Regular Model Checking for LTL(MSO)*

Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, Mayank Saksena

Dept. of Information Technology, P.O. Box 337, S-751 05 Uppsala, Sweden
e-mail: parosh@it.uu.se, bengt@it.uu.se, marcus@nilsson.tv, juldor@it.uu.se, mayanksaksena@hotmail.com

Abstract. Regular model checking is a form of symbolic model checking for parameterized and infinite-state systems whose states can be represented as words of arbitrary length over a finite alphabet, in which regular sets of words are used to represent sets of states. We present *LTL*(*MSO*), a combination of the logics *MSO* and *LTL* as a natural logic for expressing temporal properties to be verified in regular model checking. In other words, *LTL*(*MSO*) is a natural specification language for both the system and the property under consideration. *LTL*(*MSO*) is a two-dimensional modal logic, where *MSO* is used for specifying properties of system states and transitions, and *LTL* is used for specifying temporal properties. In addition, the first-order quantification in *MSO* can be used to express properties parameterized on a position or process. We give a technique for model checking *LTL*(*MSO*), which is adapted from the automata-theoretic approach: a formula is translated to a *Büchi regular transition system* with a regular set of accepting states, and regular model checking techniques are used to search for models. We have implemented the technique, and show its application to a number of parameterized algorithms from the literature.

Key words: Formal verification – Model checking – Parameterized systems – Communication protocols – Monadic logic – Temporal logic – Regular model checking

* This is a manuscript which is similar to the paper with same title and authors, and appearing in Int. J. on Software Tools for Technology Transfer 14(2): 223-241. Springer, 2012. Work supported in part by the Swedish Research Council and the UPMARC Center of Excellence

1 Introduction

Regular model checking is a framework for algorithmic symbolic verification of parameterized and infinite-state systems [7, 23, 40, 8]. It considers systems whose states can be represented as finite words of arbitrary length over a finite alphabet, including array or ring-formed parameterized systems with an arbitrary number of finite-state processes, and systems that operate on queues, stacks, integers, and other linear unbounded data structures. In a system description, the set of initial states is represented as a regular set of strings, and the transition relation is given as a finite regular length-preserving transducer. Previous work on regular model checking [22, 8, 2] has developed methods for computing the set of reachable states of a system description, as well as the set of reachable loops, obtained from the transitive closure of the transition relation. In general, this problem is undecidable, but decidability results for certain classes have been obtained [22].

The techniques for computing reachable states and reachable loops can in principle be used to verify both safety and liveness properties of parameterized system descriptions, but do not provide a convenient approach for checking arbitrary temporal logic properties of parameterized and infinite-state systems. Significant ingenuity is required in order to manually transform the verification of a temporal property of a parameterized system into a property of reachable states and reachable loops, in particular if the verification uses fairness properties that are parameterized on system components [8, 32]. It would be desirable to have a framework, analogous to the automata-theoretic approach in finite-state model checking [38], where the property of verifying a temporal property is automatically transformed into a problem of checking emptiness for a Büchi automaton.

In this paper, we address this problem by presenting an extension of the automata-theoretic approach [38] to the setting of regular model checking. We present a logic for expressing system models and temporal properties, which is a combination of the logics MSO over finite words and LTL. We use MSO for specifying sets of states and transition relations and LTL for specifying temporal constraints. The result is a two-dimensional modal logic, where MSO is used in the "space" (system state) dimension and LTL is used in the "time" dimension. Models of the logic are infinite sequences of (constant-length) words, representing computations of the specified system. We can then specify a verification problem as the conjunction of a system specification and a negation of the property to be verified, and reduce verification to checking whether this conjunction is satisfiable.

Following the automata-theoretic approach, we present an automated translation from a formula φ in $LTL(MSO)$ to a *Büchi regular transition system (BRTS)*, consisting of a regular set I of initial states, a regular length-preserving transducer T, and a regular length-preserving transducer F, representing the set of final transitions. Accepting runs of the BRTS are infinite sequences of words, where the first word is in I, consecutive words satisfy T, and infinitely many pairs of consecutive words satisfy F. We prove that φ is satisfiable if and only if the BRTS has an accepting run. Since T is length-preserving, the existence of an accepting run can be checked by searching for a reachable loop which contains a transition that satisfies F. Note that we allow F to denote a set of transitions rather than only a set of states, as in, e.g., [38]: this difference only a slight technical convenience and not essential.

A nice feature of our combination of MSO with LTL is that we get the power to express temporal properties parameterized over positions for free: MSO offers variables to represent positions and quantify over them, which can be interleaved with temporal operators. As a concrete example, for a parameterized mutual exclusion algorithm, a typical property one would want to express is the following.

> If all processes satisfy a weak fairness requirement, then each process that is interested in entering its critical section will eventually do so.

If the number of processes is fixed, the terms like "each process" can be replaced by explicit conjunctions to obtain a standard model checking problem in propositional temporal logic. However, for parameterized systems the number of processes is arbitrary. Fortunately, we can express this property directly in our logic, by a formula like

$$\begin{array}{l}\forall i : \Box\Diamond[\text{blocked}(i) \vee \text{progressing}(i)] \\ \longrightarrow \\ \qquad \forall i : \Box\,[\text{trying}(i) \rightarrow \Diamond\text{critical}(i)]\end{array}$$

where i ranges over positions in the state, and each position represents a process. In this formula, we apply LTL operators ($\Box$ and $\Diamond$) to formulas with the MSO variable i, and later use MSO quantification over i to express parameterized properties. In our logic $LTL(MSO)$, temporal operators can be applied to formulas with at most one free first-order variable and no free second-order variables. This restriction allows to express parameterized temporal properties (e.g., fairness constraints) of individual processes in a parameterized system, as well as temporal properties of pairs of adjacent processes (in positions i and $i+1$ using one free variable i). The restriction is necessary for making our translation into automata possible, explained in Section 7.

A further nice property of adapting the automata-theoretic approach is that our transformation results in a uniform problem of checking for accepting runs, for which we can develop techniques that are more uniform than those presented in previous work [22, 8, 2]. We have extended our tool for regular model checking [3] to check whether BRTS have accepting runs. This is done in two steps. First, the set of reachable states are computed as $Inv = I \circ T^*$. Secondly, loops are found by identifying identical pairs in $(F \cap T \cap (Inv \times Inv)) \circ T^*$. This computation is more uniform and more efficient than the approach to verification of temporal logic properties outlined in [8], which builds on computation of the transitive closure T^+ of the transition relation. We have verified safety properties with the tool for many of the examples in our previous work, as well as liveness properties for some of the examples.

As special cases, when the formula contains no temporal operators, our method specializes into a decision procedure for MSO similar to that of MONA [21], and when the formula contains no quantifiers our method specializes to ordinary (i.e. finite-state) LTL model checking.

The remainder of the paper is structured as follows. In the next two sections, we present the logic $LTL(MSO)$. Section 4 illustrates how it can be used to model and specify parameterized algorithms. The model checking technique, including the translation to BRTS is presented and proven correct in Section 7. Verification is discussed in Section 8.

Related Work Kesten et al. [23] and Pnueli and Shahar [32] use the logic FS1S which has the expressive power of regular expressions, to specify sets of states of parameterized systems, just as we do with our logic. The difference is essentially that we have a higher level approach, considering all of (future) LTL [29], and automatic translation. However, unlike us, Kesten et al. [23] also consider a logic for trees. Inspired by our work [1], Fisman et al. [17] use essentially the logic $LTL(MSO)$ to specify and verify faul-tolerant parameterized protocols, using techniques similar to those presented in this paper.

Bouajjani, Legay, and Wolper [9] independently (from us) characterize *global* and *local-oriented* properties in the framework of (ω-) regular model checking, and work out how to analyze such properties. They also consider

ω-regular systems, i.e., systems where configurations are infinite words. However, they do not provide an automatic translation from a system and property description into a verification problem, as we do here.

Esparza et al. [15] present techniques for model checking pushdown systems for specifications in LTL, where atomic predicates are arbirary regular sets of stack contents. Compared to this logic, *LTL*(*MSO*) is a tighter integration between LTL and MSO/regular sets, since (possibly quantified) LTL formulas can occur inside MSO formulas.

Our logic *LTL*(*MSO*) applied to words is related to existential monadic second-order logic ($EMSO$) on grids to define regular picture languages accepted by *tiling systems* (see e.g. [19]). Indeed, transducers over words can be considered as tiling systems where each transition represents a *tile*. Thus, it is expected that our logic *LTL*(*MSO*) has similar expressive power as $EMSO$ on grids. However, the two logics come from different motivations. While $EMSO$ on grids is used to reason about *pictures*, our logic is used to reason about *parameterized structures over time*. When applied to the word structure, the two logics have similar expressive power.

In addition to the work on regular model checking, cited earlier, there is a large body of research on the problem of model checking parameterized systems of *identical* processes, in which there is no ordering between processes, and hence the system state can be represented as a multiset of process states (e.g., [5,10,13,12,18]). This problem is substantially simpler, since ordering between processes need not be considered.

Emerson and Namjoshi [14] give a technique for verifying a restricted class of parameterized token-passing algorithms by reducing an arbitrary ring to a small fixed-size ring under certain conditions. These restrictions are substantially stronger than in our framework. Sistla [34] uses Büchi automata over two dimensional languages (reminding of transducers) to specify network invariants when verifying systems by induction over their linear process structure. It is unclear what class of systems can be handled automatically by this technique.

The problem of checking liveness properties of array-shaped parameterized systems was considered by Pnueli and Shahar [32], who presented a technique for computing the transitive closure of a restricted class of transition relations. They also first manually employ abstractions to make the implementation terminate.

Pnueli, Xu, and Zuck [33] present an interesting use of specialized abstractions in order to prove absence of starvation properties for Szymanski's algorithm and the Bakery algorithm. The abstractions keep track of the number of processes with certain properties, and generate a finite-state system, which can be model-checked. The presented abstraction is specialized to prove non-starvation, and loses much information so that, e.g., safety properties can no longer be checked. Fang et al. [16] present techniques for finding premises in proof rules for symbolic verification of parameterized protocols by generalizing information that is obtained when verifying the protocol for a small number of nodes. In [16] this scheme is employed to prove progress properties, and in [31] to prove invariants.

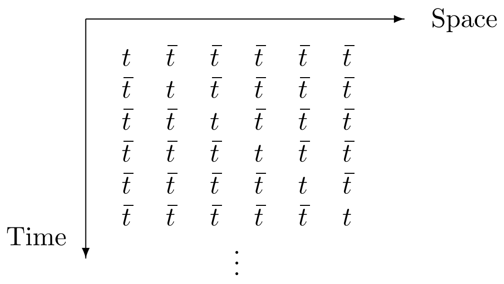
2 Introduction to *LTL*(*MSO*)

We introduce the logic *LTL*(*MSO*) [1], intended for reasoning about infinite sequences of words of arbitrary length. Such sequences are useful to model executions of parameterized systems, where there are an arbitrary number of processes organized in a linear network. Each word in an execution models a system configuration, where each position in the word contains the local state of each process.

We follow the approach of *the Temporal Logic of Actions* by Lamport [25], where both the protocol and the properties are specified by formulas in a single logic. Correctness of the protocol means that the formula specifying the protocol implies the formula specifying the property. We show how to specify protocols and properties using this logic and how to set up verification problems. Formulas in this logic can then be translated into BRTS, introduced in Section 8, which can be used to find models of the original formula.

As a running example, we use a *token passing* protocol. It consists of an arbitrary number of processes organized in a linear array and numbered from 0 to $n-1$. The processes are ordered from left to right such that process 0 is the leftmost process and process $n-1$ is the rightmost process. Initially, the leftmost process has the token. In each step, a process can pass the token to its right neighbor. We model each configuration as a word w over the alphabet $\{\bar{t}, t\}$ where the local configuration of process i is modeled by the symbol $w(i)$, i.e., the symbol at position i of the word. The symbol $\bar{t}$ denotes a process that does not have the token, while the symbol t denotes a process that has the token.

In a system where configurations are modeled as words, an execution is an infinite sequence of words. All words in an execution have the same arbitrary length. Thus, we are working with two different dimensions. One dimension refers to the positions of the word, called the *space dimension*, and the other dimension refers to the points in time, called the *time dimension*. An execution of the token passing protocol is shown below; it can be seen as a *matrix* in which each element is indexed by a *timepoint* and a *position*, where the position refers to a process.



Formulas in *LTL*(*MSO*) will be interpreted over such matrices. The logic consists of constructs for handling both the space and the time dimensions. Below, we introduce the constructs of *LTL*(*MSO*) and illustrate with the token passing protocol.

Configuration Variables and Positions The atomic formulas are of the form $x[i]$ where x is a *configuration variable* and i is a *position variable.* The configuration variables model the global state of the protocol we are modeling. Each configuration variable contains a boolean variable for each position in the word, and is therefore essentially a boolean array (bit vector). The formula $x[i]$ denotes the boolean value of x at position i, at the timepoint at which the formula is interpreted. In the case of the token passing example, we use a configuration variable t such that $t[i]$ is true if and only if process i has the token.

MSO To specify configurations, i.e., the space dimension, we use *Monadic Second-Order Logic* (*MSO*) over words [37,21], a logic that can express regular sets of words. It contains first-order position variables $i, j, \cdots$ denoting positions, and second-order position variables $I, J, \cdots$ denoting sets of positions. The atomic formulas of *MSO* are of the form $i = j + 1$ (successor), $i \in I$, and $I \subseteq J$, where i, j are position variables and I, J are sets of position variables. A configuration variable x can be seen as a special case of a second-order variable, where $x[i]$ means $i \in x$, except that a configuration variable may change over time. Configuration variables are used for the purpose of modeling configurations, and always occur free in formulas. First-order quantification over positions and second-order quantification over sets of positions are allowed. For example, the formula

$$\forall i : \, x[i]$$

can be used to specify that the configuration variable x is true at all positions. Using a combination of successor and quantification, we can express ordering, e.g., $\neg\exists j : i = j + 1$ can be used to express that $i = 0$. We can also express constant distances between positions of the form $i = j + c$ for any constant c, as well as the ordering $<$, using second-order quantification. We will use formulas with position variables like $x[i+1]$ to mean $\exists j : j = i + 1 \wedge x[j]$.

