Model Checking Self-Stabilising in Embedded Systems with Linear Temporal Logic

Rim Marah, and Abdelaaziz EL Hibaoui
Faculty of Science
Abdelmalek Essaâdi University,
P.O. Box 2121, Tetuan, Morroco

***Abstract*—Over the past two decades, the use of distributed embedded systems is wide in many applications. One way to guarantee that these systems tolerate transient faults is done by making them self-stabilizing systems, which automatically recover from any transient fault.**
In this paper we present a formalism of self-stabilization concept based on Linear Temporal Logic (LTL), and model checked the self-stabilization in embedded systems. Using a case study inspired by industrial practice, we present in detail a model checking to verify the self stabilization property of our embedded system.

Keywords*—*Distributed Embedded Systems, Linear Temporal Logic, Self-Stabilization, Model Checking, Verification

I. Introduction

A general-purpose definition of embedded systems is that they are devices used to control, monitor or assist the operation of equipment, machinery or plant. Embedded reflects the fact that they are an integral part of the system. In many cases, their embeddedness may be such that their presence is far from obvious to the casual observe [1]. There are cases in which reliability is the most important requirement of those systems. To guarantee that these systems tolerate transient faults, we should make them self-stabilizing systems. This type of fault-tolerance is desirable in many distributed embedded systems [2], [3]. Verifying the correctness of those systems is a challenging task while testing them is intractable.

To make a rigorous verification of these systems, properties should be described in a precise and unambiguous manner. This is typically done by using properties specification language. There is several variety of different logics, according to the types of properties that they can express. In particular, we will focus on the use of LTL (Linear Temporal Logic) [4] [5], as a property specification language. To verify that a system meets its specification, we use the connection between temporal logic and the automata theoretical approach to model checking [5] [6]. The latter applies the intimate relationship between LTL and automata on infinite words. In [7] it was first proven that the set of infinite words, defined by LTL formula, can be accepted by some automaton on infinite words. Several procedures have been suggested which construct a generalized Bchi automaton that recognizes all models of a LTL formula [8] [9] [10].

Within computer science, the LTL has achieved a significant role in formal specification and verification of concurrent reactive systems. It is a very powerful specification mechanism, since it allows the expression of complex requirements through simple formulas. Actually it is widely used for verification of software systems [11] [12]. It provides a formal specification mechanism allowing the quantitative definition of the desired behaver of a systems. It makes it possible to succinctly express complex objectives due to its similarity to natural language.

The model checking is an automatic technique for verifying correctness properties of reactive systems, the two feathers that made the model checking [7] [12], so popular are that it can easily be automated and that is often able to produce a counterexample when the system can not meets its specifications. However the applicability of model checking is limited by the problem of the state space explosion. But luckily, the model checking on the fly is a remedy to this problem.

The linear temporal logic can be appropriate way to formalize the definition of self stabilisation of the systems. The Self-stabilizing systems witch were first introduced by Dijkstra in 1974 [13] [14] are the systems that can start in any global configuration and achieve behaviour meeting the task specification by them self.

Our main contribution in this paper is a proposition of an LTL formalism of the self stabilization concept, and model checking the embedded systems in order to verify there self-stabilization, based on the different phases of model checking Process:

i. Modelling the system: We will model our embedded system. For this modelling, we use the transition system and KRIPKE stricture that we will define properly later.

ii. Specification of the property to check: In this step, we will present our formalism of the self-stabilization concept. Basing on the LTL language, we will check the specification of our property.

iii. Using an algorithm or method to check whether the property is satisfied by the model: For this step we choose the model checking as a method of verification.

This paper is organized as follows: After the introduction, we define the tools used in modelling our system and we describe our model, in the second section called modelling the system. In the section tree named the specifying of the property, we put on the notation used in this paper and we

present our formalism of the self stabilization based on the LTL logic. In the forth section named algorithm of verification, we describe in details the steps of model checking algorithm that we use to verify our formalism. Finally in the conclusion, we conclude and give a future extension of our work.

II. System Modelisation

The semantic framework for algorithmic verification systems is given by transition systems and Kripke structures. These later and automata are used to model the reactive systems. They must then validate the model by determining if it satisfies the required properties of the systems. The system of ownership is expressed in terms of its states, transitions or paths.

In this part we are going to give the formal definitions of transition systems and Kripke stricture used to describe our system and define in details our model.

