# Synthesizing Time-Triggered Schedules for Switched Networks with Faulty Links

Guy Avni[*]
IST Austria
guy.avni@ist.ac.at

Shibashis Guha[†]
Hebrew University, Israel
shibashis@cs.huji.ac.il

Guillermo Rodriguez-Navas[‡]
Mälardalen University, Sweden
guillermo.rodriguez-navas@mdh.se

## ABSTRACT

Time-triggered (TT) switched networks are a deterministic communication infrastructure used by real-time distributed embedded systems. These networks rely on the notion of globally discretized time (i.e. time slots) and a static TT schedule that prescribes which message is sent through which link at every time slot, such that all messages reach their destination before a global timeout. These schedules are generated offline, assuming a static network with fault-free links, and entrusting all error-handling functions to the end user. Assuming the network is static is an over-optimistic view, and indeed links tend to fail in practice. We study synthesis of TT schedules on a network in which links fail over time and we assume the switches run a very simple error-recovery protocol once they detect a crashed link. We address the problem of finding a $(k, \ell)$-resistant schedule; namely, one that, assuming the switches run a fixed error-recovery protocol, guarantees that the number of messages that arrive at their destination by the timeout is at least $\ell$, no matter what sequence of at most $k$ links fail. Thus, we maintain the simplicity of the switches while giving a guarantee on the number of messages that meet the timeout. We show how a $(k, \ell)$-resistant schedule can be obtained using a CEGAR-like approach: find a schedule, decide whether it is $(k, \ell)$-resistant, and if it is not, use the witnessing fault sequence to generate a constraint that is added to the program. The newly added constraint disallows the schedule to be regenerated in a future iteration while also eliminating several other schedules that are not $(k, \ell)$-resistant. We illustrate the applicability of our approach using an SMT-based implementation.

## Keywords

Real-time communication; Scheduling; Fault tolerance; Satisfiability Modulo Theory.

## 1. INTRODUCTION

Embedded systems can be found nowadays in almost every activity of our daily life. Be it in transportation, automation, energy or healthcare, the current tendency is to build more complex and larger embedded systems. An increasing number of these systems are distributed and rely on message exchange over *switched* communication networks. The switches used in such networks are typically fast hardware devises with limited computational power, interconnected with high-speed links that can accommodate a single message at a time.

For distributed embedded systems requiring deterministic communication, i.e. satisfaction of *hard* message deadlines, the so-called Time-Triggered (TT) scheduling of messages has been advocated [27]. TT scheduling assumes the existence of a common notion of time throughout the system, usually implemented with some kind of clock synchronization protocol, therefore allowing a globally consistent definition of *time slots* [19]. The switches follow a *schedule* that prescribes which message is sent through each link for every time slot. The schedule is synthesized offline, and it is repeated cyclically during the system operation. Adopting the TT scheduling for communication is compelling because it provides predictability (messages are known to arrive by the global timeout) but also because it demands very little computation from the switches; their functionality can be implemented with a simple lookup table.

An important limitation of the TT scheduling is the computational complexity of synthesizing a schedule, which is known to be an NP-complete problem. Much work has been done on optimization and improving the scalability of schedule synthesis. Different techniques have been adopted for synthesis of time-triggered schedules for medium size systems, such as constraint programming (CP) solvers [25], integer linear programming (ILP) solvers [14], and satisfiability modulo theory (SMT) solvers [18]. For systems or larger size, SMT solvers have been combined with iterative/segmentation techniques yielding good results [11, 4, 26, 23, 12], even if compared to ILP.

A second limitation of TT scheduling is that it assumes a static network. Namely, once a schedule is constructed, it does not adapt

[*]This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).

[†]The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 278410

[‡]Work partially funded by the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme FP7/2007-2013/ under REA grant agreement 607727



*EMSOFT'16, October 01-07, 2016, Pittsburgh, PA, USA*



DOI: http://dx.doi.org/10.1145/2968478.2968499

to changes in the network. This assumption is optimistic, since switched networks can change intentionally, for reconfiguration purposes, or unintentionally, as a consequence of either link or switch failures. Changes caused by failures are particularly challenging and constitute the focus of this work.