In the token passing protocol, we can specify the initial condition that the first process has the token by the formula

$$\forall i : \, t[i] \leftrightarrow i = 0$$

Primed Variables To specify transition relations, we need a relation between the current and the next timepoint. We use *primed* configuration variables for this, where $x'[i]$ is the value of x at position i at the next timepoint. In the token passing protocol, the transition relation where a process passes the token to its right neighbor is specified by

$$\exists i : \left[\begin{array}{c} t[i] \wedge \neg t'[i] \wedge \neg t[i+1] \wedge t'[i+1] \\ \wedge \ \forall j \notin \{i, i+1\} : t'[j] = t[j] \end{array} \right]$$

Temporal Operators While *MSO* is used to reason about the space dimension, *linear temporal logic* (*LTL*) [29,30,27] is used to reason about the time dimension. The linear temporal logic adds the connectives $\Box$ *(always in the future)*, $\Diamond$ *(eventually)* and $\mathcal{W}$ (*weak until).* In the token passing protocol, the following formula can be used to express that eventually the rightmost process has the token.

$$\Diamond \exists i : \, t[i] \wedge i = \$$$

where $i = \$$ means that i is the rightmost process (which can be expressed in *MSO*). Similarly, we can use the following formula to denote that there is always at least one token in the system

$$\Box \exists i : t[i]$$

Combining the two logics *LTL* and *MSO*, we obtain the logic *LTL*(*MSO*) by allowing the position quantifiers and the temporal connectives to interleave. For example, we can express that at some point in time there is a process which from then on always has the token:

$$\Diamond \exists i : \Box \, t[i]$$

Given a formula φ representing a transition relation, we can use the formula $\Box\varphi$ to specify that all pairs of consecutive (in time) configurations will satisfy the constraints of the transition relation. The token passing protocol can thus be specified by conjoining the specification of the set of initial configurations and the transition relation:

$$\begin{array}{l} \forall i : t[i] \leftrightarrow i = 0 \\ \wedge \ \Box \exists i : \left[\begin{array}{c} t[i] \wedge \neg t'[i] \wedge \neg t[i+1] \wedge t'[i+1] \\ \wedge \ \forall j \notin \{i, i+1\} : t'[j] = t[j] \end{array} \right] \end{array}$$

Interleaving of position quantifiers and temporal operators will be restricted so that there can be at most one free position quantifier inside temporal operators (otherwise they cannot be translated — see Section 7). For example,

$$\forall i : \Diamond \forall j : x[i] = y[j]$$

is allowed but not

$$\forall i : \forall j : \Diamond \, x[i] = y[j]$$

3 *LTL*(*MSO*)

We give the syntax and semantics of *LTL*(*MSO*).

Syntax We assume a set $\mathcal{V}$ of *configuration variables*, denoted by $x, y, z, \cdots$, a set of *first-order position variables*, denoted by $i, j, k, \cdots$, and a set of *second-order position variables*, denoted by $I, J, K, \cdots$. The set of *LTL*(*MSO*) formulas is inductively defined as follows.

$i \in I \mid I \subseteq J \mid i = j + 1$	Atomic MSO formulas
$x[i], x'[i]$	Configuration Variables
$true \mid false$	Boolean constants
$\varphi \vee \psi \mid \varphi \wedge \psi \mid \neg\varphi$	Propositional connectives
$\forall i : \varphi \mid \forall I : \varphi \mid \exists i : \varphi \mid \exists I : \varphi$	MSO Quantification
$\Box\,\varphi \mid \Diamond\,\varphi \mid \varphi\,\mathcal{W}\,\psi$	Temporal operators

We impose the restriction that in each subformula of the form $\Box\,\varphi$ or $\Diamond\,\varphi$ or $\varphi\,\mathcal{W}\,\psi$ there is at most one free first-order position variable and no free second-order position variable. Let us call a formula with this restriction a *restricted formula*. The restriction is required for the translation of a formula into Büchi Normal Form, given later in Section 7. It is well-known that the temporal operators $\mathcal{U}$ (*until*) and $\mathcal{R}$ (*release*) can be expressed using the operators above [27]. Hence we include all of (future) *LTL* [29]. We will use the following abbreviations.

$$\begin{aligned}
\varphi \to \psi &\triangleq \neg\varphi \vee \psi \\
\varphi \leftrightarrow \psi &\triangleq (\varphi \to \psi) \wedge (\psi \to \varphi) \\
\varphi\,\mathcal{U}\,\psi &\triangleq (\varphi\,\mathcal{W}\,\psi) \wedge \Diamond\psi \\
\varphi\,\mathcal{R}\,\psi &\triangleq \neg(\neg\varphi\,\mathcal{U}\,\neg\psi) \\
\varphi(f(i)) &\triangleq \exists j : j = f(i) \wedge \varphi(j) \\
&\qquad \text{where } f(i) \text{ is an expression over } i \\
i < j &\triangleq \forall K : \left[\begin{array}{c} \left[\begin{array}{c} i+1 \in K \\ \wedge\ \forall k : [k \in K \to k+1 \in K] \end{array}\right] \\ \to j \in K \end{array}\right] \\
i = 0 &\triangleq \neg\exists j : i = j + 1 \\
i = \$ &\triangleq \neg\exists j : j = i + 1
\end{aligned}$$

Semantics *LTL*(*MSO*) formulas are interpreted over *matrices* M over $2^{\mathcal{V}}$ of dimension $\infty \times n$, for some $n > 0$, given as a parameter. We call the vertical (first) dimension *time*, and the horizontal (second) dimension *space.*

Let $\mathbf{N}$ be the set of natural numbers, and $\mathbf{Z}_n = \{0, \ldots, n-1\}$. A matrix is a function $M : (\mathbf{N} \times \mathbf{Z}_n) \mapsto 2^{\mathcal{V}}$. The element $M(t,i) \subseteq \mathcal{V}$ for $t \in \mathbf{N}$ and $i \in \mathbf{Z}_n$ represents the system configuration at time t of position (or subsystem) i, which assigns truth values to the configuration variables $\mathcal{V}$ — the variables assigned true are included in $M(t,i)$, those assigned false are not. We denote by $M(t)$ the row $M(t,0)\ M(t,1) \cdots M(t,n-1)$. The row $M(t)$ represents the system configuration at time t.

In general, a formula φ depends on its free first- and second-order variables and a timepoint, and the configuration variables of M. A *valuation Val* is a mapping from first-order variables to $\mathbf{Z}_n$ and second-order variables to $2^{\mathbf{Z}_n}$. We define satisfaction of formulas, $(M, Val, t) \models \varphi$, with respect to a matrix M, a valuation *Val*, and a timepoint t as shown in Figure 1. For a closed formula φ we denote by $M \models \varphi$ that $(M, \emptyset, 0) \models \varphi$.

$false$	never
$true$	always
$i \in I$	if $Val(i) \in Val(I)$
$I \subseteq J$	if $Val(I) \subseteq Val(J)$
$i = j + 1$	if $Val(i) = Val(j) + 1$
$x[i]$	if $x \in M(t, Val(i))$
$x'[i]$	if $x \in M(t+1, Val(i))$
$\varphi \vee \psi$	if $(M, Val, t) \models \varphi$ or $(M, Val, t) \models \psi$
$\varphi \wedge \psi$	if $(M, Val, t) \models \varphi$ and $(M, Val, t) \models \psi$
$\neg\varphi$	if $(M, Val, t) \not\models \varphi$
$\forall i : \varphi$	if for all $m \in \mathbf{Z}_n$ we have $(M, Val[i \mapsto m], t) \models \varphi$
$\forall I : \varphi$	if for all $S \subseteq \mathbf{Z}_n$ we have $(M, Val[I \mapsto S], t) \models \varphi$
$\exists i : \varphi$	if there exists $m \in \mathbf{Z}_n$ such that $(M, Val[i \mapsto m], t) \models \varphi$
$\exists I : \varphi$	if there exists $S \subseteq \mathbf{Z}_n$ such that $(M, Val[I \mapsto S], t) \models \varphi$
$\Box\,\varphi$	if for all $t' \geq t$ we have $(M, Val, t') \models \varphi$
$\Diamond\,\varphi$	if there exists $t' \geq t$ such that $(M, Val, t') \models \varphi$
$\varphi\,\mathcal{W}\,\psi$	if $(M, Val, t) \models \Box\,\varphi$ or there exists $t' \geq t$ such that $(M, Val, t') \models \psi$ and for all t'' with $t \leq t'' < t'$ we have $(M, Val, t'') \models \varphi$

Fig. 1. Semantics of *LTL*(*MSO*). For each row, the expression in the right column defines when $(M, Val, t) \models \psi$, where ψ is the formula in the left column. The valuation $Val[i \mapsto m]$ acts as *Val* except that it maps i to m. The valuation $Val[I \mapsto S]$ is defined analogously.

4 Modeling in *LTL*(*MSO*)

In this section, we discuss how to model systems and set up verifications problem in *LTL*(*MSO*).

4.1 Specifying Systems

A *state formula* is a formula without temporal operators and primed variables, used for specifying constraints on only one configuration. An *action formula* is a formula over unprimed and primed configuration variables without temporal operators, used for specifying constraints on two consecutive configurations. For an action formula φ_T, we can use $\Box\,\varphi_T$ to specify that a any two successive configurations satisfy φ_T. Conjoining this with a state formula φ_I specifying the set of initial configurations, we get the formula $\varphi_I \wedge \Box\,\varphi_T$ whose models correspond to executions of the transition system where φ_I specifies the set of initial configurations and φ_T specifies the transition relation.

Extended Syntax for Modeling Apart from the abbreviations already introduced, we will also use the following abbreviations to make our models more readable.

$$\begin{array}{lcl}
\exists x' : \varphi & \triangleq & \exists K : \varphi' \\
 & & \text{where } \varphi' \text{ equals } \varphi \text{ except that all} \\
 & & \text{occurrences of the form } x'[i] \text{ are re-} \\
 & & \text{placed by } i \in K\text{, where } K \text{ is a fresh} \\
 & & \text{second-order position variable} \\
\textbf{Enabled}\ \varphi & \triangleq & \exists\, x'_1, x'_2, \ldots, x'_n : \varphi \\
 & & \text{where } \varphi \text{ is an action formula, and} \\
 & & x_1, x_2, \ldots, x_n \text{ are all configuration} \\
 & & \text{variables occurring in } \varphi \\
x[i](v, v') & \triangleq & x[i] = v \wedge x'[i] = v'
\end{array}$$

The formula **Enabled** φ is used to test if the transition represented by φ can be taken, and the formula $x[i](v, v')$ is used to say that the value of x changes from v to v'. Furthermore, we extend the range of configuration variables to any finite domain (rather than just boolean values) by using a standard encoding of a finite domain into a set of boolean variables. For example, when pc is a configuration variable representing a program counter at each position, we can use $pc[i](5, 6)$ to express that the value of pc at position i changes from 5 to 6.

To model the token passing protocol introduced in Section 2, we use a configuration variable variable t where $t[i]$ is true iff process i has the token. The protocol is modeled by the formulas below. Note that if $i = \$$, the token cannot be passed since there is no position $i + 1$.

$$\begin{array}{lcl}
\textbf{initial} & = & \forall i : t[i] \leftrightarrow i = 0 \\
\textbf{passtoken}(i) & = & \left[\begin{array}{r} t[i] \wedge \neg t'[i] \wedge \neg t[i+1] \wedge t'[i+1] \\ \wedge\ \forall j \notin \{i, i+1\} : t'[j] = t[j] \end{array} \right] \\
\textbf{transition} & = & \exists i : \textbf{passtoken}(i) \\
\textbf{idle} & = & \forall i : t'[i] = t[i] \\
\textbf{sys} & = & \textbf{initial} \wedge \Box\, (\textbf{transition} \vee \textbf{idle})
\end{array}$$

The set of initial configurations, where the first process has the token, is specified by the state formula **initial**. The formula **passtoken**(i) specifies that the token is passed by process i to its neighbor, and the formula **idle** specifies that nothing happens. The formula **idle** is used to model that the system may do things between passing the token, and will be necessary for adequately modeling liveness properties. The transition relation is obtained by conjoining the action formulas **transition** and **idle**, which is combined with **initial** to form the system formula **sys**, representing executions of the system.

