Verification of Orbitally Self-Stabilizing Distributed Algorithms using Lyapunov Functions and Poincaré Maps*

Abhishek Dhama, Jens Oehlerking, Oliver Theel
Carl von Ossietzky University of Oldenburg
Department of Computer Science
D-26111 Oldenburg, Germany
Email: oliver.theel@uni-oldenburg.de

Keywords: Fault Tolerance, Self-Stabilization, Verification, Hybrid Systems, Lyapunov Theory, Poincaré Maps

1 Introduction

A distributed system can be perceived as an environment consisting of multiple potentially heterogeneous computers that are interconnected by a communication infrastructure. The various processes of a distributed application running in this environment exchange information by using this communication infrastructure, e.g., by sending messages to each other. Since distributed applications often consist of a large number of interacting processes their overall successful functioning is highly vulnerable to malfunctioning components. Possible reasons for malfunctioning are, for example, computer failures, process failures, memory failures, communication failures, and network partitions. If the distributed application is safety-critical then it must be designed in a dependable manner able to withstand as many failure scenarios as possible.

A suitable concept for constructing dependable distributed applications is *fault tolerance*. A fault tolerant distributed application is able to contain faults (possibly caused by failures of some lower-level components). If the distributed application is able to fully "cover" the appearance of some class and number of faults to a higher level, for example, the end-user of the application, then this is referred to as *masking fault tolerance*.[1] If – on the contrary – the higher level can perceive an erroneous behavior of the distributed application for some time followed by correct behavior then *non-masking fault tolerance* is obtained. Masking fault tolerance is, for example, realized by replication techniques [5] whereas non-masking fault tolerance is achieved by stabilization techniques.

A stabilization technique for building very dependable distributed algorithm (implemented by a distributed application) is *self-stabilization* [2]. Informally, a (distributed) algorithm exhibits the self-stabilization property, if – starting from an illegal state – the algorithm is guaranteed to return to a specified set of legal states after a finite period of time. Additionally, the set of legal states must be closed under normal system execution, meaning that the algorithm does not voluntarily switch to any illegal state. The definition of legal and illegal states depends on the particular application. Generally, all legal states are specified (e.g., by a state predicate) and illegal states are defined to be those states which are not legal states. This statement seems to be trivial and unimportant but it embodies the entire potential of self-stabilizing algorithms: if the algorithm is guaranteed to reach a legal state from *any* state then it is able to tolerate *any* kind of fault that perturbs the system state (i.e., any kind of transient fault). The resulting algorithms are capable of tolerating fault scenarios which were unforeseeable at design time. Unfortunately, proving self-stabilization of an algorithm is a complicated task [7]. Consequently, research concentrates on finding more adequate verification techniques.

The self-stabilization property of a distributed algorithm as described above exhibits interesting analogies to certain notions of stability of feedback control systems as used in various engineering domains, like electrical and mechanical engineering. Informally, a feedback control system is asymptotically stable, if it converges towards a special equilibrium state. Contrary to the self-stabilization research domain, which is a rather new area of research in computer science, control theory in the engineering domain has a century-old background and offers a well-understood theoretical foundation with powerful criteria for reasoning about the stability of feedback control systems.

The aim of this paper is to narrow the gap between self-stabilization and control theory by adopting and extending criteria originally used for deciding on the stability of feedback control systems for proving self-stabilization of distributed algorithms. In previous work [13, 14], we were concerned with modeling distributed algorithms in terms of *linear* feedback control systems followed by verifying the self-stabilization property via an analysis of a *transfer function* [8]. This approach has been extended for certain classes of "non-linear" algorithms [12, 11, 10], namely those that allow modeling in terms of variable structure non-linear feedback control systems or hybrid systems: here, the verification approach is based on the *Second Method of Lyapunov* [9] which uses *Lyapunov functions* for proving convergence. Convergence together with a closedness argument proves asymptotic

*This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org) and the Graduate School of Trustworthy Software Systems (GRK 1076/1, www.trustsoft.org).