In time-triggered networks, failures are typically handled in the application layer by using redundancy [3]. Two common types of redundancy are *temporal* and *spatial* redundancy. Temporal redundancy consists in scheduling a message more than once per cycle, with the time slots assigned to the message replicas conveniently spread over the schedule. This provides good tolerance to transient faults of the links, but it does not provide tolerance to permanent crashes. To achieve tolerance to link crashes, spatial redundancy has to be used. Here, a message should be scheduled over at least two different paths, and the defined paths should be as disjoint as possible, in order to prevent common-mode failures. These solutions are simplistic and not exempt from problems. On the one hand, the additional paths and retransmissions need to be taken into consideration during the schedule synthesis, increasing the complexity of this already demanding task. On the other hand, the required additional resources (bandwidth at the links, and processing time and memory at the switches) are allocated in a static manner, thus decreasing the actual utilization of the network.

In this work we study an alternative approach to handling permanent link failures, which aims at avoiding or reducing redundancy, while maintaining the philosophy of using simple switches. Our technique is a variation of the well known Primary/Backup (P/B) approach to fault tolerance [9]. We assume the switches follow a TT schedule. Once they observe a fault, the switches resort to a simple *error-recovery protocol* for handling the situation. We describe two simple protocols to illustrate our concept.

**The do-nothing protocol** We refer to the simplest protocol we consider as the *do-nothing* protocol. As its name suggests, when a message is scheduled to be sent via a link that has crashed, then no corrective action is taken by the switch, and the message is discarded. Thus, the message does not arrive at its destination. This approach relies exclusively on the faults being handled by the application as described above. This simple protocol is often used in practice [27]. □

**The two-path protocol** The second protocol we describe will be our running example throughout this paper, and it is the protocol that we have implemented. We refer to it as the *two-path protocol*. We use $\mathcal{M}$ to refer to the set of messages. We assume that each message $m \in \mathcal{M}$ has a *first path* $\pi_m$ on which it is scheduled, where a path is a complete route from the node sending $m$ to the destination (or target) of $m$. Thus a schedule only schedules $m$ on links that $\pi_m$ traverses. Such an assumption has been used often in practice in order to find schedules in large networks [26, 23]. We allow the switches on $\pi_m$ to have a *second path* to the target of $m$, which can be thought of as a *fallback path*; systems using slight variations of this protocol can be found in the literature [29, 21]. In our protocol, and similar to the first path, the second paths are predetermined and given as input, i.e. the path is not found in an online manner. If $m$ is at a switch $v$ on $\pi_m$ and the next link in the path crashes, then $v$ attempts to forward $m$ on the second path that starts from $v$. The schedule does not specify how a message is forwarded along a second path. We assume that there is a total order $\prec$ on the messages, and $m$ uses a link $e$ on the second path only when it is *free* for the current slot, i.e., the schedule has not assigned $e$ to any message and no message $m' \prec m$ also tries to use $e$. □

Our goal is to give a *guarantee* on the performance of the system assuming that the degree of failure is reasonable. To formalize this notion, we define a $(k, \ell)$-resistant schedule. The faults we consider are *fail stop*. Namely, a link may crash and does not recover after crashing. We note that once one fixes a schedule and a protocol for the switches, the system's response to failures is deterministic. We say that a schedule is $(k, \ell)$-resistant if, assuming the switches run a fixed error-recovery protocol and at most $k$ crashes occur, at least $\ell$ messages are delivered by the global timeout.

In the context of embedded systems, the ability of a system to perform reasonably well upon unexpected (yet reasonable) environment changes is known as *robustness*; where environment refers to the assumptions upon which the system has been built [16]. A system is said to be *robust* if small disturbances of the environment can cause only small system errors, which corresponds as well with the definition of *graceful degradation* in classical fault tolerance [5]. Robustness has been discussed in relation to predictability as two "universal challenges" of embedded systems design [16], even if they are seldom addressed together. Our definition of $(k, \ell)$-resistant schedule can be interpreted as a variation of the notion of $k$-robustness introduced in [5].