A model of the formula **sys** from the token passing example is given below:

$$\begin{array}{ccccc}
t & \bar{t} & \bar{t} & \bar{t} & \bar{t} \\
\bar{t} & t & \bar{t} & \bar{t} & \bar{t} \\
\bar{t} & \bar{t} & t & \bar{t} & \bar{t} \\
\bar{t} & \bar{t} & t & \bar{t} & \bar{t} \\
\bar{t} & \bar{t} & t & \bar{t} & \bar{t} \\
\bar{t} & \bar{t} & \bar{t} & t & \bar{t} \\
\bar{t} & \bar{t} & \bar{t} & \bar{t} & t \\
\bar{t} & \bar{t} & \bar{t} & \bar{t} & t \\
\bar{t} & \bar{t} & \bar{t} & \bar{t} & t \\
 & & \vdots & &
\end{array}$$

4.2 Fairness

To verify liveness properties, we need to add *fairness assumptions*. In this paper, we use *weak fairness*, although the logic can be used to express other kinds of fairness assumptions as well, e.g., strong fairness. Weak fairness is specified on an action formula, and can be defined as

$$\mathit{WF}(\varphi_T) = \Box\Diamond\, (\varphi_T \vee \neg\textbf{Enabled}\ \varphi_T)$$

which states that the action specified by the formula φ_T is either taken infinitely often or disabled infinitely often. When specifying fairness for concurrent systems, it is natural to specify weak fairness *for each process*, stating that each process that may execute will eventually do so. This is an assumtion on the scheduler of the system, assuring that the all processes in a system are scheduled infinitely often. We call this *process fairness*, and express it as:

$$\forall i : \mathit{WF}(\varphi_T(i))$$

where $\varphi_T(i)$ specifies all transitions in which process i is active. In the token passing example, we add process fairness to the transitions specified by **passtoken**(i) using the formula:

$$\varphi_{fair} = \forall i : \mathit{WF}(\textbf{passtoken}(i))$$

Let us expand the definitions to demonstrate the meaning of φ_{fair}. Substituting the definition of **Enabled**, and expanding the definition of $t[i{+}1]$, we obtain the formula

$$\forall i : \Box\Diamond \left[\begin{array}{l} \quad \textbf{passtoken}(i) \\ \vee\ \neg\exists K : \exists j : \\ \qquad \left[\begin{array}{l} \quad j = i + 1 \\ \wedge\ t[i]\ \wedge\ i \notin K\ \wedge\ \neg t[j]\ \wedge\ j \in K \\ \wedge\ \forall k \notin \{i, j\} : k \in K \leftrightarrow t[k] \end{array} \right] \end{array} \right]$$

which after removal of the existential quantifier on K (an interpretation of K will always exist provided the other conditions hold) becomes:

$$\forall i : \Box\Diamond \left[\begin{array}{l} \quad \textbf{passtoken}(i) \\ \vee\ \neg\exists j : j = i + 1\ \wedge\ t[i]\ \wedge\ \neg t[j] \end{array} \right]$$

meaning that for all processes i, it is infinitely often the case that the token is passed *or* the token cannot be passed either because it is the rightmost process (no j exists such that $j = i{+}1$), the process does not have the token ($t[i]$ is false), or the neighboring process already has a token ($t[j]$ is true).

4.3 Specifying and Checking Properties

Let

$$\varphi_I \wedge \Box\varphi_T \wedge \varphi_{fair}$$

be a system specified with fairness assumptions. A *property* is given as a formula φ; for instance, an invariant property is of the form $\Box\varphi_{Inv}$ for a state formula φ_{Inv}. To check whether the system model satisfies the property φ, we check whether the formula

$$\varphi_I \wedge \Box\varphi_T \wedge \varphi_{fair} \wedge \neg\varphi$$

is satisfiable. If φ is a safety property, the fairness assumptions φ_{fair} are not necessary, and can be omitted.

Continuing the token passing example, we can check that there is never more than one token in the system by searching for models of the formula

$$\mathbf{initial} \wedge \Box\,(\mathbf{transition} \vee \mathbf{idle}) \wedge \neg\Box\,\neg\exists i, j\; i \neq j \wedge t[i] \wedge t[j]$$

and whether the rightmost process must eventually get the token searching for models of the formula

$$\begin{array}{l} \quad \mathbf{initial} \\ \wedge\ \Box\,(\mathbf{transition} \vee \mathbf{idle}) \\ \wedge\ \forall i : \Box\Diamond\,(\mathbf{transition}(i) \vee \neg\mathbf{Enabled\ transition}(i)) \\ \wedge\ \neg\Diamond\,\exists i : i = \$ \wedge t[i]\,. \end{array}$$

In the following sections, we discuss how to model parameterized algorithms and algorithms with different kinds of datatypes in our logic.

5 Parameterized Systems

Consider a system parameterized by the number of processes. Typical examples are algorithms designed to work for an arbitrary number of processes. In this case, we want to verify the system regardless of the number of processes.

We assume that the processes are homogeneous, i.e., that all processes have the same set of local states. We use a configuration variable x so that the value of $x[i]$ represents the local state of process i.

Local transitions, where a process can change local state from q to q' independently of other processes, can be expressed as

$$\exists i : x[i](q, q') \wedge \forall j \neq i : x[j] = x'[j]\ .$$

Other transitions need global conditions, for example that all processes at a position with a lower index should be in a particular state, say q_g. We can express this as

$$\exists i : x[i](q, q') \wedge (\forall j < i : x[j] = q_g) \wedge \forall j \neq i : x[j] = x'[j]\ .$$

We can also model transitions representing communication between two processes, e.g.,

$$\exists i : x[i](q, q') \wedge x[i{+}1](r, r') \wedge \forall j \notin \{i, i{+}1\} : x[j] = x'[j]\ .$$

We illustrate this type of representation using a number of examples.

Idle:	$ticket_i := 1 + \max\limits_j ticket_j$
Waiting:	**await** $\forall j \neq i :$
	$(ticket_i < ticket_j \vee ticket_j = 0)$
Critical:	$ticket_i := 0$

Fig. 2. Bakery algorithm

maxplusone(i)	$=$	$(i \neq 0 \longrightarrow q[i-1] \neq \perp)$
		$\wedge \forall j > i : q[j] = \perp$
min(i)	$=$	$q[i] \neq \perp \wedge \forall j < i : q[j] = \perp$
ticket(i)	$=$	$q[i](\perp, W) \wedge$ **maxplusone**(i)
enter(i)	$=$	$q[i](W, C) \wedge$ **min**(i)
exit(i)	$=$	$q[i](C, \perp)$
copy(i)	$=$	$q[i] = q'[i]$
idle	$=$	$\forall i :$ **copy**(i)
a(i)	$=$	(**ticket**(i) $\vee$ **enter**(i) $\vee$ **exit**(i))
		$\wedge \forall j \neq i :$ **copy**(j)
initial	$=$	$\forall i : q[i] = \perp$
sys	$=$	**initial** $\wedge\ \Box(\exists i :$ **a**(i) $\vee$ **idle**$)$

Fig. 3. Bakery algorithm in *LTL*(*MSO*)

5.1 The Bakery Algorithm

In the bakery algorithm for mutual exclusion due to Lamport [24], there are an arbitrary number of processes waiting to get a "ticket" to get into the critical section. Each process that wants to get into the critical section receives a ticket which is the maximum of all the outstanding tickets plus one. When a process has the lowest outstanding ticket, it enters the critical section and drops the ticket when leaving. The algorithm is shown in Fig. 2, where $ticket_i$ is used to denote the ticket value of process i or 0 if it does not have a ticket.

To model the bakery algorithm in *LTL*(*MSO*), we change the perspective: rather than modeling the vector of process states, we let a configuration represent the states of the sequence of ticket numbers, using the configuration variable q. For each i, the value of $q[i]$ is

- $\perp$ if there is no process that has ticket $i + 1$,
- W if some process with ticket $i + 1$ is Waiting, and
- C if some process with ticket $i + 1$ is in Critical.

Note that we do not model tickets with number 0, since this is the ticket number of all "inactive" processes, and that ticket $i + 1$ is modeled by $q[i]$. We implicitly use the invariant that each positive ticket number can be held by at most one process. This invariant can be verified separately, or not be assumed (for example by adding one more value of $q[i]$ representing that several processes have this ticket number).

The initial configuration and transition relation of the bakery algorithm can then be specified by the formulas shown in Fig. 3.

We use the auxiliary formula **maxplusone**(i) to specify that i refers to the position representing next ticket, i.e., the maximum ticket number plus one, and the auxiliary formula $\mathbf{min}(i)$ to specify that i refers to the position representing the ticket that is next in line, i.e., the ticket with the minimum ticket number.

We use one action formula for the transition between states: $\mathbf{ticket}(i)$ specifies the transitions from $\bot$ to W, allowing ticket number $i+1$ to be taken, $\mathbf{enter}(i)$ specifies the transition from W to C, allowing a process with ticket number $i+1$ to proceed to the critical section, and finally $\mathbf{exit}(i)$ specifies the transitions from C to $\bot$, allowing a process with ticket number $i+1$ to leave the critical section and return the ticket.

The system is specified by the formula **sys** which is the conjunction of the formula **initial** specifying the set of initial configurations and the formula $\Box(\exists i : \mathbf{a}(i) \vee \mathbf{idle})$ specifying that in each step either some action $\mathbf{a}(i)$ is taken by process i, or the system idles. The idle transitions are needed to verify liveness properties.

Mutual exclusion can be specified by the formula

$$\mathbf{mutex} \quad = \quad \Box \neg(\exists i : \exists j : i \neq j \wedge q[i] = C \wedge q[j] = C) \ .$$

In order to specify non-starvation, we add a fairness assumption for the actions $\mathbf{enter}(i)$ and $\mathbf{exit}(i)$. We add no fairness assumption for $\mathbf{ticket}(i)$, since the arrival of new processes should not be controlled by the algorithm itself.

$$\begin{array}{lcl}
\mathbf{faira}(i) & = & (\mathbf{enter}(i) \vee \mathbf{exit}(i)) \\
 & & \wedge (\forall j \neq i : \mathbf{copy}(j)) \\
\mathbf{fairness} & = & \forall i : \Box\Diamond \left[\begin{array}{c} \mathbf{faira}(i) \\ \vee \neg\mathbf{Enabled}(\mathbf{faira}(i)) \end{array} \right] \\
\mathbf{non\text{-}starvation} & = & \forall i : \Box\,(q[i] = W \longrightarrow \Diamond q[i] = C)
\end{array}$$

To check that the algorithm satisfies mutual exclusion and non-starvation, we check whether the formulas

$$\begin{array}{l}
\mathbf{sys} \wedge \neg\mathbf{mutex} \\
\mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}
\end{array}$$

have any models.

The property that models are of arbitrary but fixed size implies that we actually verify the algorithm under the assumption that there is an arbitrarily chosen upper bound on the number of tickets in use at any time. For safety properties, this is not a limitation since violations will be finite sequences of execution steps, but for fairness assumptions it can play a role. For the bakery algorithm, it can be seen that an arbitrary upper limit on ticket numbers does not affect non-starvation for waiting processes, but in general one must be aware of this modeling constraint.

5.2 Szymanski's Algorithm

In the previous example there were an arbitrary number of processes, but there was a complete symmetry between the processes. In this example we will look at another algorithm that works for an arbitrary number of processes, but with the difference that they are organized in a linear array and thus will not be completely symmetric with respect to each other.

```
1:  await $\forall j : j \neq i : \neg s[j]$
2:  $w[i], s[i] := true, true$
3:  if $\exists j : j \neq i : (pc[j] \neq 1) \wedge (\neg w[j])$
        then $s[i] := false$ ; goto 4
        else $w[i] := false$ ; goto 5
4:  await $\exists j : j \neq i : s[j] \wedge \neg w[j]$
        then $w[i], s[i] := false, true$
5:  await $\forall j : j \neq i : \neg w[j]$
6:  await $\forall j : j < i : \neg s[j]$
7:  $s[i] := false$ ; goto 1
```

Fig. 4. Szymanski's algorithm

In Szymanski's algorithm for mutual exclusion [35, 20], there are an arbitrary number of processes organized in a linear array, where the index of the array denotes the process ID. In the algorithm, the local state of each process i consists of a control state $pc[i]$, ranging over the integers from 1 to 7 and of two boolean flags, $w[i]$ and $s[i]$. A process i is in the critical section when its control state $pc[i]$ is equal to 7. We model this using three variables named pc, and w, and s, ranging over an array of the same length as the number of processes. The behavior for each process i is given in Fig. 4, expressed in pseudo-code where the lines are numbered with the value of the control state pc. The version considered here is an idealized version. In most implementations a global guard (such as, e.g., $\forall j : j < i : s[j]$) is not atomic: in a more refined description of the algorithm this is a loop which checks the states of other processes.

For instance, according to the statement at line 6, if the control state of a process i is 6, and the value of s is *false* in all processes with a lower index (i.e., for all processes j with $j < i$), then the control state of process i may be changed to 7. In a similar manner, according to the statement at line 4, if the control state of a process i is 4, and if there is at least another process j (either with a lower index or a higher index than i) where the value of $s[j]$ is *true* and the value of $w[j]$ is *false*, then the control state, $w[i]$, and $s[i]$, in i may be changed to 5, *false*, and *true*, respectively.

The full model in $LTL(MSO)$ is given in Fig. 5. Auxiliary predicates **copy**, **copy-w**, **copy-s** and **copy-other** have been added to denote that some variables are not affected by the transition. The action formulas $\mathbf{a1}(i)$ through $\mathbf{a7}(i)$ are used to specify the transitions in the algorithm. To see how the above statements are modeled, line 1 can for example be modeled by the following formula:

$$\exists i : \left[\begin{array}{l} \quad pc[i](1,2) \\ \wedge\ (\forall j : j \neq i : \neg s[j]) \\ \wedge\ w'[i] = w[i] \ \wedge\ s'[i] = s[i] \end{array} \right]$$

$$
\begin{array}{lcl}
\mathbf{copy}(i) & = & pc[i] = pc'[i] \wedge w[i] = w'[i] \wedge s[i] = s'[i] \\
\mathbf{idle} & = & \forall i : \mathbf{copy}(i) \\
\mathbf{copy\text{-}w}(i) & = & w[i] = w'[i] \\
\mathbf{copy\text{-}s}(i) & = & s[i] = s'[i] \\
\mathbf{copy\text{-}other}(i) & = & (\forall j \neq i : \mathbf{copy}(j)) \\
\mathbf{a1}(i) & = & pc[i](1,2) \wedge (\forall j \neq i : \neg s[j]) \wedge \\
 & & \mathbf{copy\text{-}w}(i) \wedge \mathbf{copy\text{-}s}(i) \\
\mathbf{a2}(i) & = & pc[i](2,3) \wedge w'[i] \wedge s'[i] \\
\mathbf{a3a}(i) & = & pc[i](3,4) \wedge \neg s'[i] \wedge \mathbf{copy\text{-}w}(i) \wedge \\
 & & \exists j \neq i : \neg(pc[j] = 1) \wedge \neg w[j] \\
\mathbf{a3b}(i) & = & pc[i](3,5) \wedge \neg w'[i] \wedge \mathbf{copy\text{-}s}(i) \wedge \\
 & & \neg(\exists j \neq i : \neg(pc[j] = 1) \wedge \neg w[j]) \\
\mathbf{a3}(i) & = & \mathbf{a3a}(i) \vee \mathbf{a3b}(i) \\
\mathbf{a4}(i) & = & pc[i](4,5) \wedge \neg w'[i] \wedge s'[i] \wedge \\
 & & (\exists j \neq i : s[j] \wedge \neg w[j]) \\
\mathbf{a5}(i) & = & pc[i](5,6) \wedge (\forall j \neq i : \neg w[j]) \wedge \\
 & & \mathbf{copy\text{-}w}(i) \wedge \mathbf{copy\text{-}s}(i) \\
\mathbf{a6}(i) & = & pc[i](6,7) \wedge (\forall j < i : \neg s[j]) \wedge \\
 & & \mathbf{copy\text{-}w}(i) \wedge \mathbf{copy\text{-}s}(i) \\
\mathbf{a7}(i) & = & pc[i](7,1) \wedge \neg s'[i] \wedge \mathbf{copy\text{-}w}(i) \\
\mathbf{a}(i) & = & \mathbf{a1}(i) \vee \mathbf{a2}(i) \vee \mathbf{a3}(i) \vee \mathbf{a4}(i) \vee \\
 & & \mathbf{a5}(i) \vee \mathbf{a6}(i) \vee \mathbf{a7}(i) \\
\mathbf{initial} & = & \forall i : pc[i] = 1 \\
\mathbf{sys} & = & \mathbf{initial} \wedge \\
 & & \Box(\exists i : (\mathbf{a}(i) \wedge \mathbf{copy\text{-}other}(i)) \vee \mathbf{idle}) \\
\mathbf{fairness} & = & \forall i : \Box\Diamond(\mathbf{a}(i) \vee \neg\mathbf{Enabled}(\mathbf{a}(i))) \\
\mathbf{mutex} & = & \Box\neg\exists i : \exists j : i \neq j \wedge pc[i] = 7 \wedge pc[j] = 7 \\
\mathbf{non\text{-}starvation} & = & \forall i : \Box\,(pc[i] = 2 \rightarrow \Diamond pc[i] = 7) \\
\mathbf{safety} & = & \mathbf{sys} \wedge \neg\mathbf{mutex} \\
\mathbf{liveness} & = & \mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}
\end{array}
$$