[1] wrt. to this class and number of faults

stability.

In this paper, we extend the class of distributed algorithms whose self-stabilization property can be verified from so-called *silent* to certain *non-silent* algorithms. Whereas a silent algorithms self-stabilizes to a single legal state and remains in it in the absence of faults, a non-silent algorithm converges to a closed set of legal states with more than one legal state and subsequently "transits between these legal states." In this paper, we present an approach based on *Lyapunov theory* [9] and *Poincaré maps* [4] to verify self-stabilization with respect to a closed set of legal states where – in the absence of faults – the legal states are adopted in a *cyclic* manner (i.e., they form an *orbit*). We call those feedback control systems *orbitally stable*. In the context of self-stabilizing distributed algorithms, we call them *orbitally self-stabilizing*. Clearly, orbitally self-stabilizing algorithms are non-silent. Non-silent algorithms are very common in distributed computing. Network or control protocols, for example, represent non-silent distributed algorithms. The basic idea of our approach is to model orbitally self-stabilizing distributed algorithms in terms of hybrid systems and subsequently apply the Lyapunov theory and the *Poincaré map* technique on them. Through Poincaré maps and Lyapunov functions convergence towards a closed set of legal states that form an orbit are proven.

The paper is structured as follows: In the next section, we give the model of computation assumed for self-stabilizing distributed algorithms together with a template of the class of distributed algorithms we cope with in the scope of this paper. Additionally, we introduce hybrid systems, the Poincaré approach as well as Lyapunov functions. In Section 3, we present a non-silent example algorithm and verify its orbital self-stabilization property using the techniques of Section 2. Finally, Section 4 concludes the paper.

2 Proof Methods for Orbital Stability

Generic Algorithm and Computation Model We give an outline of the generic distributed algorithm whose convergence property can be verified using the method presented in this paper. Our model consists of a master process P_S and n slave processes P_{C_i}. All the processes have unique ids. The communication takes place between the master process and the slave processes via shared memory registers. The process bodies[2] are represented as a collection of *guarded commands* [1] (see figures 1 and 2). We assume that all the $n+1$ processes have their clocks synchronized and the execution of the algorithm proceeds in synchronous rounds. In each clock cycle, every process evaluates its guards and executes the active guarded command. The master process P_S has a set of vectors X_R and an integer $next$ as its local variables. X_R uses the set $\mathcal{N}$ as its index set and the integer $next$ is used to index through X_R. The conditions in the guarded commands of the process P_S, $C_k (1 \leq k \leq m)$, partition its state space. The boolean communication variable $reset_{SC}$ indicates that the slave process P_{C_i} should read the value sent by P_S in the next round. If the indexing variable $next$ does not belong to the index set $\mathcal{N}$ then the process P_S does not communicate with any of the slave processes. Every slave process P_{C_i} has a vector X_{C_i} as its local variable. In every clock cycle, the value of X_{C_i} is calculated using its value at the beginning of the cycle if $reset_{SC}$ is false. Otherwise X_{C_i} is found using the value sent by the process P_S.

```
Data: local variable: X_Ci : vector
Data: communication variable: LX_Si, LX_iS : vector
Data: communication variable: reset_Si : boolean
begin
    clock ∧ ¬reset_Si → LX_iS := X_Ci := g_i1(X_Ci)
    clock ∧ reset_Si  → LX_iS := X_Ci := g_i2(LX_Si)
end
```