A. Background

Transition systems Transitions systems define the possible states of a system, its initial states and transitions. They provide a framework for describing the operating semantics for reactive system. To describe the behaviour of systems, we can model them by transition systems with are a digraphs where nodes represent states, and edges model transitions.

A transition system TS is a tuple (S, S_0, R) where

- S is a set of states.
- S_0 is a set of initial states.
- $R \subseteq S \times S$ is a transition relation.

To define Kripke stricture we use the usual definition of transitions systems and extended it by adding a labelling of states with atomic propositions.

Definition AP is a set of atomic propositions. A Kripke structure on AP is $M = (S, S_0, R, L)$ where:

- (S, S_0, R) is a transition system.
- $L : S \longrightarrow 2^{AP}$ is a function that label each state $s \in S$ the set $L(s)$ of atomic propositions true in s.

A path π of a M is a path of transitions system associated.

The trace σ of π is ω-word $L(\pi) = L(s_0)L(s_1)...$ in the alphabet $\sum = 2^{AP}$

1) Our Model: Our model is a parts conveying robot presented with finite state model. It's as a case study of embedded systems. This example is presenting in [15]. The robot has 3 devices: An inlet device parts called Dp, a workpiece transport device, called Td which is an arm provided with a clamp, and finally a workpiece removal device called Rd, where Td transports parts arriving on Dp. At a certain level of abstraction, the system is defined by three principles operations that can be stated as:

1) Td transport device files the piece on the discharge device if De is free.
2) Td transport device can mount just if it is taking up a devise.
3) Td transport device can get off just if it is empty.

For simplicity, we ignore Dp and will only interested in the introduction of the devise in Td and its release on De. We will not take in consideration operations (opening, closing, etc.) of the clamp, we will essentially look at the transport parts. All the developments of the system, in this operation mode is represented by the graph in bellow: The drawn

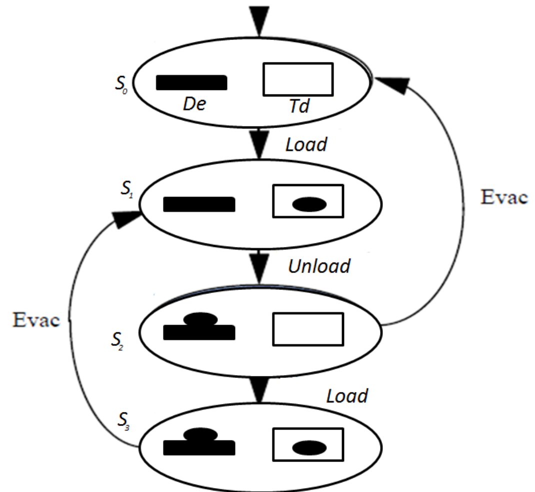

Fig. 1: Kripke Stricture of parts conveying robot

graph is called reachability graph. It represents all possible executions of the system. These are infinite sequences of couples (state transition) represented by a finite graph. As there is no distinction of terminal states, we say that this structure is a transition system. The states are decorated by the values of variables De and Td, where 'free' and 'occupied' values are represented schematically by the presence or absence of a part shown by a gray rectangle. TS with decorated states is called Kripke Structure. It is the The formal model that represents all executions.

III. Property Specification

Temporal logic was originally developed in order to represent tense in natural language. Temporal logic extends propositional or predicate logic by modalities that permit to referral to the infinite behaviour of a reactive system. They provide a very intuitive and mathematical precise notation for expressing properties about the relation between the state labels in executions. Temporal logic allows for the specification of the relative order of events. The linear temporal logic has achieved its popularity from the number of useful concepts that can formally and concisely be specified by using it.

A. Linear Temporal Logic

The linear temporal logic extends classical logic by temporal modalities. Its formulas are interpreted on infinite sequences of states such as executions of a Kripke structure. Before introducing LTL in more detail, LTL may be used to express the timing for the class of synchronous systems in which all components proceed in a lock-step fashion. In this setting, a transition corresponds to the advance of a single

time-unit. The underlying time domain is thus discrete, i.e., the present moment refers to the current state, and the next moment corresponds to the immediate successor state.

This subsection describes the syntactic rules according to which formulae in LTL can be constructed. The basic ingredients of LTL-formulae are atomic propositions. The Boolean connectors like conjunction $\wedge$, and negation $\neg$, and two basic temporal modalities $\bigcirc$ (pronounced next) and $\cup$ (ponounced until). The elementary temporal modalities that are present in most temporal logics include the operators:

- $\Diamond$ eventually (eventually in the future).
- $\Box$ always (now and forever in the future).