Fig. 5. Szymanski's algorithm in *LTL*(*MSO*)

where the difference to line 1 is mainly that the program counter *pc* is made explicit.

Like in the Bakery algorithm in Section 5.1, we add a system formula **sys** by conjoining the formula **initial** specifying the set of initial configurations and the formulas for the transitions of the algorithm. The formulas **safety** for verifying mutual exclution and **liveness** for verifying non-starvation are also written in a similar way.

5.3 Dijkstra's Algorithm

In Fig. 6, we show an idealized version of Dijkstra's protocol [26] for ensuring mutual exclusion among an arbitrary number of processes. Each process i has a control state ranging over the integers from 1 to 7 and a variable $flag[i]$ ranging over $\{0, 1, 2\}$. Furthermore, a global variable p ranging over process indices is used. In the algorithm, line 6 represents the critical section.

We model the global variable with a configuration variable p such that $p[i]$ is true iff the global variable p points to process i. The resulting *LTL*(*MSO*) model is given in Fig. 7.

```
1:   $flag[i] := 1$
2:   if $p \neq i$ then
         await $flag[p] = 0$ then
3:           $p := i$
4:   $flag[i] := 2$
5:   if $\exists j \neq i : flag[j] = 2$ then goto 1
6:   $flag[i] := 0$ ; goto 1
```

Fig. 6. Dijkstra's algorithm

$$
\begin{array}{lcl}
\mathbf{copy}(i) & = & pc[i] = pc'[i] \wedge flag[i] = flag'[i] \wedge \\
 & & p[i] = p'[i] \\
\mathbf{copy\text{-}flag}(i) & = & flag[i] = flag'[i] \\
\mathbf{copy\text{-}p} & = & \forall k : p[k] = p'[k] \\
\mathbf{copy\text{-}other}(i) & = & \forall j \neq i : \mathbf{copy}(j) \\
\mathbf{idle} & = & \forall i : \mathbf{copy}(i) \\
\mathbf{set\text{-}p}(i) & = & \forall j : p'[j] \leftrightarrow j = i \\
\mathbf{zeropflag} & = & \forall k : (p[k] \longrightarrow flag[k] = 0) \\
\mathbf{a1}(i) & = & pc[i](1,2) \wedge flag'[i] = 1 \wedge \mathbf{copy\text{-}p} \\
\mathbf{a2a}(i) & = & pc[i](2,3) \wedge \neg p[i] \wedge \mathbf{zeropflag} \wedge \mathbf{copy\text{-}p} \\
\mathbf{a2b}(i) & = & pc[i](2,4) \wedge p[i] \wedge \mathbf{copy\text{-}flag}(i) \wedge \mathbf{copy\text{-}p} \\
\mathbf{a2}(i) & = & \mathbf{a2a}(i) \vee \mathbf{a2b}(i) \\
\mathbf{a3}(i) & = & pc[i](3,4) \wedge \mathbf{set\text{-}p}(i) \wedge \mathbf{copy\text{-}flag}(i) \\
\mathbf{a4}(i) & = & pc[i](4,5) \wedge flag'[i] = 2 \wedge \mathbf{copy\text{-}p} \\
\mathbf{a5a}(i) & = & pc[i](5,1) \wedge \mathbf{copy\text{-}flag}(i) \wedge \mathbf{copy\text{-}p} \wedge \\
 & & \exists j \neq i : flag[j] = 2 \\
\mathbf{a5b}(i) & = & pc[i](5,6) \wedge \mathbf{copy\text{-}flag}(i) \wedge \mathbf{copy\text{-}p} \wedge \\
 & & \neg\exists j \neq i : flag[j] = 2 \\
\mathbf{a5}(i) & = & \mathbf{a5a}(i) \vee \mathbf{a5b}(i) \\
\mathbf{a6}(i) & = & pc[i](6,1) \wedge flag'[i] = 0 \wedge \mathbf{copy\text{-}p} \\
\mathbf{a}(i) & = & \mathbf{a1}(i) \vee \mathbf{a2}(i) \vee \mathbf{a3}(i) \vee \\
 & & \mathbf{a4}(i) \vee \mathbf{a5}(i) \vee \mathbf{a6}(i) \\
\mathbf{initial} & = & \forall i : pc[i] = 1 \wedge flag[i] = 0 \wedge \neg p[i] \\
\mathbf{sys} & = & \mathbf{initial} \wedge \\
 & & \Box(\exists i : (\mathbf{a}(i) \wedge \mathbf{copy\text{-}other}(i)) \vee \mathbf{idle}) \\
\mathbf{fairness} & = & \forall i : \Box\Diamond(\mathbf{a}(i) \vee \neg\mathbf{Enabled}(\mathbf{a}(i))) \\
\mathbf{mutex} & = & \Box\neg\exists i : \exists j : i \neq j \wedge pc[i] = 6 \wedge pc[j] = 6 \\
\mathbf{non\text{-}starvation} & = & \forall i : \Box\,(pc[i] = 1 \rightarrow \Diamond pc[i] = 6) \\
\mathbf{safety} & = & \mathbf{sys} \wedge \neg\mathbf{mutex} \\
\mathbf{liveness} & = & \mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}
\end{array}
$$

Fig. 7. Dijkstra's algorithm in *LTL*(*MSO*)

```
1:   $flag[i] := 0$
2:   if $\exists j < i : flag[j] = 1$ then goto 1
3:   $flag[i] := 1$
4:   if $\exists j < i : flag[j] = 1$ then goto 1
5:   await $\forall j > i : flag[j] \neq 1$
6:   $flag[i] := 0$ ; goto 1
```

Fig. 8. Burns's algorithm

5.4 Burns's Algorithm

Burns's mutual exclusion algorithm [26] is given in Fig. 8. Each process i has a control state ranging over the integers from 1 to 7 and a variable $flag[i]$ ranging over $\{0, 1\}$. The critical section is represented by line 6.

$$
\begin{array}{lll}
\mathbf{copy}(i) & = & pc[i] = pc'[i] \wedge flag[i] = flag'[i] \\
\mathbf{copy\text{-}flag}(i) & = & flag[i] = flag'[i] \\
\mathbf{copy\text{-}other}(i) & = & \forall j \neq i : \mathbf{copy}(j) \\
\mathbf{idle} & = & \forall i : \mathbf{copy}(i) \\
\mathbf{a1}(i) & = & pc[i](1,2) \wedge \neg flag'[i] \\
\mathbf{a2a}(i) & = & pc[i](2,1) \wedge (\exists j < i : flag[j]) \wedge \\
 & & \mathbf{copy\text{-}flag}(i) \\
\mathbf{a2b}(i) & = & pc[i](2,3) \wedge (\neg\exists j < i : flag[j]) \wedge \\
 & & \mathbf{copy\text{-}flag}(i) \\
\mathbf{a2}(i) & = & \mathbf{a2a}(i) \vee \mathbf{a2b}(i) \\
\mathbf{a3}(i) & = & pc[i](3,4) \wedge flag'[i] \\
\mathbf{a4a}(i) & = & pc[i](4,1) \wedge (\exists j < i : flag[j]) \wedge \\
 & & \mathbf{copy\text{-}flag}(i) \\
\mathbf{a4b}(i) & = & pc[i](4,5) \wedge (\neg\exists j < i : flag[j]) \wedge \\
 & & \mathbf{copy\text{-}flag}(i) \\
\mathbf{a4}(i) & = & \mathbf{a4a}(i) \vee \mathbf{a4b}(i) \\
\mathbf{a5}(i) & = & pc[i](5,6) \wedge (\forall j > i : \neg flag[j]) \wedge \\
 & & \mathbf{copy\text{-}flag}(i) \\
\mathbf{a6}(i) & = & pc[i](6,1) \wedge \neg flag'[i] \\
\mathbf{a}(i) & = & \mathbf{a1}(i) \vee \mathbf{a2}(i) \vee \mathbf{a3}(i) \vee \\
 & & \mathbf{a4}(i) \vee \mathbf{a5}(i) \vee \mathbf{a6}(i) \\
\mathbf{initial} & = & \forall i : pc[i] = 1 \wedge flag[i] = 0 \\
\mathbf{sys} & = & \mathbf{initial} \wedge \\
 & & \Box(\exists i : (\mathbf{a}(i) \wedge \mathbf{copy\text{-}other}(i)) \vee \mathbf{idle}) \\
\mathbf{fairness} & = & \forall i : \Box\Diamond(\mathbf{a}(i) \vee \neg\mathbf{Enabled}(\mathbf{a}(i))) \\
\mathbf{mutex} & = & \Box\neg\exists i : \exists j : i \neq j \wedge pc[i] = 6 \wedge pc[j] = 6 \\
\mathbf{non\text{-}starvation} & = & \forall i : \Box\,(pc[i] = 1 \rightarrow \Diamond pc[i] = 6) \\
\mathbf{safety} & = & \mathbf{sys} \wedge \neg\mathbf{mutex} \\
\mathbf{liveness} & = & \mathbf{sys} \wedge \mathbf{fairness} \wedge \neg\mathbf{non\text{-}starvation}
\end{array}
$$

Fig. 9. Burns's algorithm in *LTL*(*MSO*)

We model the values 0 and 1 with the booleans such that 0 is false and 1 is true. The *LTL*(*MSO*) model for the algorithm is given in Fig. 9.

5.5 A Termination Detection Algorithm

We can also models ring shaped parameterized systems in our framework, which we illustrate with an algorithm for termination detection among an arbitrary number of processes organized in a ring shaped network, due to Dijkstra et al. [11]. The algorithm uses a colored token which is passed around the ring to check that all processes in the ring have terminated.

A process can either be *non-idle* or *idle*. When all processes are idle, we say that the system has terminated. A process can spontaneously change its state from non-idle to idle, i.e., it terminates. To detect that all processes are idle, a designated processes sends out a token which it colors *white*. When the token is passed to the next processes, the process passing the token paints it black if it is non-idle. When the token comes back to the process which sent out the token, it is white if the system has terminated, and black otherwise.

The system can be modeled by numbering the processes from 0 to $n-1$ and using three arrays holding three local variables the processes. Only process 0 may initiate the algorithm by sending out a new token. The variables are $q[i]$ which is true iff process i is idle, and $t[i]$ ranging over $\{\mathbf{black}, \mathbf{white}, \mathbf{none}\}$, which has the value **none** when process i does *not* have the token, and otherwise denotes the color of the token. In addition, process 0 has a boolean variable w, which is true if it has stayed idle during the current round. The value of w is only relevant for process 0.

```
-    $q[i] := true$
-    if $i > 0 \wedge \neg q[i-1]$
         then $q[i] := false$
-    if $\neg q[n-1]$
         then $q[0], w := false, false$
-    if $i = 0 \wedge q[0] \wedge (t[0] = \mathbf{black} \vee \neg w)$
         then $t[0], t[1], w := \mathbf{none}, \mathbf{white}, true$
-    if $i < n-1 \wedge t[i] \neq \mathbf{none} \wedge q[i]$
         then $t[i], t[i+1] := \mathbf{none}, t[i]$
-    if $i = n-1 \wedge t[n-1] \neq \mathbf{none} \wedge \neg q[n-1]$
         then $t[n-1], t[0] := \mathbf{none}, t[i]$
-    if $i < n-1 \wedge t[i] \neq \mathbf{none} \wedge \neg q[i]$
         then $t[i], t[i+1] := \mathbf{none}, \mathbf{black}$
-    if $i = n-1 \wedge t[n-1] \neq \mathbf{none} \wedge \neg q[n-1]$
         then $t[n-1], t[0] := \mathbf{none}, \mathbf{black}$
```

Fig. 10. A Termination Detection Algorithm

Initially, we have $q[i] = false$ for all i, and $t[0] = \mathbf{black}$, and $t[i] = \mathbf{none}$ for all $0 < i < n$, and $w = false$. The algorithm can be described by the statements in Fig. 10, for each process i.