Figure 1. Skeleton of the slave process $P_{C_i}, 0 \leq i \leq n-1$

```
Data: local variable: X_R[N] : vector array
Data: local variable: next : integer
Data: communication variable: LX_SC[N] : vector array
Data: communication variable: LX_CS[N] : vector array
Data: communication variable: reset_SC[N] : boolean
N ⊆ {0, ..., n' − 1}
Begin
 clock ∧ C_1  → ∀_{i∈N} X_R[i] := f_1(X_R[i], LX_CS[i], i)
                LX_SC[next] := h_1(X_R[next], LX_CS[next])
                ∀_{(i∈N∧(i≠next))} reset_SC[i] := false
                reset_SC[next] := true
                next := (next + 1)%n'
   ⋮
[] clock ∧ C_m → ∀_{i∈N} X_R[i] := f_m(X_R[i], LX_CS[i], i)
                LX_SC[next] := h_m(X_R[next], LX_CS[next])
                ∀_{(i∈N∧(i≠next))} reset_SC[i] := false
                reset_SC[next] := true
                next := (next + 1)%n'
[] clock ∧ (next ∉ N) → ∀_{i∈N} X_R[i] := f_s(X_R[i])
                        ∀_{i∈N} reset_SC[i] := false
                        next := (next + 1)%n'
End
```

Figure 2. Skeleton of the master process P_S

Hybrid Systems Hybrid systems are feedback control systems that contain various modes of operation. Each mode exhibits a possibly different dynamics. While they were originally introduced for the continuous time domain, there have been adaptations of the concept for discrete time as well. Discrete-time hybrid systems are usually specified in the following form:[3]

$$\begin{aligned} x(k+1) &= f_m(x,k) \\ m(k+1) &= g_m(x,k) \end{aligned} \tag{1}$$

The variable m represents a mode of operation and x the state of the system. The functions f_m and g_m govern the dynamics and the mode switches respectively. Therefore, they actually consists only of discrete jumps – a discrete-time hybrid system is nothing but a transition system. However, due to the similarity to the continuous-time case, one can basically employ the same control theoretic methods to analyze them.

[2] % is used to represent the modulo operator

[3] $x(k+1)$ will be shortened to x^+ and $m(k+1)$ to m^+ in the following.

Modeling A distributed algorithm of this type can be modelled as a discrete-time hybrid system in a straightforward manner. One mode $m = (k, j)$ corresponds to one guard C_k and one value of $next$, i.e. the hybrid system will be in mode m if and only if C_k is active and $next = j$. The dynamics for each mode can be derived directly from the processes, taking into account only the commands guarded by that particular C_k. The communication variables $LX_{CS}[i]$ and LX_{iS} need not to be modeled – they are equal to X_{C_i} at any time. Furthermore, LX_{Si} is always equal to $LX_{SC}[i]$. For $\mathcal{N} = \{i_1, \ldots, i_n\}$ with $\forall p : 1 \leq p < n : i_p < i_{p+1}, next = i_1, C_1$ active, $(i_1 - 1)\%n' = i_n$, one obtains these dynamics:

$$\begin{bmatrix} X_R[i_1]^+ \\ LX_{SC}[i_1]^+ \\ X^+_{C_{i_1}} \\ X_R[i_2]^+ \\ LX_{SC}[i_2]^+ \\ X^+_{C_{i_2}} \\ \vdots \\ X_R[i_n]^+ \\ LX_{SC}[i_n]^+ \\ X^+_{C_{i_n}} \\ next^+ \end{bmatrix} = \begin{bmatrix} f_1(X_R[i_1], X_{C_{i_1}}, i_1) \\ h_1(X_R[i_1], X_{C_{i_1}}) \\ g_{i_1 1}(X_{C_{i_1}}) \\ f_1(X_R[i_2], X_{C_{i_2}}, i_2) \\ LX_{SC}[i_2] \\ g_{i_2 1}(X_{C_{i_2}}) \\ \vdots \\ f_1(X_R[i_n], X_{C_{i_n}}, i_n) \\ LX_{SC}[i_n] \\ g_{i_{n-1} 2}(L_{SC}[i_{n-1}]) \\ i_1 + 1 \end{bmatrix} \quad (2)$$

In case $(i_1 - 1)\%n' \notin \mathcal{N}$, the second last line must be changed to $X^+_{C_{i_n}} = g_{i_n 1}(X_{C_{i_n}})$, as there was no writing to $LX_{SC}[i_n]$ during the previous clock cycle. Other cases are derived analogously, directly transferring the guarded commands into this vector form.