We study the problem of finding a $(k, \ell)$-resistant schedule for a given set of messages in a network, assuming the switches run a fixed protocol PROT. As a first step, we solve the problem of deciding whether a given schedule $S$ is $(k, \ell)$-resistant. We describe an SMT program that is satisfiable iff $S$ is not $(k, \ell)$-resistant. In the case the program is satisfiable, a satisfying assignment corresponds to a fault sequence $F$ that can be thought of as a *witness* to the non-resistance of $S$. Indeed, when the switches follow $S$ and run the protocol PROT, if the faults $F$ occur, then less than $\ell$ messages arrive at their destination by the timeout.

We use our solution above to devise a CEGAR-like method[1] to find a $(k, \ell)$-resistant schedule. Consider a network $\mathcal{N}$ and a set of messages $\mathcal{M}$. We start with a program $P$ that is satisfiable iff there is a schedule for $\mathcal{M}$ in $\mathcal{N}$. Note that a valid schedule is one that delivers all messages when there is no link failure. If $P$ is not satisfiable, there is no schedule, and clearly there is no $(k, \ell)$-resistant schedule, and we terminate. Otherwise, we find a schedule $S$, and we use the method above to check whether $S$ is $(k, \ell)$-resistant. If it is, we are done. If it is not, we use the witness fault sequence to generate a constraint $\neg\psi$, add it to $P$, and repeat the process.

The crux of the procedure involves choosing $\neg\psi$. On one hand, we want $\neg\psi$ to be as strong as possible, thereby eliminating as many schedules as possible in one iteration. On the other hand, we want it to be sound, i.e., we want to eliminate only schedules that are not $(k, \ell)$-resistant. Recall that $F$ is a fault sequence that witnesses the non-resistance of $S$. We choose $\neg\psi$ as follows. Consider a schedule $S'$ that satisfies $\psi$, and is thus eliminated when adding $\neg\psi$ to $P$. If the switches follow $S'$ and $F$ occurs, then the execution of the system, namely how the messages are directed in the network over time, is identical to the one when they follow $S$. In particular, since less than $\ell$ messages meet the timeout when the switches follow $S$, less than $\ell$ messages meet the timeout when the switches follow $S'$, thus $S'$ is not $(k, \ell)$-resistant.

Finally, we add an optimization. We note that the execution of the system is discrete and can be thought of as snapshots; the $i$-th snapshot includes the positions of the messages at time $i$. We

[1] *Counter-example guided abstraction refinement* (CEGAR, for short) [10] is a well-known technique in model checking in which an abstraction of a system is constructed and tested for correctness. The abstraction is an over approximation, in the sense that if the abstract system is correct, so is the concrete one. But, a counterexample showing incorrectness of the abstract system might not have a corresponding concrete counterexample. In that case, the trace is *spurious* and it is used in order to refine the abstraction.

check whether there is a time point $i$ such that the $i$-th snapshot is *doomed*: assuming the faults in $F$ occur, there is no schedule $S'$ whose execution matches the execution of $S$ up to time $i$ and is still able to deliver at least $\ell$ messages. Thus, we can strengthen $\neg\psi$ to $\neg\psi_i$; if a schedule $S'$ satisfies $\psi_i$, then its execution matches that of $S$ up to time $i$. Since the $i$-snapshot is doomed, the schedule $S'$ is not $(k, \ell)$-resistant. In order to decide whether there exists such a schedule we devise a third SMT program that can be thought of as a mix between the two previous programs.

We have implemented our approach and evaluated it on randomly generated networks. In our implementation, the switches use the two-path protocol that is described above. We use Z3 as our SMT solver [13]. We show that the optimization described above is very useful. It significantly reduces the number of iterations needed in the CEGAR loop, especially when the answer is that there is no $(k, \ell)$-resistant schedule for the given input. Also, during the evaluation, we found that the notion of a *best schedule* is helpful. Namely, a schedule for a given setting that is $(k, \ell)$-resistant such that no $(k, \ell + 1)$-resistant schedule exists. We suggest a simple workflow for finding such a schedule, and show that it performs well.