The three first types of statements describe the underlying computation: a process can become idle autonomously (first statement), and it can become non-idle if its predecessor is non-idle (second statement). In addition (third statement), process 0 must set w to *false* if it becomes non-idle. The fourth statement starts a round of the detection algorithm. In the next two statements, a process just forwards the token if it is idle. Finally, in the last two statements, if a process is non-idle, the token is painted black and then forwarded. Note how the ring is modeled by allowing process $n-1$ to communicate with process 0.

The model is given in Fig. 11. The formula **safety** is used to verify that if process 0 signals termination, then all processes are idle.

6 Communication Protocols

Our framework can be used to model queues and stacks by letting each position in the word represent a position in the queue or the stack. Integer variables can also be modeled, using the word to represent the digits of the word in some base. These data types are common in communication protocols, where processes communicate through a queue and integer variables can be used to model sequence numbers of the messages that are

$$
\begin{array}{lcl}
\mathbf{copy}(i) & = & t[i]=t'[i] \wedge w[i]=w'[i] \wedge q[i]=q'[i] \\
\mathbf{copy\text{-}other}(i) & = & \forall j \neq i : \mathbf{copy}(j) \\
\mathbf{copy\text{-}other2}(i,j) & = & \forall k : \neg(k=i \vee k=j) \rightarrow \mathbf{copy}(k) \\
\mathbf{copy\text{-}q}(i) & = & q[i]=q'[i] \\
\mathbf{copy\text{-}t}(i) & = & t[i]=t'[i] \\
\mathbf{idle} & = & \forall i : \mathbf{copy}(i) \\
\mathbf{move\text{-}token}(i,j) & = & t[i]=t'[j] \\
\mathbf{adjacent}(i,j) & = & j=i+1 \vee (j=0 \wedge i=\$) \\
\mathbf{pass}(i,j) & = & i \neq 0 \wedge t[i] \neq \mathbf{none} \wedge \\
 & & (\neg q[i] \longrightarrow t'[j]=\mathbf{black}) \wedge \\
 & & (q[i] \longrightarrow \mathbf{move\text{-}token}(i,j)) \wedge \\
 & & t'[i]=\mathbf{none} \wedge \\
 & & \mathbf{copy\text{-}q}(i) \wedge \mathbf{copy\text{-}q}(j) \wedge \\
 & & w[0]=w'[0] \\
\mathbf{start}(i,j) & = & i=0 \wedge q[i] \wedge \\
 & & (t[i]=\mathbf{black} \vee \neg w[0]) \wedge \\
 & & t'[i]=\mathbf{none} \wedge t'[j]=\mathbf{white} \wedge \\
 & & w'[0] \wedge \mathbf{copy\text{-}q}(i) \wedge \mathbf{copy\text{-}q}(j) \\
\mathbf{comp1}(i) & = & q'[i] \wedge \mathbf{copy\text{-}t}(i) \wedge w[0]=w'[0] \\
\mathbf{comp2}(i,j) & = & \neg q[i] \wedge \mathbf{copy}(i) \wedge \mathbf{copy\text{-}t}(j) \wedge q'[j] \wedge \\
 & & (j=0 \longrightarrow \neg w'[0]) \wedge \\
 & & (j \neq 0 \longrightarrow w[0]=w'[0]) \\
\mathbf{a1}(i) & = & \mathbf{copy\text{-}other}(i) \wedge \mathbf{comp1}(i) \\
\mathbf{a2}(i) & = & \exists j : \mathbf{adjacent}(i,j) \wedge \\
 & & \mathbf{copy\text{-}other2}(i,j) \wedge \\
 & & (\mathbf{start}(i,j) \vee \mathbf{pass}(i,j) \vee \mathbf{comp2}(i,j)) \\
\mathbf{a}(i) & = & \mathbf{a1}(i) \vee \mathbf{a2}(i) \\
\mathbf{initial} & = & \forall i : (i=0 \longrightarrow t[i]=\mathbf{black} \wedge \neg w[i]) \wedge \\
 & & (i \neq 0 \longrightarrow t[i]=\mathbf{none}) \wedge \neg q[i] \\
\mathbf{sys} & = & \mathbf{initial} \wedge \Box(\exists i : \mathbf{a}(i) \vee \mathbf{idle}) \\
\mathbf{termination} & = & \Box \left[\begin{array}{l} (\exists i = 0 : t[i]=\mathbf{white} \wedge w[0]) \\ \quad \longrightarrow \forall i : q[i] \end{array} \right] \\
\mathbf{safety} & = & \mathbf{sys} \wedge \neg\mathbf{termination}
\end{array}
$$

Fig. 11. A Termination Detection Algorithm in *LTL*(*MSO*)

passed. We will use communication protocols to illustrate how we can represent these data types and operations on them.

Queues and Stacks Let us describe how to represent queues and stacks in our framework. We use a configuration variable q where $q[i]$ is the queue or stack content at position i. Since our transitions preserve the length of the words, we cannot dynamically create new positions. Therefore, to allow for a dynamic data structure, we add a *padding symbol* $\perp$ to represent empty slots. Recall that configurations are of arbitrary length, so even though we can not model unbounded queues, we can model arbitrary-length queues. The difference between unbounded and arbitrary length can play a role for liveness properties, but not for safety properties.

Below, we model sending and receiving a message denoted by the parameter m to and from a queue represented using a configuration variable denoted by the parameter q. Messages are sent by replacing the $\perp$ to the right of the rightmost message, and received by replacing the leftmost message by a $\perp$. The empty queue is described by $\mathbf{empty}(q)$.

$$
\mathbf{send}(q,m) \quad = \quad \exists i : \left[\begin{array}{l} \quad q'[i]=m \wedge q[i]=\perp \\ \wedge\ \forall j \neq i : q[j]=q'[j] \\ \wedge\ \forall j : i=j+1 \rightarrow q[j] \neq \perp \\ \wedge\ \forall j > i : q[j]=\perp \end{array} \right]
$$

$$
\begin{array}{lcl}
\mathbf{receive}(q,m) & = & \exists i : \left[\begin{array}{l} \quad q'[i]=\perp \wedge q[i]=m \\ \wedge\ \forall j \neq i : q[j]=q'[j] \\ \wedge\ \forall j < i : q[j]=\perp \end{array} \right] \\
\mathbf{empty}(q) & = & \forall i : q[i]=\perp
\end{array}
$$

Using this technique for modeling a queue, the contents of the queue do not change position. Send and receive operations change only a single position in the word. This property makes it easier to analyze the model using our verification techniques, described in [8,2,28]. A side-effect is that the contents of the queue will shift towards right unless the queue becomes empty. This makes no difference for the verification of safety properties, since the queue is initialized with any finite capacity, and can be made large enough to accomodate any finite execution.

For stacks, we model the push and pop operations below. The stack grows from left to right. The empty stack is described by $\mathbf{empty}(q)$.

$$
\mathbf{push}(q,m) \quad = \quad \left[\begin{array}{l} \quad q'[i]=m \wedge q[i]=\perp \\ \wedge\ \forall j \neq i : q[j]=q'[j] \\ \wedge\ \forall j : i=j+1 \rightarrow q[j] \neq \perp \\ \wedge\ \forall j > i : q[j]=\perp \end{array} \right]
$$

$$
\begin{array}{lcl}
\mathbf{pop}(q,m) & = & \left[\begin{array}{l} \quad q'[i]=\perp \wedge q[i]=m \\ \wedge\ \forall j \neq i : q[j]=q'[j] \\ \wedge\ \forall j : i=j+1 \rightarrow q[j] \neq \perp \\ \wedge\ \forall j > i : q[j]=\perp \end{array} \right] \\
\mathbf{empty}(q) & = & \forall i : q[i]=\perp
\end{array}
$$

We model sends to *lossy channels*, where messages may be lost, with the formula $\mathbf{lossend}(q,m)$ defined as

$$
\mathbf{send}(q,m) \vee \forall i : q[i]=q'[i]
$$

i.e., the message can be lost immediately when sending.

Integers Integer variables can be represented in many ways using a word. One alternative is to use a binary encoding of the integer value, such that the word represents the value of the integer variable in binary with the most significant bit to the left. This has the advantage that addition and multiplication can be performed using a regular transition relation. For example, if we use the configuration variable x and y to represent two numbers, the operation $x := x+y$ can be modeled by the formula

$$
\exists C : \left[\begin{array}{l} \quad \$ \notin C \\ \wedge\ \forall i : (x'[i] \leftrightarrow x[i]) \leftrightarrow (y[i] \leftrightarrow i \in C) \\ \wedge\ \forall i : i-1 \in C \leftrightarrow \left[\begin{array}{l} \quad (x[i] \wedge y[i]) \\ \vee\ (x[i] \wedge i \in C) \\ \vee\ (y[i] \wedge i \in C) \end{array} \right] \end{array} \right]
$$

The second-order variable C is used to implement a carry-bit in the addition. The formula consists of three conjuncts. The first sets the carry-bit to false in the last position, corresponding to the least significant bit. The second part adds $x[i]$ and $y[i]$ and the carrybit $i \in C$, putting the result in $x'[i]$ (to see this, note that $(\varphi_1 \leftrightarrow \varphi_2) \leftrightarrow (\varphi_3 \leftrightarrow \varphi_4)$ is true iff an even number of the formulas $\varphi_1, \varphi_2, \varphi_3, \varphi_4$ are true). The last part updates the carry-bit for $i - 1$ in case there was an overflow.

The binary encoding works well when the system consists only of integer variables and has been used for the verification of numerous examples, for example in the tool LASH [6]. When integer variables are used in combination with other datatypes, for example as a process index or a sequence number in a communication protocol, it can be more natural to use a *unary encoding*. With this encoding, addition and multiplication can not be expressed as a regular transition relation, but operations relating the variable with the other datatypes, for example changing the state of a process pointed to by a process index variable, can be performed.

In the following subsections, we model two communication protocols using the encodings of data types described above.

6.1 The Alternating Bit Protocol

We illustrate encoding of queues in our framework with the well-known Alternating Bit Protocol [4], a protocol used for delivering messages over unbounded channels which are faulty in the sense that they may lose messages but not reorder them.

There are two channels, one for sending messages from the sender to the receiver, and one for sending acknowledgments from the receiver to the sender. Each message is given a sequence number and the sender waits for an acknowledgment from the receiver before sending a new message. Until this acknowledgment is received, the sender may resend the message. When the receiver has acknowledged the message, the procedure is repeated but with the sequence number inverted. Both the sender and the receiver ignore messages with unexpected sequence numbers.

To model the service provided by the protocol, we consider two operations **protsend** and **protreceive**, modeling calls from the upper layers of the protocols. Thus, **protsend** denotes that there is a new message from the sender side, and **protreceive** denotes that the receiver side signals that a message has been received. We denote the two channels *msg* and *ack*, where *msg* is the channel used for messages and *ack* is the channel used for acknowledgments.

A high level description for the sender and the receiver is given in Fig. 12. The notation S **OR** S' means that either S or S' is executed, but not both of them.

One property of the algorithm specifies that the operations **protsend** and **protreceive** alternate after each

```
Sender
 1:   protsend
 2:   (lossend(msg, 0)  OR  receive(ack, 1)) ; goto 2
       OR
      receive(ack, 0)
 3:   protsend
 4:   (lossend(msg, 1)  OR  receive(ack, 0)) ; goto 4
       OR
      receive(ack, 1) ; goto 1

Receiver
 1:   (lossend(ack, 1)  OR  receive(msg, 1)) ; goto 1
       OR
      receive(msg, 0)
 2:   protreceive
 3:   (lossend(ack, 0)  OR  receive(msg, 0)) ; goto 3
       OR
      receive(msg, 1)
 4:   protreceive ; goto 1
```

Fig. 12. The Alternating Bit Protocol

other such that the two operations never occur consecutively. We model this by adding an *observer* that records the last operation (**protsend** or **protreceive**) initialized to **protreceive** and checks that a **protsend** operation can not occur when the observer is in state **protsend** and similarly that a **protreceive** operation can not occur when the observer is in state **protreceive**.

An *LTL*(*MSO*) model of the Alternating Bit Protocol is given in Fig. 13.

6.2 A Sliding Window Protocol

We illustrate the use of integers with a sliding window protocol (for a general description on sliding window protocols, see, e.g., Tannenbaum [36] Ch. 3). Like the Alternating Bit Protocol, the protocol is intended to provide reliable transmission of messages across an unreliable channel.

The sender and receiver employ a so-called sliding window protocol, in which messages sent over the channel are provided with a sequence number, assigned in a cyclic fashion from 0 to $max - 1$ and then starting at 0 again. The receiver acknowledges messages using a separate channel, which we model with a direct communication between the receiver and the sender.