Poincaré Maps To verify stability of limit cycles of dynamic systems, the notion of *Poincaré maps* [4] can be used. The idea is to only consider the points of a trajectory where it intersects an explicitly chosen hyperplane in the state space. If the system dynamics are time-invariant, continuous and unique for every initial state $x(0)$ and if there exists an infinite number of intersection points of the trajectory and the hyperplane, then stability analysis can be limited to these intersection points. If, for all arbitrary trajectories, the infinite sequence of intersection points $x(t)$ converges to the same state $\tilde{x}$ then the system possesses a stable limit cycle. Furthermore, this limit cycle is given by the trajectory starting with $x(0) = \tilde{x}$. This trajectory will always form a closed orbit, i.e. $\exists T > 0 : x(t) = x(T + t)$.
We will adapt the notion of Poincaré maps for the class of discrete-time systems given above, with a period of n' and hyperplanes given by $next = j, j \in \{1, \ldots, n'\}$. For each P_{C_i}, a separate Poincaré map can be used, as the variables corresponding to different slave processes are independent of one another. For every such P_{C_i}, the Poincaré map encompassing the behavior over one period can then be brought into the form $x^+ = f_P(x)$, where x contains all variables connected to process P_{C_i} and x^+ denotes the value of x exactly one period later. The function f_P can be obtained through n-times composition of the dynamics from the previous section. If asymptotic stability of all these Poincaré maps is shown (i.e. for each map, each trajectory converges to the same point), then this proves the self-stabilization property. This leaves us with proving convergence to a point now, this point being the intersection between the hyperplane and the limit cycle candidate. Therefore, we can employ Lyapunov methods.

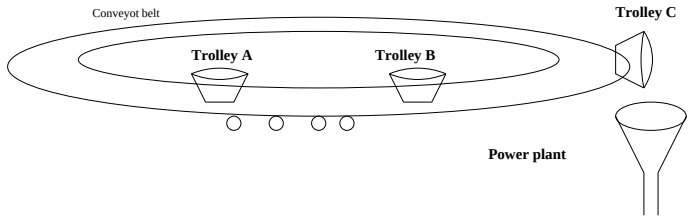

Figure 3. Schematic representing power plant and conveyor-trolley system

Lyapunov Functions To show asymptotic stability of the Poincaré map, Lyapunov functions [6] can be used. For discrete-time systems, Lyapunov functions can be defined as follows.
A function $V(x)$ is a common Lyapunov function of the hybrid system given by Eq. (1) with respect to $\tilde{x}$ with $f_m(\tilde{x}) = \tilde{x}$ for all m, if and only if

$$V(\tilde{x}) = 0 \text{ and } V(x) > 0 \text{ for all } x \neq \tilde{x} \quad (3)$$

$$V(f_m(x)) < V(x) \text{ for all } x \neq \tilde{x} \text{ and for all } m \quad (4)$$

If such a function exists then this system is asymptotically stable in $\tilde{x}$; that is all trajectories converge towards $\tilde{x}$.
Therefore, finding such a function for every Poincaré map concludes the proof, as convergence can only take a finite number of steps, when using integer variables.

3 Example Algorithm

Physical Model We present a distributed algorithm which is used to show the functioning of the verification technique presented in this paper. This algorithm is a *non-silent* distributed algorithm. The non-silent algorithms are a class of distributed algorithms where the contents of local and communication registers change continuously over the execution period [3]. Later in this section we will also explain that the example algorithm exhibits a certain cyclic behavior. The example algorithm roughly models the operation of the conveyor-trolley systems used to transport coal and other raw materials in power plants.