1) LTL Syntax: LTL formulae over the set AP of atomic proposition are formed according to the following grammar:

$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2$

where $a \in AP$.

We mostly abstain from explicitly indicating the set AP of propositions as this follows either from the context or can be defined as the set of atomic propositions occurring in the LTL formula at hand.

The precedence order on the operators is as follows: The unary operators bind stronger than the binary ones. $\neg$ and $\Box$ bind equally strong. The temporal operator $\cup$ takes precedence over $\wedge, \vee$ and $\rightarrow$ Parentheses are omitted whenever appropriate.

LTL formulae stand for properties of paths or in fact their trace. A path can either fulfil an LTL-formula or not. To precisely formulate when a path satisfies an LTL formula, we can follow this steps: First, the semantics of LTL formula φ is defined as a language $Words(\varphi)$ that contains all infinite words over the alphabet 2^{AP} satisfy φ. That is, to every LTL formula a single LT property is associated. Then, the semantics is extended to an interpretation over paths and states of a transition system.

2) LTL Semantic: Let φ be an LTL formula over AP. The LT property induced by φ is: $words(\varphi) = \sigma \in (2^{AP})^\omega \mid \sigma \models \varphi$ where the satisfaction relation $\models \subseteq (2^{AP})^\omega \times LTL$ is the smallest relation with the properties.

Here, for $\sigma = A_0 A_1 A_2 \cdots \in (2^{AP})^\omega, \sigma[j \ldots] = A_j A_{j+1} A_{j+2} \ldots$ is the suffix of σ starting in the (j+1)st symbol A_j .

- $\varphi \models true$
- $\varphi \models a$ iff $a \in A_0 (i.e, A_0 \models a)$
- $\varphi \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ et $\sigma \models \varphi_2$
- $\varphi \models \neg\varphi$ iff $\sigma \not\models \varphi$
- $\varphi \models \bigcirc\varphi$ iff $\sigma \in [1 \ldots] = A_1 A_2 A_3 \ldots \models \varphi$
- $\varphi \models \varphi_1 \cup \varphi_2$ iff $\exists j \geqslant 0$ such that $\sigma[j \ldots] \models \varphi_2$ and $\sigma[i \ldots] \models \varphi_1$, for all $0 \leqslant i < j$

Essentially, temporal logic extends classical propositional logic with a set of temporal operators that navigate between worlds using this accessibility relation. Typical temporal operators used in LTL are:

- $\bigcirc p$: p is true in the next moment in time
- $\Box p$: p is true in all future moment
- $\Diamond p$: p is true in some future moment
- $p \cup q$: p is true until q is true

B. Self-Stabilization

The idea of self-stabilization in distributed computing was first proposed by Dijkstra in 1974 [13]. The concept of self-stabilization is that, regardless of its initial state, the system is guaranteed to converge to a legitimate state in a bounded amount of time by itself and without any outside intervention.

The self-stabilization principle applies to any system built on a significant number of components which are evolving independently from one another, but which are cooperating or competing to achieve common goals. This applies, in particular, to large distributed systems which tend to result from the integration of many subsystems and components developed separately at earlier times or by different people.

1) Formal Definition: Arora and Gouda [16] introduced a more generalized definition of self-stabilization, called stabilization, which is defined as follows:

The definition of stabilization for a system S with respect to two predicates P and Q, over its set of global states. Predicate Q denotes a restricted start condition. S satisfies $Q \longrightarrow P$(read as Q stabilizes to P) if it satisfies the following two properties:

i. Closure: P is closed under the execution of S. That is, once P is established in S, it cannot be falsified.

ii. Convergence: If S starts from any global state that satisfies Q, then S is guaranteed to reach a global state satisfying P within a finite number of state transitions.

The self-stabilization is a special case of stabilization where Q is always true, that is, if S is self-stabilizing with respect to P, then this may be restated as $TRUE \longrightarrow P$ in S.

2) Self-Stabilization Formalism: Based on the definition above, we propose a formalism of the self-stabilization. The advantage that this definition has among the other versions, is that it uses the predicate Q and P, thing that makes the use of LTL logic easier. To formalize the self stabilization using the LTL logic, we should do it for its two properties: closure and Convergence.

Let P and Q be a predicates of S, and $\sigma = [s_1 s_2 s_3 ....]$ is an execution of S, the $\models$, $\Box$ and $\Diamond$ are defined in section: semantic of LTL.