Initially, the sender transmits messages with consecutive sequence numbers $0, 1, 2$, etc. Since the channel may lose messages, the sender cannot know whether the messages will reach the receiver. Therefore, the sender also waits for acknowledgments from the receiver. An acknowledgment with sequence number n signals that the receiver has correctly received messages up to sequence number $n - 1$. There must never be more than $max - 1$ outstanding messages. Therefore, after sending messages 0 through $max - 2$, the sender must wait for

$$
\begin{array}{lcl}
\textbf{copy}(x) & = & \forall i : x[i] = x'[i] \\
\textbf{copy-other}(x,i) & = & \forall j \neq i : x[j] = x'[j] \\
\textbf{copy-channels} & = & \textbf{copy}(msg) \wedge \textbf{copy}(ack) \\
\textbf{idle} & = & \textbf{copy-channels} \wedge \textbf{copy}(pc) \wedge \textbf{copy}(obs) \\
\textbf{observe}(v) & = & obs'[0] = v \wedge \textbf{copy-other}(obs,i) \\
\textbf{sa1} & = & pc[0](1,2) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{copy-channels} \wedge \textbf{observe}(\textbf{protsend}) \\
\textbf{sa2a} & = & pc[0](2,2) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{lossend}(msg,0) \wedge \textbf{copy}(ack) \wedge \textbf{copy}(obs) \\
\textbf{sa2b} & = & pc[0](2,2) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{receive}(ack,1) \wedge \textbf{copy}(msg) \wedge \textbf{copy}(obs) \\
\textbf{sa2c} & = & pc[0](2,3) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{receive}(ack,0) \wedge \textbf{copy}(msg) \wedge \textbf{copy}(obs) \\
\textbf{sa3} & = & pc[0](3,4) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{copy-channels} \wedge \textbf{observe}(\textbf{protsend}) \\
\textbf{sa4a} & = & pc[0](4,4) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{lossend}(msg,1) \wedge \textbf{copy}(ack) \wedge \textbf{copy}(obs) \\
\textbf{sa4b} & = & pc[0](4,4) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{receive}(ack,0) \wedge \textbf{copy}(msg) \wedge \textbf{copy}(obs) \\
\textbf{sa4c} & = & pc[0](4,1) \wedge \textbf{copy-other}(pc,0) \wedge \\
 & & \textbf{receive}(ack,1) \wedge \textbf{copy}(msg) \wedge \textbf{copy}(obs) \\
\textbf{sender} & = & \textbf{sa1} \vee \textbf{sa2a} \vee \textbf{sa2b} \vee \textbf{sa2c} \vee \textbf{sa3} \vee \\
 & & \textbf{sa4a} \vee \textbf{sa4b} \vee \textbf{sa4c} \\
\textbf{receiver} & = & \textit{Defined similarly as sender with} \\
 & & pc[1] \textit{ instead of } pc[0] \textit{ and observing} \\
 & & \textbf{protreceive} \\
\textbf{a} & = & \textbf{sender} \vee \textbf{receiver} \\
\textbf{initial} & = & pc[0] = 1 \wedge pc[1] = 1 \wedge \\
 & & \textbf{empty}(msg) \wedge \textbf{empty}(ack) \wedge \\
 & & obs[0] = \textbf{protreceive} \\
\textbf{sys} & = & \textbf{initial} \wedge \Box(\textbf{a} \vee \textbf{idle}) \\
\textbf{receivealt} & = & \Box\,(obs[0] = \textbf{protreceive} \rightarrow \neg(\textbf{ra2} \vee \textbf{ra4})) \\
\textbf{sendalt} & = & \Box\,(obs[0] = \textbf{protsend} \rightarrow \neg(\textbf{sa1} \vee \textbf{sa3})) \\
\textbf{safety} & = & \textbf{sys} \wedge \neg\textbf{sendalt} \wedge \neg\textbf{receivealt}
\end{array}
$$

Fig. 13. The Alternating Bit Protocol in *LTL*(*MSO*)

```
Initially, $low = 0$, $next = 0$, and $high = 0$.
1:  (enlarge window)
        if $low \neq high + 1$
        then $high := high + 1$
2:  (send)
    for any $n$ if
        $\left[\begin{array}{l} (low \leq high \rightarrow low \leq n \wedge n < high) \\ \wedge\ (high < low \rightarrow low \leq n \vee n < high) \end{array}\right]$
        then send$(c, n)$
3:  (receive)
        receive$(c, next)$; $next := next + 1$
4:  (synchronous ack)
        $low := next$
```

Fig. 14. A Sliding Window Protocol

$$
\begin{array}{lcl}
\textbf{copy}(x) & = & \forall i : x[i] = x'[i] \\
\textbf{copy-other}(x,i) & = & \forall j \neq i : x[j] = x'[j] \\
\textbf{copy-channel} & = & \textbf{copy}(c_1) \wedge \textbf{copy}(c_2) \wedge \textbf{copy}(c_3) \\
\textbf{copy-proc} & = & \textbf{copy}(low) \wedge \textbf{copy}(high) \wedge \textbf{copy}(next) \\
\textbf{idle} & = & \textbf{copy-channel} \wedge \textbf{copy-proc} \\
\textbf{adjacent}(i,j) & = & j = i + 1 \vee (j = 0 \wedge i = \$) \\
\textbf{between}(i,j,k) & = & (i \leq k \rightarrow i \leq j \wedge j < k) \wedge \\
 & & (k < i \rightarrow i \leq j \vee j < k) \\
\textbf{addone}(x) & = & \exists p, q : \textbf{adjacent}(p,q) \wedge \\
 & & x[p] \wedge \neg x'[p] \wedge x'[q] \wedge \\
 & & \forall r : r = p \vee r = q \vee (\neg x[r] \wedge \neg x'[r]) \\
\textbf{allfalse}(x) & = & \forall i : \neg x[i] \\
\textbf{a1} & = & \exists l, h : low[l] \wedge high[h] \wedge \neg\textbf{adjacent}(h,l) \wedge \\
 & & \textbf{copy}(low) \wedge \textbf{addone}(high) \wedge \textbf{copy}(next) \wedge \\
 & & \textbf{copy-channel} \\
\textbf{a2} & = & \exists l, h, m : low[l] \wedge next[m] \wedge high[h] \wedge \\
 & & \textbf{between}(l,m,h) \wedge \textbf{copy-proc} \\
 & & \wedge\, c_1'[m] \wedge \textbf{allfalse}(c_1) \wedge \\
 & & \textbf{copy-other}(c_1,m) \wedge \textbf{copy}(c_2) \wedge \textbf{copy}(c_3) \\
\textbf{a3} & = & \exists n : c_3[n] \wedge \neg c_3'[n] \wedge next[n] \wedge \\
 & & \textbf{copy}(low) \wedge \textbf{copy}(high) \wedge \textbf{addone}(next) \wedge \\
 & & \textbf{copy}(c_1) \wedge \textbf{copy}(c_2) \wedge \textbf{copy-other}(c_3,n) \\
\textbf{a4} & = & (\forall j : low'[j] \leftrightarrow next[j]) \wedge \\
 & & \textbf{copy}(high) \wedge \textbf{copy}(next) \wedge \textbf{copy-channel} \\
\textbf{a5} & = & \exists j : c_1[j] \wedge \neg c_1'[j] \wedge \textbf{allfalse}(c_2) \wedge c_2'[j] \wedge \\
 & & \textbf{copy-proc} \wedge \textbf{copy-other}(c_1,j) \wedge \\
 & & \textbf{copy-other}(c_2,j) \wedge \textbf{copy}(c_3) \\
\textbf{a6} & = & \exists j : c_2[j] \wedge \neg c_2'[j] \wedge \textbf{allfalse}(c_3) \wedge c_3'[j] \wedge \\
 & & \textbf{copy-proc} \wedge \textbf{copy}(c_1) \wedge \\
 & & \textbf{copy-other}(c_2,j) \wedge \textbf{copy-other}(c_3,j) \\
\textbf{a} & = & \textbf{a1} \vee \textbf{a2} \vee \textbf{a3} \vee \textbf{a4} \vee \textbf{a5} \vee \textbf{a6} \\
\textbf{sys} & = & \textbf{initial} \wedge \Box(\textbf{a} \vee \textbf{idle}) \\
\textbf{initial} & = & \forall i : (i = 0 \leftrightarrow low[i]) \wedge (i = 0 \leftrightarrow high[i]) \wedge \\
 & & (i = 0 \leftrightarrow next[i]) \wedge \neg c_1[i] \wedge \neg c_2[i] \wedge \neg c_3[i] \\
\textbf{inside-window} & = & \Box\,\forall l, n, h : \\
 & & \left[\begin{array}{l} \quad low[l] \wedge next[n] \wedge high[h] \\ \rightarrow n = h \vee \textbf{between}(l,n,h) \end{array}\right] \\
\textbf{safety} & = & \textbf{sys} \wedge \neg\textbf{inside-window}
\end{array}
$$

Fig. 15. A Sliding Window Protocol in *LTL*(*MSO*)

an acknowledgment. After receiving an acknowledgment for a message, say 3, the sender may continue to send messages *max* − 1, 0, and 1. If no acknowledgment arrives for any outstanding messages, it is assumed to be lost and the sender should resend outstanding messages after some period of time.

The range of sequence number representing the outstanding messages is called the *sender window* and is modeled by two variables *low* and *high*, where the outstanding messages have sequence numbers n with $low \leq n < high$, if $low \leq high$, and with $low \leq n$ or $n < high$, if $high < low$. The integer variable *next* denotes the sequence number of the next message the receiver expects to receive. A high level version of the protocol is given in Fig. 14, where addition is performed modulo *max*.

We model this protocol in *LTL*(*MSO*) with a configuration variable for each of the integer variables with the same name. The formula $low[i]$ will be true if and only if the integer variable *low* is equal to i. The channel will be limited to a fixed capacity (say 3). Since the messages contain arbitrary sequence numbers and we have a finite alphabet, we can not model a channel of arbitrary size. Instead, we use three configuration variables c_1, c_2, and c_3, where $c_k[i]$ is true if and only if position k in the channel contains a message with sequence number i.

The full *LTL*(*MSO*) model is given in Fig. 15. The formula **a1** corresponds to enlarging the window, the formula **a2** to sending a message, the formula **a3** to receiving a message, the formula **a4** to a synchronous ack, and the formulas **a5** and **a6** to movement within the channel.

The safety property **inside-window** specifies that the receiver is never outside the sending window, which can be seen as a check that the protocol synchronizes correctly.

7 Büchi Normal Form

In this section, we describe how to transform a *restricted* formula in *LTL*(*MSO*) into an equivalent formula in *Büchi Normal Form*, defined as follows.

Definition 1. (**Büchi Normal Form**) A formula is in *Büchi Normal Form* if it is of the form

$$\phi_I \wedge \Box\,\phi_T \wedge \Box\Diamond\,\phi_F$$

where the formulas ϕ_I, ϕ_T, ϕ_F are MSO formulas without temporal operators, and ϕ_I contains no primed configuration variables. $\Box$

Formulas in Büchi Normal Form correspond to *Büchi regular transition systems (BRTS)*, defined in Section 8, which accept models of a formula. In this section, we show how to transform a formula in *LTL*(*MSO*) into an equivalent formula in Büchi Normal Form.

The idea of the construction is to generalize the standard translation of propositional temporal logic to Büchi Automata [38, 39] — the semantics of temporal operators is translated to additional state and transition information in the BRTS. In our case, temporal operators are translated to new configuration variables which represent the values of certain temporal subformulas. The semantics of temporal operators is maintained by constraints on the possible changes of the new configuration variables.

We assume, without loss of generality, that a formula ϕ is in negative normal form, i.e., that negations only occur in front of atomic formulas (as negations can always be "pushed" to the atomic formulas). Note that $\neg(\varphi\,\mathcal{W}\,\psi)$ equals $\neg\psi\,\mathcal{W}\,(\neg\varphi \wedge \neg\psi) \wedge \Diamond\neg\varphi$. Define a *core subformula* of ϕ as a subformula of ϕ which has a temporal operator as its main connective. We will denote by $\psi(i)$ a formula where i is the (possibly) only free variable of ψ. We introduce auxiliary variables to track the values of core subformulas of ϕ, as follows.

- For each core subformula $\psi(i)$ we introduce an auxiliary configuration variable x_ψ. Intuitively, the value of $x_\psi[i]$ represents the same value as $\psi(i)$ at each timepoint.
- For each core subformula of the form $\Diamond\,\psi_1(i)$ we introduce an auxiliary configuration variable $y_{\Diamond\psi_1}$ (called an *eventuality variable*). Intuitively, if the formula $y_{\Diamond\psi_1}[i]$ is true, then the formula $\psi_1(i)$ must be true at some future time point.

Here is the reason why our translation is only applicable to restricted *LTL*(*MSO*) formulas: since words are one-dimensional, it is not possible to use configuration variables to encode the value of subformulas with more than one free variable.

The value of any subformula ψ can be represented by an encoding $\langle\!\langle\psi\rangle\!\rangle$ into the extended set of configuration variables, together with constraints on the auxiliary variables. We first define the *encoding* $\langle\!\langle\psi\rangle\!\rangle$ of a formula ψ as follows. Note that the only change is to replace core subformulas by a corresponding auxiliary variable.

$$\begin{array}{lcll}
\langle\!\langle\psi\rangle\!\rangle & \triangleq & \psi & \text{for } \psi \text{ in MSO}\\
\langle\!\langle\psi_1 \wedge \psi_2\rangle\!\rangle & \triangleq & \langle\!\langle\psi_1\rangle\!\rangle \wedge \langle\!\langle\psi_2\rangle\!\rangle & \\
\langle\!\langle\psi_1 \vee \psi_2\rangle\!\rangle & \triangleq & \langle\!\langle\psi_1\rangle\!\rangle \vee \langle\!\langle\psi_2\rangle\!\rangle & \\
\langle\!\langle\exists i : \psi_1\rangle\!\rangle & \triangleq & \exists i : \langle\!\langle\psi_1\rangle\!\rangle & \\
\langle\!\langle\forall i : \psi_1\rangle\!\rangle & \triangleq & \forall i : \langle\!\langle\psi_1\rangle\!\rangle & \\
\langle\!\langle\exists I : \psi_1\rangle\!\rangle & \triangleq & \exists I : \langle\!\langle\psi_1\rangle\!\rangle & \\
\langle\!\langle\forall I : \psi_1\rangle\!\rangle & \triangleq & \forall I : \langle\!\langle\psi_1\rangle\!\rangle & \\
\langle\!\langle\Box\,\psi_1(i)\rangle\!\rangle & \triangleq & x_{\Box\psi_1}[i] & \\
\langle\!\langle\Diamond\,\psi_1(i)\rangle\!\rangle & \triangleq & x_{\Diamond\psi_1}[i] & \\
\langle\!\langle\psi_1(i)\,\mathcal{W}\,\psi_2(i)\rangle\!\rangle & \triangleq & x_{\psi_1\mathcal{W}\psi_2}[i] &
\end{array}$$

Let **localconstr**(ϕ) be the conjunction of a set of *local constraints* on the auxiliary variables of ϕ as defined below.