We assume that the conveyor belt is circular (See Fig. 3) and that the conveyor-trolley system consists of a finite number of trolleys. The power plant is assumed to be controlled by a robot that synchronizes the arrival of the trolleys with the power-plant operations. In a run free from any malfunction, the arrival of a trolley will be at the instant when the power-plant needs its contents. But events such as fluctuations in speed of the wheels of the conveyor belt or delay in consumption of raw material in the power-plant can lead to situations where the power-plant and the arrival of the trolleys are "out of synch." In order to avoid such situations,

the robot has to be equipped with fault-tolerance. The robot should maintain the synchronism between the arrival and the consumption in absence of any faults. It would also be highly desirable if the robot again synchronizes the arrival of trolleys with consumption, within a finite number of arrivals, after the malfunctioning has occurred. These properties are analogous to *self-stabilization* of the distributed algorithms [2]. We will next present a self-stabilizing algorithm for the system presented here.

Algorithm Modeling the Physical System The distributed system, mirroring the system presented in the section 3, consists of $n+1$ processes. There are n processes, P_{C_i}, representing the trolleys in the system. The $(n+1)^{th}$ process, P_S, represents the controller robot of the power-plant.

The algorithms implemented by the processes P_{C_i} and

Data: local variable: $\mathsf{X_R}[0..\mathsf{n}-1]$, next : **integer**
Data: communication variable: $\mathsf{LX_{TR}}[0..\mathsf{n}-1]$: **integer**
Data: communication variable: $\mathsf{LX_{RT}}[0..\mathsf{n}-1]$: **integer**
Data: communication variable: $reset_{RT}[0..n-1]$: **boolean**
const : $\mathsf{n}' := (\mathsf{n}+1)$ if $\mathsf{n}\%2 = 0$ else n
cond1 $\equiv \mathsf{X_R}[\mathsf{next}] = \mathsf{LX_{TR}}[\mathsf{next}] = 0$
cond2 $\equiv (\mathsf{next} = \mathsf{n}'-1) \wedge (\mathsf{n}'\%2 \neq 0)$
cond3 $\equiv \neg\mathsf{cond1} \wedge \neg\mathsf{cond2} \wedge (\mathsf{X_R}[\mathsf{next}] \geq \mathsf{LX_{TR}}[\mathsf{next}])$
cond4 $\equiv \neg\mathsf{cond1} \wedge \neg\mathsf{cond2} \wedge (\mathsf{X_R}[\mathsf{next}] < \mathsf{LX_{TR}}[\mathsf{next}])$
Begin
clock $\wedge$ (cond1 $\vee$ cond2) $\rightarrow$
$next := (next+1)\%n'$
$\forall_{0\leq \mathsf{i}<\mathsf{n}} \mathsf{X_R}[\mathsf{i}] := (\mathsf{X_R}[\mathsf{i}]+1)\%\mathsf{n}'$
$\forall_{(0\leq \mathsf{i}<\mathsf{n})\wedge(\mathsf{i}\neq\mathsf{next})} \mathsf{reset_{RT}}[\mathsf{i}] := \mathsf{false}$
$\mathsf{reset_{RT}}[\mathsf{next}] := \mathsf{true}$
$\mathsf{LX_{RT}}[\mathsf{next}] := (\mathsf{LX_{TR}}[\mathsf{next}]+1)\%\mathsf{n}'$
[] clock $\wedge$ cond3 $\rightarrow$
$X_R[next] := (X_R[next]-1)\%n'$
$\mathsf{LX_{RT}}[\mathsf{next}] := (\mathsf{LX_{RT}}[\mathsf{next}]+1)\%\mathsf{n}'$
$\forall_{(0\leq \mathsf{i}<\mathsf{n})\wedge(\mathsf{i}\neq\mathsf{next})} \mathsf{X_R}[\mathsf{i}] := (\mathsf{X_R}[\mathsf{i}]+1)\%\mathsf{n}'$
$\mathsf{reset_{RT}}[\mathsf{next}] := \mathsf{true}$
$\forall_{(0\leq \mathsf{i}<\mathsf{n})\wedge(\mathsf{i}\neq\mathsf{next})} \mathsf{reset_{RT}}[\mathsf{i}] := \mathsf{false}$
$\mathsf{next} := (\mathsf{next}+1)\%\mathsf{n}'$
[] clock $\wedge$ cond4 $\rightarrow$
$X_R[next] := (X_R[next]+1)\%n'$
$\mathsf{LX_{RT}}[\mathsf{next}] := (\mathsf{LX_{RT}}[\mathsf{next}]-1)\%\mathsf{n}'$
$\forall_{(0\leq \mathsf{i}<\mathsf{n})\wedge(\mathsf{i}\neq\mathsf{next})} \mathsf{X_R}[\mathsf{i}] := (\mathsf{X_R}[\mathsf{i}]+1)\%\mathsf{n}'$
$\mathsf{reset_{RT}}[\mathsf{next}] := \mathsf{true}$
$\forall_{(0\leq \mathsf{i}<\mathsf{n})\wedge(\mathsf{i}\neq\mathsf{next})} \mathsf{reset_{RT}}[\mathsf{i}] := \mathsf{false}$
$\mathsf{next} := (\mathsf{next}+1)\%\mathsf{n}'$
End