The semantic of the closure is provided by the definition: Once S is in a legitimate state P, it will stay in that legitimate state. Formally, we interpret it as follow:

- $s \models \Diamond\Box p$

For the convergence, it can be defined as: From any arbitrary state that satisfy Q, S is guaranteed to reach a configuration satisfying P, in a finite number of state transitions. This can be translated to LTL language as follow: $((\sigma, i) \models Q) \Longrightarrow (\Diamond((\sigma, k)_{i \leq k < \propto} \models P))$ witch means that: $\forall i, \exists j$ such that $i \leqslant j < \infty$ $(s_i \models Q) \Longrightarrow \Diamond(s_j \models P)$

And we have
$S \models \phi \iff \sigma \models \phi(\forall\sigma \in Exc(S))$ and $\sigma \models \phi \iff s_i \models \phi(\forall s_i \in \sigma)$
Hence we can write
$(S \models Q) \implies \lozenge(S \models P)$
Taking in consideration that $(a \implies b) \iff (\neg a$ ou $b)$ we can formalise the convergence as follow:
$(S \not\models Q) \vee (\lozenge(S \models P))$

IV. Verification Algorithm

In order to test the validity of this both LTL formula, we should follow one of the verification methods. From the variety of methods that exist in literature, we chose the verification of model checking.

The model checking is a verification technique that explores all possible system states in a brute-force manner. It is interested by the determination of whether a property φ is verified by the system M as mention this figure:

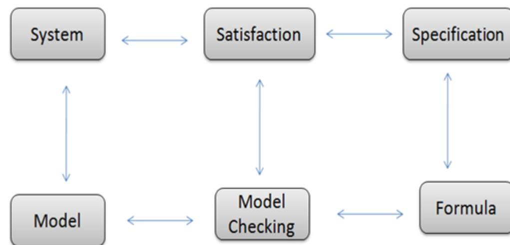

Fig. 2: Modelization with Model Checking

In this way, it can be a good method of verification of our self stabilization formalism, since it refers to the question of whether a formula is true in an interpretation, denoted $M \models \varphi$. Where a Kripke structure M can be a Petri net or a computer system, and the formula φ specifies our formalism witch is a property of system.

A. Model Checking

As principles, the model checking is an automated technique that, given a finite-state model of a system and a formal property, it checks systematically whether this property holds for that model.

A different phases can be distinguished In applying model checking Process:
Modeling phase: We model the system under consideration using the model description language of the model checker. Formalization phase: We formalize the property to be checked using the property specification language. Running phase: We run the model checker to check the validity of the property in the system model.

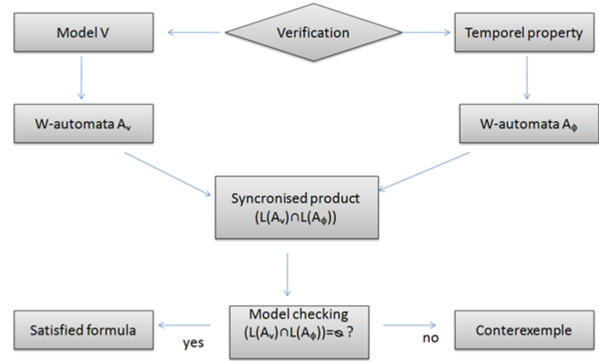

Fig. 3: The principle of model checking

In addition to these steps, the entire verification should be planned, administered, and well organized as shown in this algorithm:

- Input: KRIPKE stricture M, and LTL formula φ.
- Issue: If $M \models \varphi$
- Steps:
  1) Transformation of Kripke stricture M to Buchi automata A_M.
  2) Transformation of the LTL formula φ to Buchi automata A_φ.
  3) Test whether $L(A_M) \subseteq L(A_\varphi) \iff L(A_M) \cap L(A_\varphi)^c$

This can be formally written as: Input: finite transition system TS and LTL formula φ. Output: yes if TS$\models \varphi$; otherwise, no plus a counterexample. Transforming φ to GBA: Construct an NBA $A_{\neg\varphi}$ such that $\iota_\omega(A_{\neg\varphi}) = Words(\neg\varphi)$. Construct the product transition system $TS \otimes A$ if there exists a path π in $TS \otimes A$ satisfying the accepting condition of A then return no and an expressive prefix of π else return yes End if.