## 2. RELATED WORK

Message scheduling is only one of the problems in distributed systems that is solved using TT-scheduling. Another important problem is *task scheduling*, namely deciding which task is executed by each processor in every time slot. Both these scheduling problems are related, but they have been traditionally addressed separately (c.f., [26, 18, 23]). In this paper we follow this traditional approach, where we focus on the message scheduling aspect of the system while disregarding the task scheduling, thus we assume reasoning on the second is performed separately. Solving the two problems together is an interesting problem and positive results have recently been shown [12].

We assume the switches in the network have very limited computational power. Taking the other extreme are Software Defined Networks (SDN), which are growing in popularity. These networks bring new opportunities regarding traffic management to distributed systems [20], including novel and advanced mechanisms for handling link failures [24]. The problem we study here might be useful for reasoning and comparing different forwarding rules applied in the control plane of an SDN.

Traditional synthesis of TT schedules is a Boolean problem; a schedule is correct if it satisfies all its requirements, and otherwise it is incorrect. However, there may be many correct schedules for a certain input. The schedules may differ in "quality", and choosing a schedule of high quality is an interesting problem. "Quality" is an amorphous concept and can change according to the context. For example, a schedule that spreads the messages and thus lowers the memory consumption of the switches might be preferable in contexts where memory is an issue, and may be undesirable when there are many faults. Our $(k, \ell)$-resistant definition can be viewed as a measure of quality, and thus as a method to compare correct schedules.

A similar challenge is subject of increasing research in the formal methods community [17]. There, the traditional notion of correctness of a system is Boolean; it either satisfies the specification, or does not. In recent years there has been a growing effort to lift the ideas of traditional formal methods to the quantitative setting. So, for example, rather than asking whether the system satisfies the specification, one might ask *how well* the system satisfies the specification. The origin of the quantitative aspect can be in the system [7], the specification [1], or both [6]. Similar to the examples above, it can be used to model consumption of resources by the system or other systems using the same resources [2].

Our approach is inspired by *Sabotage games* [28]. The traditional game is played on a directed graph by two players that alternate turns. The game starts by placing a token on the initial vertex. *Runner* moves the token. In his turn, he chooses one of the outgoing edges from the vertex on which the token is placed, and moves the token to the endpoint of the edge. In his turn, *Saboteur* "breaks" one of the edges and removes it from the graph. Runner wins iff the token reaches a designated target vertex. The central question in sabotage games is deciding which one of the players has a winning strategy, namely a strategy such that no matter how the other player plays, guarantees winning. This problem is PSPACE-complete already in the traditional game [22].

Our setting can be formalized as a type of sabotage game in which Runner moves several tokens at once, where each token corresponds to a message, and thus has its own source and target vertices. We think of a schedule as a Runner strategy and a fault sequence as a Saboteur strategy. Note that the high complexity of the traditional game follows to our game, and we overcome it by adding a timeout and by imposing strict restrictions on the possible strategies of the players. Our Saboteur does not break an edge in each turn, rather he has a limit ($k$) on the number of edges he is allowed to break. Runner's restriction is even stronger. Our restriction that the switches computational power is limited corresponds to a restriction that Runner has *bounded rationality*, and can select only very simple strategies.

Now, the problem of finding a $(k, \ell)$-resistant schedule can be rephrased as finding a winning strategy for Runner (that obeys the restrictions above). The problem of deciding whether a schedule is $(k, \ell)$-resistant can be rephrased as deciding whether a given Runner strategy is winning. Note that if it is not, then there is a Saboteur strategy that wins against it, which corresponds to a fault sequence. Finally, in the optimization in the CEGAR loop, we alternate roles. We fix a Saboteur strategy and ask whether it is winning against every Runner strategy.

Recently, sabotage games where considered in weighted graphs [8]. One of the challenges of lifting the Boolean setting to the quantitative setting as described above, is that often, there is not only one possible way to lift a problem. The variant of sabotage games that arises from our setting is a different approach to adding quantity to these games. Namely we require that at least $\ell$ tokens (messages) arrive on time.