Figure 4. Process P_S

Data: local variable: $\mathsf{x_{T_i}}$: **integer**
Data: communication variable: $\mathsf{LX_{Ri}}$: **integer**
Data: communication variable: $\mathsf{LX_{iR}}$: **integer**
Data: communication variable: $\mathsf{reset_{Ri}}$: **boolean**
Begin
clock $\wedge \neg\mathsf{reset_{Ri}} \rightarrow \mathsf{LX_{iR}} := \mathsf{x_{T_i}} := (\mathsf{x_{T_i}}+1)\%\mathsf{n}'$
[] clock $\wedge\ \mathsf{reset_{Ri}} \rightarrow \mathsf{LX_{iR}} := \mathsf{x_{T_i}} := (\mathsf{LX_{Ri}}+1)\%\mathsf{n}'$
End

Figure 5. Process P_{C_i}

the process P_S are shown in Figures 4 and 5. The controller process P_S maintains an integer array of size n and an integer $next$ as its local variables. The number n' depends on n, the number of trolleys[4] in the system. n' is $n+1$ if n is an even number else it is equal to n. In each cycle, the process P_S compares the local state of the trolley process P_{C_i} – indexed by the variable $next$ – with $X_R[next]$ and if both the values are equal to zero, implying the guard containing cond1 is active, then P_S increments all the elements of the array X_R modulo n'. If $X_{T_{next}}$ and $X_R[next]$ are not equal to zero then either the guard with cond3 or one with cond4 becomes active and the process P_S increments the lower of the two values and decrements the higher value by 1 (modulo n'). Process P_{C_i}, in every cycle, increments X_{T_i} by 1 modulo n' if there is no reset signal from process P_S. Otherwise it increments the value sent by P_S and assigns it to X_{T_i}.
The global state of the system at any time instant can be represented as a vector whose elements are the local states of the individual processes. The elements of the array X_R are the expected values of X_{T_i}. During a fault-free execution, the trolley local variables and corresponding elements of the array X_R follow the same sequence of integers. The robot tries to overcome a mismatch by adjusting the corresponding variables. The number of rounds required to synchronize the trolley with the power plant depends on the degree of mismatch between respective local variables. The robot also makes sure that once the synchrony has been established, it is not disturbed voluntarily. Trolley failures are independent of each other and caused by the corruption of local variables. The following theorem summarizes the self-stabilizing behavior of the algorithm formally.

Theorem 1 *Let $\mathcal{X} = [X_R, X_{T_1}, \ldots, X_{T_n}]^T$ be the vector representing the state of the system where X_R is the integer array encoding the local state of the process P_S. X_{T_i} is the integer representing the state of the process P_{C_i}. Let $X_{T_i}(k)$ and $X_R[i](k)$ be the values of X_{T_i} and $X_R[i]$, respectively, at the beginning of cycle k and let n' be the successor of n if n is even and n if n is odd where n is the number of trolley processes. Then, the algorithm is self-stabilizing with respect to a state predicate $\mathcal{A}$ where $\mathcal{A} := \{\mathcal{X} \mid (X_R[i](k) = X_{T_i}(k)) \wedge (X_R[i](k) = (i+k-1)\%n')\}$.* □