We presente the concept of model checking in somewhat in the diagram bellow: Overview of LTL model checking

B. Running Phase

In this subsection we will respect the model checking algorithm, and implement explicitly each step in more details. In the execution of model checking algorithm, the most difficult phase, is the second phase which is the transformation of the LTL formula φ to a Buchi automaton A_φ. For that reason, we will make such a big deal about this step of algorithm.

1) φ to BA Transformation: This stage of model checking algorithm is known of its difficulty. There are several ways to realize it[ref]. We are going to choose the method of Tables. Based on the following equivalences:
$\varphi \cup \varphi_1 \equiv \varphi_1 \vee (\varphi \wedge X(\varphi \cup \varphi_1))$.
$\varphi R \varphi - 1 \equiv (\varphi \wedge \varphi - 1) \vee (\varphi - 1 \wedge X(\varphi \wedge \varphi - 1))$.

And considering a Z-shaped set of negative normal formulas, Z is reduced if:
For all $z \in Z$, z is of the form p, $\neg p$ or $X(z')$.
We obtain the follow reduction of temporal connectors:

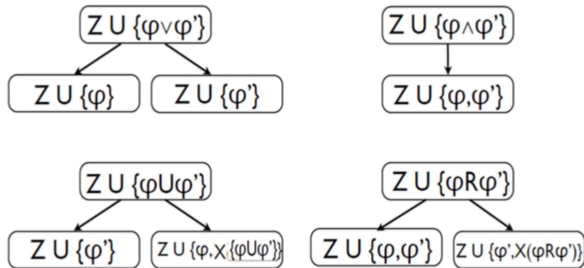

Fig. 4: Reduction of temporal conectors

This method involves, the reduction of the LTL formula φ to the normale negative form, the reduction of the temporal connectors, and finally the transformation to a Buchi automata.

From above, we obtain our Buchi automaton of closure and convergence properties:

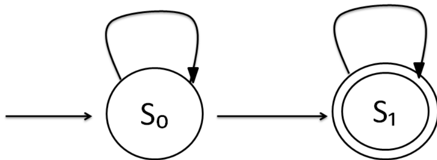

Fig. 5: Buchi automata for closure

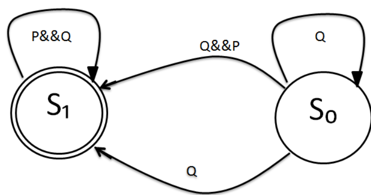

Fig. 6: Buchi automata for convergence

2) Kripk Stricture to BA Transformation : From the Kripke structure of our model, we can is obtain the follow Buchi automata that it correspond:

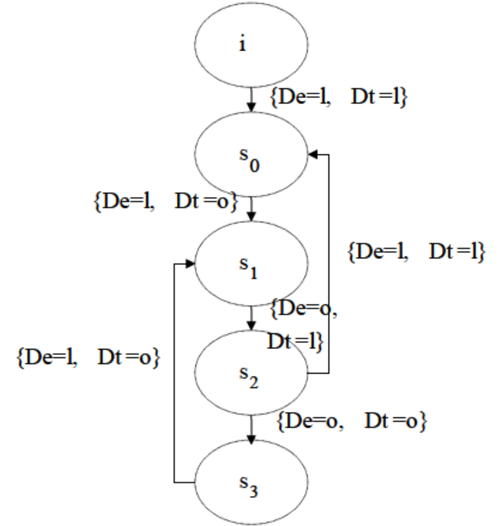

Fig. 7: Buchi automata for convergence

This automaton is obtained by performing three transformations:

1) Introduction of the initial condition i connected by a transition to the initial state of the Kripke structure.
2) Replacing Labels transitions by the decor of their target state.
3) Make all statements as acceptance states.

3) Buchi Product Checking : The model-checking is reduced to a problem in automata theory, since finite-state reactive programs can be represented quite naturally as Buchi automata [17]. A Buchi automaton is a non-deterministic finite-state automaton taking infinite words as input. A word is accepted if the automaton goes through some designated good states infinitely often while reading it. In this level, we are going to make the synchronized product of both Buchi automata, formula's automata and model's automata. The product of model automata and formula's automata is given by : .