1. For each auxiliary variable x_ψ the corresponding local constraint is:

$$\forall i : \Big(x_{\Box\psi_1}[i] \leftrightarrow \Big[\langle\!\langle\psi_1(i)\rangle\!\rangle \wedge x'_{\Box\psi_1}[i]\Big]\Big)$$
when $\psi(i)$ is $\Box\,\psi_1(i)$,
$$\forall i : \Big(x_{\Diamond\psi_1}[i] \leftrightarrow \Big[\langle\!\langle\psi_1(i)\rangle\!\rangle \vee x'_{\Diamond\psi_1}[i]\Big]\Big)$$
when $\psi(i)$ is $\Diamond\,\psi_1(i)$, and
$$\forall i : \Big(x_{\psi_1\mathcal{W}\psi_2}[i] \leftrightarrow \Big[\langle\!\langle\psi_2(i)\rangle\!\rangle \vee \Big(\langle\!\langle\psi_1(i)\rangle\!\rangle \wedge x'_{\psi_1\mathcal{W}\psi_2}[i]\Big)\Big]\Big)$$
when $\psi(i)$ is $\psi_1(i)\,\mathcal{W}\,\psi_2(i)$.

2. Let $y_{\Diamond\psi_1}, \ldots, y_{\Diamond\psi_k}$ be the set of eventuality variables. We define their local constraint as follows.

$$\bigwedge_{m=1}^{k} \forall i : \left(\left[y_{\Diamond\psi_m}[i] \wedge \neg y'_{\Diamond\psi_m}[i]\right] \rightarrow \langle\!\langle\psi_m(i)\rangle\!\rangle\right)$$

Intuitively, whenever $y_{\Diamond\psi_m}[i]$ flips from true to false, it has "observed" that $\psi_m(i)$ was true in the previous state. Then we know that $\psi_m(i)$ was true at least once in the past.

We will require that all eventuality variables are false infinitely often and that they become true when appropriate. Let **evconstr**(ϕ) be the *eventuality constraint*, defined below.

$$\bigwedge_{m=1}^{k} \forall i : \left(\neg y_{\Diamond\psi_m}[i] \wedge \left[y'_{\Diamond\psi_m}[i] \leftrightarrow x'_{\Diamond\psi_m}[i]\right]\right)$$

Intuitively, that the eventuality variables are false means that they have witnessed the "eventuality" (that which should become true). The second constraint says that they should "reset" — i.e., they should check whether another eventuality should be witnessed, which is the case precisely when $x'_{\Diamond\psi_m}[i]$ is true.

Note that, in case some core subformula ψ does not have a free variable i, the local constraints encode the value of ψ on each of the positions i. This is correct, but perhaps not optimal.

We will transform a formula ϕ into the formula $\langle\!\langle\phi\rangle\!\rangle \wedge \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)$, which is clearly in Büchi Normal Form. The rest of this section will establish soundness of this transformation — meaning that a formula is satisfiable if and only if the transformed formula is satisfiable. The proof is done in two steps. The first lemma below states properties of the auxiliary variables, while the second proves soundness of the construction.

Lemma 1. *If* $(M, \mathit{Val}, t) \models \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)$, *then for all core subformulas* $\psi(i)$ *of* ϕ *we have*

1. $(M, \mathit{Val}, t) \models \forall i : (x_{\Box\psi_1}[i] \to \Box\langle\!\langle\psi_1(i)\rangle\!\rangle)$ *for* $\psi(i) = \Box\,\psi_1(i)$
2. $(M, \mathit{Val}, t) \models \forall i : (x_{\Diamond\psi_1}[i] \to \Diamond\langle\!\langle\psi_1(i)\rangle\!\rangle)$ *for* $\psi(i) = \Diamond\,\psi_1(i)$
3. $(M, \mathit{Val}, t) \models \forall i : (x_{\psi_1\mathcal{W}\psi_2}[i] \to \langle\!\langle\psi_1(i)\rangle\!\rangle\,\mathcal{W}\,\langle\!\langle\psi_2(i)\rangle\!\rangle)$ *for* $\psi(i) = \psi_1(i)\,\mathcal{W}\,\psi_2(i)$.

Proof.

1. Suppose $(M, \mathit{Val}', t) \models x_{\Box\psi_1}[i]$ for some valuation $\mathit{Val}' = \mathit{Val}[i \mapsto m]$. Since $(M, \mathit{Val}, t) \models \Box\,\mathbf{localconstr}(\phi)$ we have
$$(M, \mathit{Val}', t) \models \Box\left(x_{\Box\psi_1}[i] \leftrightarrow \left[\langle\!\langle\psi_1(i)\rangle\!\rangle \wedge x'_{\Box\psi_1}[i]\right]\right).$$
By induction, it follows that $(M, \mathit{Val}', t') \models \langle\!\langle\psi_1(i)\rangle\!\rangle$ for every $t' \geq t$ and thus $(M, \mathit{Val}', t) \models \Box\,\langle\!\langle\psi_1(i)\rangle\!\rangle$.
2. Suppose $(M, \mathit{Val}', t) \models x_{\Diamond\psi_1}[i]$ for some valuation $\mathit{Val}' = \mathit{Val}[i \mapsto m]$. Suppose that $(M, \mathit{Val}', t) \not\models \Diamond\langle\!\langle\psi_1(i)\rangle\!\rangle$. Then $(M, \mathit{Val}', t) \models \Box\neg\langle\!\langle\psi_1(i)\rangle\!\rangle$. Together with
$$(M, \mathit{Val}', t) \models \Box\left(x_{\Diamond\psi_1}[i] \leftrightarrow \left[\langle\!\langle\psi_1(i)\rangle\!\rangle \vee x'_{\Diamond\psi_1}[i]\right]\right)$$
from the local constraints, we therefore get
$$(M, \mathit{Val}', t) \models \Box\,x_{\Diamond\psi_1}[i] \quad .$$
The eventuality constraint gives
$$(M, \mathit{Val}', t') \models y'_{\Diamond\psi_1}[i] \leftrightarrow x'_{\Diamond\psi_1}[i], \text{ for some } t' \geq t\,.$$
Then it follows from $(M, \mathit{Val}', t) \models \Box\,x_{\Diamond\psi_1}[i]$ that
$$(M, \mathit{Val}', t') \models y'_{\Diamond\psi_1}[i]$$
and thus
$$(M, \mathit{Val}', t'+1) \models y_{\Diamond\psi_1}[i]\,.$$
Let $t'' > t'+1$ be the earliest point in time after t' (which has to exist because of the eventuality constraint) when
$$(M, \mathit{Val}', t'') \models \neg y_{\Diamond\psi_1}[i]\,.$$
But then since t'' was the earliest point in time we have
$$(M, \mathit{Val}', t''-1) \models y_{\Diamond\psi_1}[i] \wedge \neg y'_{\Diamond\psi_1}[i]$$
which together with the local constraint of $y_{\Diamond\psi}$ gives us
$$(M, \mathit{Val}', t''-1) \models \langle\!\langle\psi_1(i)\rangle\!\rangle\,.$$
Since $t''-1 > t' \geq t$ we conclude that
$$(M, \mathit{Val}', t) \models \Diamond\langle\!\langle\psi_1(i)\rangle\!\rangle$$
which contradicts the assumption.
3. Suppose $(M, \mathit{Val}', t) \models x_{\psi_1\mathcal{W}\psi_2}[i]$ for some valuation $\mathit{Val}' = \mathit{Val}[i \mapsto m]$. Since $(M, \mathit{Val}, t) \models \Box\,\mathbf{localconstr}(\phi)$ we have
$$(M, \mathit{Val}', t) \models \Box\begin{pmatrix} x_{\psi_1\mathcal{W}\psi_2}[i] \leftrightarrow \\ \left[\langle\!\langle\psi_2(i)\rangle\!\rangle \vee \left(\langle\!\langle\psi_1(i)\rangle\!\rangle \wedge x'_{\psi_1\mathcal{W}\psi_2}[i]\right)\right]\end{pmatrix}.$$
By induction on t it follows that either $(M, \mathit{Val}', t) \models \Box\,\langle\!\langle\psi_1(i)\rangle\!\rangle$, or that eventually for some $t' \geq t$ we have $(M, \mathit{Val}', t') \models \langle\!\langle\psi_2(i)\rangle\!\rangle$ before which we have $(M, \mathit{Val}', t'') \models \langle\!\langle\psi_1(i)\rangle\!\rangle$ for each $t'' : t \leq t'' < t'$. Hence $(M, \mathit{Val}', t) \models \langle\!\langle\psi_1(i)\rangle\!\rangle\,\mathcal{W}\,\langle\!\langle\psi_2(i)\rangle\!\rangle$ as desired. $\Box$

Lemma 2. *Let* ψ *be a subformula of* ϕ, *Val a valuation, and* t *a timepoint. There is a matrix* M *such that* $(M, \mathit{Val}, t) \models \psi$ *if and only if there is a matrix* M', *different from* M *only in the auxiliary variables of* ϕ, *such that* $(M', \mathit{Val}, t) \models \langle\!\langle\psi\rangle\!\rangle \wedge \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)$.

Proof. $\Longrightarrow$: Define M' to be the same as M (of width n) except for the auxiliary variables. We will show that the auxiliary variables can be set in M' so that $(M', \mathit{Val}, t) \models \langle\!\langle\psi\rangle\!\rangle \wedge \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)$.

- For each core subformula $\psi'(i)$ of ψ and for each $t'' \in \mathbf{N}$ and $m \in \mathbf{Z}_n$ let:
$$x_{\psi'} \in M'(t'', m) \iff (M, \mathit{Val}[i \mapsto m], t'') \models \psi'(i)\,. \qquad (\star)$$
- We show that there exists an infinite sequence of timepoints $(t_k)_{k\geq 0}$ with $t = t_0 < t_1 < \cdots$ such that for each $k > 1$ $(M', \mathit{Val}, t_k) \models \mathbf{evconstr}(\phi)$. For each such t_k and for each core subformula of ψ of the form $\Diamond\,\psi_1(i)$ and $m \in \mathbf{Z}_n$ we thus put:
  - $y_{\Diamond\psi_1} \notin M'(t_k, m)$, and
  - $y_{\Diamond\psi_1} \in M'(1+t_k, m) \iff x_{\Diamond\psi_1} \in M'(1+t_k, m)$. $(\star\star)$

  From t_k we find t_{k+1} by defining M' for t' with $t_k < t' \leq t_{k+1}$ inductively, as follows. The strategy we employ is to choose the timepoint t_{k+1} such that the values of each variable $y_{\Diamond\psi_1}$ are all false, i.e.:

– For each core subformula $\Diamond\,\psi_1(i)$ let

$$y_{\Diamond\psi_1} \in M'(t'+1,m) \\ \Longleftrightarrow \\ y_{\Diamond\psi_1} \in M'(t',m) \,\wedge\, (M,\, \mathit{Val}[i \mapsto m], t') \not\models \psi_1(i)\,.$$

– If for some earliest point in time $t' > t_k$ we for every core subformula $\Diamond\,\psi_1(i)$ and $m \in \mathbf{Z}_n$ have $y_{\Diamond\psi_1} \notin M'(t',m)$ then let $t_{k+1} = t'$.

Thus we allow the values of $y_{\Diamond\psi_1}$ between t_k and t_{k+1} to change from true to false, but not from false to true. Note that the eventuality variables satisfy their local constraints. Now we show that we can always find t_{k+1} from t_k . Suppose, in contrary, that we cannot. Then there is some core subformula $\Diamond\,\psi_1(i)$ and $m \in \mathbf{Z}_n$ such that:

$$y_{\Diamond\psi_1} \in M'(t',m) \text{ for all } \; t' > t_k\,.$$

Since the above holds in particular for $t' = 1 + t_k$ we have by $(\star\star)$ that $x_{\Diamond\psi_1} \in M'(1+t_k, m)$ and therefore by $(\star)$ we get $(M,\, \mathit{Val}[i \mapsto m], 1+t_k) \models \Diamond\,\psi_1(i)$. Then $(M,\, \mathit{Val}[i \mapsto m], t'') \models \psi_1(i)$ for some $t'' > t_k$ and thus our strategy described above gives $y_{\Diamond\psi_1} \notin M'(t''+1,m)$. This is a contradiction.

$\Longleftarrow$: We prove that
$(M',\, \mathit{Val}, t) \models \langle\!\langle \psi \rangle\!\rangle \wedge \Box\mathbf{localconstr}(\phi) \wedge \Box\Diamond\mathbf{evconstr}(\phi)$
implies $(M',\, \mathit{Val}, t) \models \psi$.

Let thus $M = M'$. We proceed by induction over the structure of ψ .