Modeling the Example as a Hybrid System Using the transformation technique presented in section 2, we obtain a hybrid system with one mode per guard and possible value of $next$. For example, for $next = 0$, the first guard and an odd n we obtain the following mode dynamics.

$$\begin{bmatrix} X_R[0]^+ \\ LX_{RT}[0]^+ \\ X^+_{T_0} \\ X_R[1]^+ \\ LX_{RT}[1]^+ \\ X^+_{T_1} \\ \vdots \\ X_R[n-1]^+ \\ LX_{RT}[n-1]^+ \\ X^+_{T_{n-1}} \\ next^+ \end{bmatrix} = \begin{bmatrix} (X_R[0]+1)\%n' \\ LX_{RT}[0] \\ (X_{T_0}+1)\%n' \\ (X_R[1]+1)\%n' \\ LX_{RT}[1] \\ (X_{T_1}+1)\%n' \\ \vdots \\ (X_R[n-1]+1)\%n' \\ LX_{RT}[n-1] \\ (LX_{RT}[n-1]+1)\%n' \\ (next+1)\%n' \end{bmatrix} \quad (5)$$

[4]One can think of $(n')^{th}$ trolley as a "phantom" trolley which does nothing.

Note that process $P_{C_{n-1}}$ does its assignment for $reset_{R_{n-1}} = true$ when $next = 0$. This is due to the one-step communication delay.

Poincaré Maps for each Trolley To obtain the Poincaré maps for each trolley, we need to calculate the dynamics of all variables involving the trolley over one period of n' clock cycles. If we take care that the cutting plane is chosen such that communication is always completed within one cycle, the communication variables become redundant for the Poincaré map. Whenever the robot is not currently communicating with the trolley (i.e. $next$ is not equal to the trolley's number), then $X_R[i]$ and X_{T_i} are just increased by 1 (modulo n') per clock cycle. The Poincaré map for each trolley can be described by three dynamics, each one being tied to one guard of P_S. For the first trolley these are:
First guard:

$$\begin{bmatrix} X_R[i]^+ \\ X_{T_i}^+ \end{bmatrix} = \begin{bmatrix} X_R[i] \\ X_{T_i} \end{bmatrix} \tag{6}$$

Second guard:

$$\begin{bmatrix} X_R[i]^+ \\ X_{T_i}^+ \end{bmatrix} = \begin{bmatrix} (X_R[i]-2)\%n' \\ X_{T_i} \end{bmatrix} \tag{7}$$

Third Guard:

$$\begin{bmatrix} X_R[i]^+ \\ X_{T_i}^+ \end{bmatrix} = \begin{bmatrix} X_R[i] \\ (X_{T_i}-2)\%n' \end{bmatrix} \tag{8}$$

Combining them into a discrete-time hybrid system with a switching according to the guards, we obtain a description of a trolley's behavior over one period. Orbital stability is then equivalent to stability of each Poincaré map system obtained this way.

Lyapunov Functions for each Trolley A Lyapunov function for each Poincaré map is given by the following expression:

$$V_i(X_R[i], X_{T_i}) = X_R[i] + X_{T_i} + n' * (X_R[i]\%2 + X_{T_i}\%2) \tag{9}$$

Proof. We have $V_i(0,0) = 0$ and $V_i(X_R[i], X_{T_i}) > 0$ everywhere else, if we assume that $X_R[i] \geq 0$ and $X_{T_i} \geq 0$. This assumption is no limitation, as negative variable values would disappear after one clock cycle in any case.

$$V_i(X_R[i](t+n'), X_{T_i}(k+n')) < \tag{10}$$
$$V_i(X_R[i](k), X_{T_i}(k)). \tag{11}$$

