttt{S'}\end{array} & \text{if } S \equiv ({\sim}m) : S' \\ \begin{array}{l}\texttt{TUPLE\_TYPE1\_TYPE2 lst;} \\ \texttt{S'}\end{array} & \text{if } S \equiv tuple[type_1, type_2] : S' \\ \begin{array}{l}\texttt{LIST\_TYPE lst;} \\ \texttt{S'}\end{array} & \text{if } S \equiv list[type] \mapsto [\,] : S' \end{cases}$$

Note that, to correctly translate delays, a postprocessing step that declares and renames timers and that moves `set` functions to outside of `if :: fi` and `do :: od` statements, should be performed.

4.3 Manufacturing line in PROMELA

To illustrate our techniques and translation scheme we present the PROMELA translation of the manufacturing line model presented in Section 2.2.

```
#include "dtime.h"
#include "list.h"

chan gd = [0] of {bool};
chan dm1 = [0] of {bool};
chan dm2 = [0] of {bool};
chan dr = [0] of {bool};
chan cm1 = [0] of {bool};
chan cm2 = [0] of {bool};
chan me = [0] of {bool};
chan mb1 = [0] of {bool,int};
chan mb2 = [0] of {bool,int};
chan bm1 = [0] of {bool,int};
chan bm2 = [0] of {bool,int};
chan mm1 = [0] of {bool};
chan mm2 = [0] of {bool};

active proctype G() {
    timer t;
    bool x = 0;

    do :: delay(t,7); gd!x od;
}

active proctype D() {
    timer t;
    bool x;

    do
     :: gd?x; set(t,5);
        if
         :: dm1!x
         :: dm2!x
         :: expire(t); dr!x
        fi
    od
}

active proctype R() {
    bool x;

    do :: dr?x od
}

active proctype MRW1() {
    bool x;
    timer t;

    do
     :: if
         :: dm1?x
         :: mm1?x; x = 1
        fi;
        delay(t,4);
        mb1!x,1
    od
}
```

```
active proctype MRW2() {
    bool x;
    timer t;

    do
     :: if
         :: dm2?x
         :: mm2?x; x = 1
        fi;
        delay(t,4);
        mb2!x,1
    od
}

active proctype B1() {
    bool x;
    LISTBOOL bf;

    do
     :: mb1?x,eval(length(bf) < 5);
        add(x,bf)
     :: bm1!hd(bf),(length(bf) > 0);
        tail(bf)
    od
}

active proctype B2() {
    bool x;
    LISTBOOL bf;

    do
     :: mb2?x,eval(length(bf) < 5);
        add(x,bf)
     :: bm2!hd(bf),(length(bf) > 0);
        tail(bf)
    od
}

active proctype M1() {
    bool x;
    timer t;

    do
     :: bm1?x,1;
        delay(t,4);
        if
         :: x -> cm1!x
         :: !x -> mm1!x
        fi
    od
}

active proctype M2() {
    bool x;
    timer t;

    do
     :: bm2?x,1;
        delay(t,4);
        if
         :: x -> cm2!x
         :: !x -> mm2!x
        fi
    od
}

active proctype MA() {
    bool x,y;
    timer t;

    do
     :: if
         :: cm1?x; cm2?y
         :: cm2?y; cm1?x
        fi;
        delay(t,4);
        me!(x && y)
    od
}

active proctype E() {
    bool x;

    do :: me?x od
} .
```

Let us now verify that products that are only assembled once do not leave the system. First note that this is equivalent to saying that, in all states of the system, the variable `x` from the process `E` has the value `1` (if also it was initially `1`). In the linear temporal logic built in SPIN this is expressed as `[] (x == 1)`. Since this logic allows reasoning only about global variables, we have to move `x` to the global scope (and initialize it to `1`). SPIN verified this property almost instantly.

5 Conclusion

In this paper we investigated the possibility to translate χ specifications to PROMELA. We found that most χ models have equivalents in PROMELA, but also that sometimes, what seems to be an obvious translation, can have very different behavior.

Some constructions that we cannot translate, like e.g. ε and $p^* \mathbin{[\!]} q$, are very rare in models of industrial systems. On the other hand, terms like $(p \parallel q)^*$ are quite common in such models. Nested parallelism is a major problem that we cannot (yet) deal with in a satisfactory way. This is a subject for further investigations.

We were able to syntactically define a translatable subset and we presented a translation scheme. We also defined the phases of a translation process. Based on this we are developing an automatic translator from χ to Promela.

Together, the simulator of χ and a tool that translates χ models into Promela will constitute an effective environment in which in which performance analysis and functional analysis of industrial systems are combined.

References

[1] 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, 2002.

[2] S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *Proceedings of CAV2001*, LNCS 2102, pages 250–254, 2001.

[3] E. Bortnik, N. Trčka, A.J. Wijs, S.P. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and Uppaal. Technical Report CS-04/23, Eindhoven University of Technology, 2004. To appear in *Journal Of Logic and Algebraic Programming.*

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

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

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

[7] E. J. J. van Campen. *Design of a Multi-Process Multi-Product Wafer Fab.* PhD thesis, Eindhoven University of Technology, 2000.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[9] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *Proceedings 8th of CAV'96*, LNCS 1102, pages 437–440, 1996.

[10] J.J.H. Fey. *Design of a Fruit Juice Blending and Packaging Plant.* PhD thesis, Eindhoven University of Technology, 2000.

[11] R. Gerth. Concise Promela reference. Obtainable from: `http://spinroot.com/spin/Man/Quick.html`.

[12] J. A. Govaarts. Efficiency in a lean assembly line: a case study at NedCar born. Master Thesis, October, 1997.

[13] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Trans. Softw. Eng.*, 27(8):749–765, 2001.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

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

[16] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

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

[18] O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *13th IEEE Symposium on Logic in Computer Science*, Indianapolis, Indiana, USA, 1998.

[19] 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.

[20] B. Luttik and N. Trčka. Stuttering congruence for χ. Technical Report CS-05/13, Eindhoven University of Technology, 2005.

[21] R.R.H. Schiffelers, D.A. van Beek, K.L. Man, M.A. Reniers, and J.E. Rooda. Syntax and consistent equation semantics of hybrid Chi. Technical Report CS-04/37, Eindhoven University of Technology, 2004.

[22] C. Stirling. *Modal and Temporal Properties of Processes.* Springer, 2001.