ψ in MSO : Since $\langle\!\langle \psi \rangle\!\rangle = \psi$, we get $(M,\, \mathit{Val}, t) \models \psi$.
$\psi = \psi_1 \vee \psi_2$: We get $(M,\, \mathit{Val}, t) \models \psi_1$ or $(M,\, \mathit{Val}, t) \models \psi_2$ by induction.
$\psi = \psi_1 \wedge \psi_2$: We get $(M,\, \mathit{Val}, t) \models \psi_1$ and $(M,\, \mathit{Val}, t) \models \psi_2$ by induction.
$\psi = \neg\psi_1$: Then ψ must be in MSO, since ψ is in negative normal form.
$\psi = \exists i : \psi_1$: We get $(M,\, \mathit{Val}, t) \models \exists i : \langle\!\langle \psi_1 \rangle\!\rangle$ and by the semantics

$$(M,\, \mathit{Val}[i \mapsto m], t) \models \langle\!\langle \psi_1 \rangle\!\rangle$$

for some $m \in \mathbf{Z}_n$. Hence $(M,\, \mathit{Val}, t) \models \langle\!\langle \psi_1(m) \rangle\!\rangle$. Since

$$\Box\mathbf{localconstr}(\phi) \,\wedge\, \Box\Diamond\mathbf{evconstr}(\phi)$$

is a closed formula, and thus does not depend on i, it follows that

$$(M,\, \mathit{Val}, t) \models \langle\!\langle \psi_1(m) \rangle\!\rangle \wedge \Box\mathbf{localconstr}(\phi) \wedge \Box\Diamond\mathbf{evconstr}(\phi)\,.$$

By the induction hypothesis we get $(M,\, \mathit{Val}, t) \models \psi_1(m)$ and by the semantics we obtain $(M,\, \mathit{Val}, t) \models \exists i : \psi_1(i)$.
$\psi \in \{\exists I : \psi_1, \forall i : \psi_1, \forall I : \psi_1\}$: Analogous with $\psi = \exists i : \psi_1$.
$\psi = \Box\psi_1(i)$: We get $(M,\, \mathit{Val}, t) \models x_{\Box\psi_1}[i]$. Hence $(M,\, \mathit{Val}, t) \models \Box\langle\!\langle \psi_1(i) \rangle\!\rangle$ by Lemma 1. This means that $(M,\, \mathit{Val}, t') \models \langle\!\langle \psi_1(i) \rangle\!\rangle$ for all $t' \geq t$. By induction we thus obtain $(M,\, \mathit{Val}, t') \models \psi_1(i)$ for all $t' \geq t$ which means that $(M,\, \mathit{Val}, t) \models \Box\psi_1(i)$.
$\psi = \Diamond\psi_1(i)$: Analogous with $\psi = \Box\psi_1(i)$.
$\psi = \psi_1(i)\,\mathcal{W}\,\psi_2(i)$: We get $(M,\, \mathit{Val}, t) \models x_{\psi_1\mathcal{W}\psi_2}[i]$. Hence by Lemma 1 we have $(M,\, \mathit{Val}, t) \models \langle\!\langle \psi_1(i) \rangle\!\rangle\,\mathcal{W}\,\langle\!\langle \psi_2(i) \rangle\!\rangle$. By the semantics and the induction hypothesis we thus obtain that either $(M,\, \mathit{Val}, t) \models \Box\,\psi_1(i)$, or that eventually for some $t' \geq t$ we have $(M,\, \mathit{Val}, t') \models \psi_2(i)$ before which $(M,\, \mathit{Val}, t'') \models \psi_1(i)$ for each $t'' : t \leq t'' < t'$. Thus $(M,\, \mathit{Val}, t) \models \psi_1(i)\,\mathcal{W}\,\psi_2(i)$. □

We are now ready to prove the main theorem.

Theorem 1. *For any* restricted formula ϕ *there exists a formula $BNF(\phi)$ in Büchi Normal Form such that*

$$M \models \phi \textit{ for some matrix } M \\ \textit{if and only if} \\ M' \models BNF(\phi) \textit{ for some matrix } M'\,.$$

Proof. The following formula is in Büchi Normal Form:

$$BNF(\phi) \;=\; \langle\!\langle \phi \rangle\!\rangle \wedge \Box\,\mathbf{localconstr}(\phi) \wedge \Box\Diamond\,\mathbf{evconstr}(\phi)\,.$$

It follows from Lemma 2 that there is a matrix M such that $M \models \phi$ if and only if there is a matrix M' such that $M' \models BNF(\phi)$. □

8 Verification

As shown in Section 4.3, to verify that a property holds for a system, we search for models of a formula that is a conjunction of the formula describing the system and the negation of the property. If no such models exist, the property holds. Models of the formula are counterexamples that explain why the property does not hold. Thus, the verification task is to find models of formulas.

To search for models of formulas, we use *Büchi regular transition systems*, defined below. They play the role of Büchi automata in the automata-theoretic approach but for $LTL(MSO)$ instead of LTL. A Büchi regular transition system is an automaton whose states are words and where the transition relation is represented using a regular set. We say that a length-preserving relation $\mathcal{R}$ on Σ^* is *regular* if the set (of words over $\Sigma \times \Sigma$)

$$(w(1), w'(1))(w(2), w'(2)) \cdots (w(n), w'(n))$$

such that $(w, w') \in \mathcal{T}$ is regular. The transition relation of a BRTS is given by such a regular length-preserving relation, which can also be described by a finite-state transducer — a finite-state automaton over pairs of words.

Definition 2. (**Büchi Regular Transition System**) A *Büchi regular transition system (BRTS)* over an alphabet Σ is a tuple $(\Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F})$ where

- $\mathcal{I} \subseteq \Sigma^*$ is a regular set of words over Σ called the set of *initial configurations*,
- $\mathcal{T} \subseteq \Sigma^* \times \Sigma^*$ is a regular length-preserving relation on words over Σ, called the *transition relation*, and
- $\mathcal{F} \subseteq \Sigma^* \times \Sigma^*$ is a regular length-preserving relation on words over Σ, called the set of *final transitions.*

An *accepting run* of $(\Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F})$ is a matrix M such that

- $M(0) \in \mathcal{I}$,
- $(M(i), M(i+1)) \in \mathcal{T}$ for any $i \geq 0$, and
- $(M(i), M(i+1)) \in \mathcal{F}$ for infinitely many i. $\square$

In the previous section, we showed how to translate a formula in *LTL*(*MSO*) into an equivalent formula in Büchi Normal Form. This form is characterized by BRTS.

Theorem 2. *For every formula φ in Büchi Normal Form, there is a Büchi regular transition system $(\Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F})$ such that, for every matrix M, we have $M \models \varphi$ if and only if M is an accepting run of $(\Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F})$.*

Proof. Let φ be in Büchi Normal Form

$$\phi_I \wedge \Box\phi_T \wedge \Box\Diamond\phi_F$$

and $(\Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F})$ be the BRTS such that for all matrices M:

$$\begin{array}{llcll} - & M(0) \in \mathcal{I} & \iff & M \models \phi_I & \text{and} \\ - & (M(0), M(1)) \in \mathcal{T} & \iff & M \models \phi_T & \text{and} \\ - & (M(0), M(1)) \in \mathcal{F} & \iff & M \models \phi_F \,. & \end{array}$$

The BRTS $(\Sigma^*, \mathcal{I}, \mathcal{T}, \mathcal{F})$ exists because ϕ_I, ϕ_T, ϕ_F are formulas in MSO, and thus can be translated into finite-state automata. $\square$

Just like in the automata-theoretic approach, checking models of a formula thus reduces into checking for accepting runs of a BRTS. Since the transition relation of a BRTS is length-preserving, the existence of an accepting run can be checked by searching for a reachable loop which contains an accepting state. Unlike the automata-theoretic approach, however, the set of states of a BRTS is infinite, requiring new techniques for finding accepting runs.

The procedure we use for finding accepting runs can in principle be described as follows. First, the set of reachable states is computed as $Inv = \mathcal{I} \circ \mathcal{T}^*$. Secondly, loops are found by identifying identical pairs in $(\mathcal{F} \cap \mathcal{T} \cap (Inv \times Inv)) \circ \mathcal{T}^*$. Thus, the problem reduces to computing transitive closures and reachability sets.

We have verified safety properties with our tool for regular model checking with techniques for computing transitive closures and reachability sets from [8,2], as well as liveness properties for some of the examples. Execution times are given in Table 1.

	Safety	Liveness
Token Pass	5.5	16.0
Token Ring	8.4	9.8
Bakery	13.9	44.2
Burns	39.6	
Szymanski	34.3	
Dijkstra	36.4	
Termination Detection	38.0	
Alternating Bit	179.2	
Sliding Window	1687.2	

Table 1. Experimental Results: Running times (in seconds) for verifying safety and liveness properties of the models in the paper

9 Conclusions

We have presented the logic *LTL*(*MSO*) for specifying properties of a class of parameterized and infinite-state systems, whose state vector can be modeled as a finite word of arbitrary length. By a sequence of modeling examples, we showed how this logic can be used to model and specify different types of protocols. We presented a technique for verifying that a system model satisfies a specification, where both the model and the specification are formulated in *LTL*(*MSO*). This technique is a natural extension of the automata-theoretic approach for finite-state model checking [38,39], and reduces the verification problem to checking whether a Büchi regular transition system has some accepting runs. In general, this problem is undecidable, but decidability results for certain classes have been obtained [22]. We have implemented techniques for checking whether BRTS have accepting runs, which work well on a number of examples.

References

1. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). In Alur, Peled (eds.) *Proc. CAV'04,* 16th *Int. Conf. on Computer Aided Verification*, LNCS 3114, pp. 348–360. Springer (2004)
2. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular model checking made simple and efficient. In Brim, Jancar, Kretínský, Kucera (eds.) *Proc. CONCUR 2002,* 13th *Int. Conf. on Concurrency Theory*, LNCS 2421, pp. 116–130. Springer (2002)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Algorithmic improvements in regular model checking. In Hunt, Somenzi (eds.) *Proc. CAV'03,* 15th *Int. Conf. on Computer Aided Verification*, LNCS 2725, pp. 236-248. Springer (2003)
4. Bartlett, K., Scantlebury, R., Wilkinson, P.: A note on reliable full-duplex transmissions over half duplex lines. *Communications of the ACM*, 2(5):260–261 (1969)
5. Baukus, K., Lakhnech, Y., Stahl, K.: Verification of parameterized networks. *Journal of Universal Computer Science*, 7(2):141–158 (2001)
6. Boigelot, B., Franois, J.-M., Latour, L.: The Liège automata-based symbolic handler (LASH). `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/` (2011)

7. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In Hunt, Somenzi (eds.) *Proc. CAV'03,* 15^{th} *Int. Conf. on Computer Aided Verification*, LNCS 2725, pp. 223–235. Springer (2003)
8. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In Emerson, Sistla (eds.) *Proc. CAV'00,* 12^{th} *Int. Conf. on Computer Aided Verification*, LNCS 1855, pp. 403–418. Springer (2000)
9. Bouajjani, A., Legay, A., Wolper, P.: Handling liveness properties in (ω-)regular model checking. *Electr. Notes Theor. Comp. Sci.*, 138(3):101–115 (2005)
10. Delzanno, G.: Automatic verification of cache coherence protocols. In Emerson, Sistla (eds.) *Proc. CAV'00,* 12^{th} *Int. Conf. on Computer Aided Verification*, LNCS 1855, pp. 53–68. Springer (2000)
11. Dijkstra, E.W., Feijen, W.H.J., van Gasteren, A.J.M.: Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219 (1983)
12. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In McAllester (ed.) *Proc. CADE-17, 17th International Conference on Automated Deduction*, LNCS 1831, pp. 236–254. Springer (2000)
13. Emerson, E.A., Kahlon, V.: Rapid parameterized model checking of snoopy cache coherence protocols. In Garavel, Hatcliff (eds.) *Proc. TACAS'03,* 9^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2619, pp. 144–159. Springer (2003)
14. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In *Proc.* 22^{th} *ACM Symp. on Principles of Programming Languages*, pp. 85–94 (1995)
15. Esparza, J., Kucera, A., Mayr, R.: Model-checking LTL with regular valuations for pushdown systems. In Kobayashi, Pierce (eds.) *Proc. TACS2001, 4th Int. Conf. on Theoretical Aspects of Computer Software*, LNCS 2215, pp. 316–339. Springer (2001)
16. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with invisible ranking. *Software Tools for Technology Transfer*, 8(3):261–279 (2006)
17. Fisman, D., Kupferman, O., Lustig, Y.: On verifying fault tolerance of distributed protocols. In Ramakrishnan, Rehof (eds.) *Proc. TACAS'08,* 14^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 4963, pp. 315–331. Springer (2008)
18. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735 (1992)
19. Giammarresi, D., Restivo, A.: Two-dimensional languages. In Salomaa, A., Rozenberg, G. (eds.) *Handbook of Formal Languages*, volume 3, Beyond Words, pp. 215–267. Springer (1997)
20. Gribomont, E.P., Zenner, G.: Automated verification of Szymanski's algorithm. In Steffen, editor, *Proc. TACAS'98,* 4^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pp. 424–438. Springer Verlag, 1998.
21. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In Brinksma, Cleaveland, Larsen, Margaria, Steffen (eds.) *Proc. TACAS'95,* 1^{st} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1019, pp. 89–110. Springer (1995)
22. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In Graf, Schwartzbach (eds.) *Proc. TACAS'00,* 6^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1785, pp. 220–234. Springer (2000)
23. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. *Theor. Comp. Sci.*, 256:93–112 (2001)
24. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455 (1974)
25. Lamport, L.: The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923 (1994)
26. Lynch, N.: *Distributed Algorithms.* Morgan Kaufmann (1996)
27. Manna, Z., Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems.* Springer (1992)
28. Nilsson, M.: *Regular Model Checking.* PhD thesis, Uppsala University (2005)
29. Pnueli, A.: The temporal logic of programs. In *Proc.* 18^{th} *Annual Symp. Foundations of Computer Science*, pp. 46–57. (1977)
30. Pnueli, A.: The temporal semantics of concurrent programs. *Theor. Comp. Sci.*, 13:45–60 (1982)
31. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In Margaria, Yi (eds.) *Proc. TACAS'01,* 7^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2031, pp. 82–97. Springer (2001)
32. Pnueli, A., Shahar, E.: Liveness and acceleration in parameterized verification. In Emerson, Sistla (eds.) *Proc. CAV'00,* 12^{th} *Int. Conf. on Computer Aided Verification*, LNCS 1855, pp. 328–343. Springer (2000)
33. Pnueli, A., Xu, J., Zuck, L.: Liveness with (0, 1, ∞)-counter abstraction. In Brinskma, Larsen (eds.) *Proc. CAV'02,* 14^{th} *Int. Conf. on Computer Aided Verification*, LNCS 2404, pp. 107–122. Springer (2002)
34. Sistla, P.A.: Parametrized verification of linear networks using automata as invariants. In O. Grumberg, editor, *Proc. CAV'97,* 9^{th} *Int. Conf. on Computer Aided Verification*, LNCS 1254, pp. 412–423, Springer (1997)
35. Szymanski, B.K.: Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conf. on Information Technology*, pp. 110–117. IEEE Computer Society Press (1990)
36. Tannenbaum, A.S.: *Computer Networks.* Prentice-Hall (1996)
37. Thomas, W.: Automata on infinite objects. In van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pp. 133–191. Elsevier, Amsterdam (1990)
38. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In *Proc. LICS'86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pp. 332–344. (1986)
39. Vardi, M.Y.: Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1–2):79–98 (1991)
40. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In Hu, Vardi (eds.) *Proc. CAV'98,* 10^{th} *Int. Conf. on Computer Aided Verification*, LNCS 1427, pp. 88–97. Springer (1998)