## 3. PRELIMINARIES

We model a network as a directed[2] graph $\mathcal{N} = \langle V, E \rangle$. A collection $\mathcal{M}$ of messages are sent through the network. Each message $m \in \mathcal{M}$ has a source and a target vertex, which we refer to as $s(m)$ and $t(m)$, respectively. There is a global timeout $t \in \mathcal{N}$ and a message meets the timeout if it arrives in its destination by time $t$.

### *Schedule*

A schedule is a function $S : E \times \mathbb{N} \to (\mathcal{M} \cup \{\perp\})$, where $\mathbb{N}$ are the positive integers. Having $S(e, i) = m \in \mathcal{M}$ means that message $m$ is forwarded from $s(e)$ to $t(e)$ at time $i$, and having $S(e, i) = \perp$ means that no message is assigned to $e$ at time $i$. We require the schedule to satisfy the following constraints:

1. *Path:* Every message originates from its source vertex and reaches its target vertex, and a switch cannot forward a message that has not arrived in it.

[2]We choose a directed graph rather than undirected graph for ease of notation.

2. *Contention free:* Two messages cannot be sent on the same link at the same time.

3. *Timeout:* All messages arrive by a global timeout $t$.

**Theorem 3.1.** *The problem of finding a correct schedule can be solved using a SAT solver.*

**Proof:** Consider a network $\mathcal{N} = \langle V, E \rangle$, a set of messages $\mathcal{M}$, and a timeout $t \in \mathbb{N}$. Let $X = \{x_{e,m,i} : e \in E, m \in \mathcal{M}, 0 \leqslant i < t\}$ be a set of variables, which we refer to as *schedule variables*. We describe the constraints of the program, which match the requirements on schedules above, so that a truth assignment $f : X \to \{\mathtt{tt}, \mathtt{ff}\}$ corresponds to a schedule in a natural manner; For $e \in E$ and $0 \leqslant i < t$, we have $S(e,i) = m$, if $f(x_{e,m,i}) = \mathtt{tt}$, and if for every $m \in \mathcal{M}$ we have $f(x_{e,m,i}) = \mathtt{ff}$, then $S(e,i) = \perp$.

We start with the path requirement. For every vertex $v \in V$, let $out(v) \subseteq E$ be the outgoing edges from $v$. Consider a message $m \in \mathcal{M}$. Recall that we require that $m$ originates from its source $s(m)$ and that it is scheduled on an edge $e = \langle u, v \rangle$ only after it arrived in $u$. Further recall that we view an assignment of $\mathtt{tt}$ to $x_{e,m,i}$ as a schedule that schedules $m$ on $e$ at time $i$. So, if $x_{e,m,i}$ gets $\mathtt{tt}$, there should be an edge $e'$ with $t(e') = s(e)$ and a time $j < i$ such that $x_{e',m,j}$ gets $\mathtt{tt}$. That is, $m$ "arrives" in $s(e)$ on $e'$ before it is sent on $e$. We leave out outgoing edges from $s(m)$ thereby allowing $m$ to originate from $s(m)$. Formally, for every message $m$, edge $e$ that is not in $out(s(m))$, and time $i > 1$, we have the constraint

$$x_{e,m,i} \rightarrow \bigvee_{e' \in E \text{ s.t. } t(e') = s(e) \text{ and } j < i} x_{e',m,j}$$

. Note that the constraint above does not take care of the first time step. Thus, we add, for every message $m$ and edge $e$ such that $s(e) \neq s(m)$, the constraint $\neg x_{e,m,0}$.