Case $\mathbf{X_R[i](k) > X_{T_i}(k)}$: In this case, the states are updated according to Eq.(7). If $X_R[i](k+n') < X_R[i](k)$, then $X_R[i]\%2 + X_{T_i}\%2$ will remain unchanged, this implies Eq. (10). If $X_R[i](k+n') \geq X_R[i](k)$, then $X_R[i](k) = 1$ and $X_R[i](k+n') = n'-2$. Therefore $X_R[i](k)\%2 = 1$ and $X_R[i](k+n')\%2 = 0$. Together with $X_R[i](k+n') < X_R[i](k) + n'$ we obtain Eq. (10).
Case $\mathbf{X_R[i](k) < X_{T_i}(k)}$: Analogous, exchanging $X_R[i]$ and X_{T_i}.
Case $\mathbf{X_R[i](k) = X_{T_i}(k)}$: If $X_R[i](k) = X_{T_i}(k) = 0$ then there is nothing to show. Otherwise, the considerations for the first case apply. □
This proves stability of each Poincaré map with respect to $X_R[i](k) = X_{T_i}(k) = 0$ and therefore orbital stability of the entire system according to Theorem 1.

4 Conclusion

We have described a class of distributed algorithms which shows a certain cyclic behavior. For this class, we have detailed a method for proving self-stabilization with respect to an orbit of states. This is done using the control theoretic notions of Poincaré maps and Lyapunov functions.

References

[1] Edsger W. Dijkstra. Guarded commands, nondeterminancy, and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

[2] Shlomi Dolev. *Self-Stabilization*. MIT Press, March 2000.

[3] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Proceedings of the fifteenth annual ACM PODC*, pages 27–34, 1996.

[4] J. Guckenheimer and P. Holmes. *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*. Springer, 1983.

[5] Pankaj Jalote. *Fault tolerance in Distributed Systems*. Prentice-Hall, 1994.

[6] R. E. Kalman and J. E. Bertram. Control system analysis and design via the "second method" of Lyapunov. *Trans. of the ASME, Journal of Basic Engineering*, pages 371–400, 1960.

[7] J. L. W. Kessels. An Exercise in Proving Self-Stabilization with a Variant Function. *Information Processing Letters*, 29:39–42, 1988.

[8] William S. Levine. *The Control Handbook*. CRC Press, 1995.

[9] M. A. Lyapunov. Problème général de la stabilité du movement. *Ann. Fac. Sci. Toulouse*, 9:203–474, 1907. (Translation of a paper published in Comm. Soc. math. Kharkow, 1893, reprinted in Ann. math. Studies No. 17, Princeton University Press, 1949).

[10] Jens Oehlerking, Abhishek Dhama, and Oliver Theel. Towards Automtic Convergence Verification of Self-Stabilizing Algorithms. In *Proc. of the 7th Intern. Symposium on Self-Stabilizing Systems (SSS 2005)*, LNCS 3764. Springer, September 2005.

[11] Oliver Theel. Exploitation of Ljapunov Theory for Verifying Self-Stabilizing Algorithms. In *Proc. of the 14th Symposium on Distributed Computing (DISC'00), Toledo, Spain, LNCS Vol. 1914*, LNCS, pages 209–222. Springer, October 2000.

[12] Oliver Theel. Proving Convergence and Closure of Self-Stabilizing Algorithms through Ljapunov's "Second Method". *Proc. of the 3rd Intern. Symp. on Intelligent Automation and Control (ISIAC'00), Maui, HI, U.S.A.*, Intelligent Automation and Control: Recent Trends in Development and Applications. TSI Press, June 2000.

[13] Oliver Theel and Felix C. Gärtner. On Proving the Stability of Distributed Algorithms: Self-Stabilitzation vs. Control Theory. *Proc. of the International Conference on Systems, Signals, Control, Computers (SSCC'98)*, pages 58–66. IAAMSAD and SA branch of ANS, September 1998.

[14] Oliver Theel and Felix C. Gärtner. An Exercise in Proving Convergence through Transfer Functions. In *Proc. of the 4th Workshop on Self-Stabilizing Systems (WSS'99), being part of the 19th International Conference on Distributed Computer Systems (ICDCS'99)*, pages 41–47. IEEE, June 1999.