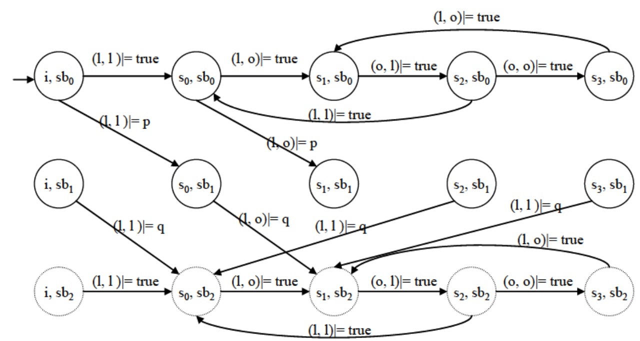

Fig. 8: Product of Buchi automata for closure

With the description of language of infinite strings with an automaton having finitely many states, the infinite string must

make a cycle though some of the states of the automaton. For the string to be accepted, it must also be possible to reach this cycle from the start state. We thus process the product automaton by first finding the set of states that are reachable from the start state, and then checking whether any of them is in a cycle. This can be made more efficient by checking for cycles as each state becomes reachable. If a reachable cycle is found, then there is some string in the intersection of the language of the model and the language of the negation of the desired formula. Thus, the desired formula is not valid in the model, and the found string is a counterexample. On the other hand, if there is no reachable state that is part of a cycle, then the language is empty and the original formula is valid in the model.

I our case, the existence of these acceptors cycles indicates that the property of closure is not satisfied, therefore our system is not self stabilized.

V. Conclusion

In summary, we model checked our embedded system to verify LTL self stabilization formalism requiring the following steps:

1) Model: view it as a Buchi automaton, with all states as final states.
2) Formula: Negate the formula of self stabilization and perform the following conversion steps to obtain Buchi automaton $A_{\neg\varphi}$
    - Convert the formula to normal form (essentially, push negations down to atoms).
    - Convert the normal form to a graph.
    - Convert the graph to a generalized Buchi automaton (essentially identify the final states and appropriate state labelling).
    - Convert the generalized Buchi automaton to a Buchi automaton (essentially convert the set of sets of final states to a single set of final states, by duplicating the automaton)
3) Compute the product of the two automata, which accepts the intersection of the two languages.
4) Determine whether the product automaton accepts any strings. we noticed that closure formulas is not satisfied in our system, then our robot is not self-stabilizing system.

References

[1] D. Estrin, R. Govindan, and J. Heideman, “Embedding the internet.” in *Communications of the ACM*, May 2000.

[2] Whittlesey-Harris, R. S., and M. Nesterenko, “Fault-tolerance verification of the fluids and combustion facility of the international space station,” in *Distributed Computing Systems*, 2006.

[3] M. Arumugam, , and Kulkarni, “Self-stabilizing deterministic tdma for sensor networks,” in *Distributed Computing and Internet Technology.*, 2005.

[4] D.Gabbay, A.Pnneli, S.Shelah, and J.Stavi, “The temporal analysys of fairness,” in *Princ. of Prog. Lang.*, 1980, pp. 163– 173.

[5] A. Pnuelii, “The temporal logic of programs,” in *The temporal logic .*, 1977, pp. 46–57.

[6] E. Clarke, O. Grumberg, and D. Peled, “Model checking,” *MIT Press*, 1999.

[7] R. Gerth, D. Peled, M. Vardi, , and P. Wolper, “Simple on-the-fly automatic verification of linear temporal logic,” in *Specification, Testing, and Verification.*, June 1995.

[8] F. Somenzio and R. Bloem, “Efficient bchi automata from ltl formula,” *Springer-Verlag*, 2000.

[9] R. Sherman, A. Pnueli, and D. Harel, “the interesting part of process logic,” *SIAM Journal on Computing*, 1984.

[10] J.-M. Couvreur, “On-the-fly verification of linear temporal logic,” in *Formal Methods in the Development of Computing Systems.*, 1999.

[11] Z. Manna and A. Pnueli, “The temporal logic of reactive and concurrent systems,” *Springer-Verlag*, 1992.

[12] E. M. M. Clarke, D. Peled, and O. Grumberg, “Model checking,” *MIT Press*, 1999.

[13] E. Dijkstra, “Self-stabilizing systems in spite of distributed control,” 1974.

[14] M. J. Fischer and H. Jiang, “Self-stabilizing leader election in networks of finite-state anonymous agents,” in *Principles of Distributed Systems.*, 2006.

[15] J. ABRIAL, “Formal approch for software devloppement (approches formelles pour l’aide au dveloppement de logiciels),” 1997.

[16] E. Cohen and S. Shenker, “eplication strategies in unstructured peer-to-peer networks,” *ACM SIGCOMM*, 2002.

[17] J. Buchi, “On a decision method in restricted second order arithmetic,” in *Logik Grundlag. Math*, 1960, p. 6692.