Next, we handle the contention-free requirement. Recall that we require that two messages are not sent on the same link at the same time, thus if $m \in \mathcal{M}$ is scheduled on an edge $\forall e \in E$ at time $1 \leqslant i \leqslant t$, no other message should be scheduled on it:

$$x_{e,m,i} \rightarrow \neg\big(\bigvee_{m \neq m' \in \mathcal{M}} x_{e,m',i}\big).$$

Finally, we require that every message $m \in \mathcal{M}$ arrives by the timeout $t$:

$$\bigvee_{1 \leqslant i \leqslant t \text{ and } e = \langle u, t(m) \rangle \in E} x_{e,m,i}.$$

We view an assignment of $\mathtt{tt}$ to a variable $x_{e,m,i}$ as scheduling $m$ on $e$ at time $i$. Each of the constraints above correspond to a requirement on schedules. Thus, there is a one-to-one correspondence between assignments that satisfy all the constraints to correct schedules. ∎

## *Fault model*

We consider *fail stop* failures: if an edge crashes, it does not recover. We assume that the switches are aware of crashes immediately when they occur. A failure sequence is a sequence of subsets of edges $F = T_0, \ldots, T_{t-1}$, where $T_i \subseteq E$, for $0 \leqslant i \leqslant t-1$. Having $e \in T_i$ means that $e$ has crashed at time $i$. Since edges that crashed do not recover, we have $T_i \subseteq T_{i+1}$, for $0 \leqslant i < t-1$. We say that a fault sequence performs $k$ faults if $|T_{t-1}| = k$.

## *Error-recovery protocol*

Switches react to link failures. In the absence of faults, they follow the schedule. When they detect a crash, they resort to a predefined *error-recovery protocol* that dictates how to forward the messages given the failed edges. An error-recovery protocol is an algorithm that is run in a switch. It takes as input the messages in the switch's queue, the crashed edges, and the time, and returns through which link the messages are to be forwarded[3]. While forwarding the messages, the protocol needs to satisfy the constraints of the system. Namely, two messages cannot use the same link at the same time, and a message cannot use a link that has crashed. In the presence of a link failure, it cannot however be guaranteed that all messages will reach their destinations before the global timeout.

Forwarding depends on several parameters; the messages in $v$'s queue, the crashed edges, and the time. We think of the schedule as being "hard coded" into the protocol. We consider very simple protocols as we assume the switch's computational power is very limited. Namely, we consider protocols that are given as a set of propositional rules of the form $\varphi \implies u$. Such a rule corresponds to a message $m \in \mathcal{M}$. If $\varphi$ is satisfied, then $m$ is forwarded from $v$ to $u$ in the next time step. We consider deterministic protocols, thus we assume that for every message at a given time point there is exactly one assertion that is satisfied.

Formally, we use PROT to denote a protocol, we use $\text{PROT}_{S,v}$ to indicate that PROT is run at vertex $v \in V$ with respect to a schedule $S$. A rule for a message $m \in \mathcal{M}$ at $v \in V$, is of the form $\varphi \implies u$, where we refer to $u \in V$ as the *target vertex* of the rule, and to $\varphi$ as the *assertion* of the rule, where the assertion $\varphi$ is given by the following grammar:

$$\varphi \ ::= \ m \mid e \mid S(e,\$) = \mu, \text{ for } \mu \in \mathcal{M} \cup \{\perp\} \mid \neg\varphi \mid \varphi \vee \varphi$$

We denote by $\text{PROT}^m_{S,v}$ the subset of rules that corresponds to a message $m \in \mathcal{M}$.

The semantics of $\varphi$ is with respect to the input to the protocol, namely, a set of messages $M \subseteq \mathcal{M}$, which can be thought of as the messages in $v$'s queue, a set of crashed edges $T \subseteq E$, and a time point $0 \leqslant i \leqslant t-1$. Consider a rule $\varphi \implies u$ in $\text{PROT}_{S,v}$. We use $(M,T,i) \models_S \varphi$ to denote the fact that $(M,T,i)$ satisfies the assertion $\varphi$, in which case $m$ is forwarded to $u$ at the next time step. In particular, we assume that $\varphi$ is satisfiable only when $m$ is in $v$, i.e., $m \in \mathcal{M}$. The semantics is defined inductively on the structure of $\varphi$. If $\varphi = m$, then $(M,T,i) \models_S \varphi$ iff $m \in M$, if $\varphi = e$, then $(M,T,i) \models_S \varphi$ iff $e \in T$, if $\varphi = \big(S(e,\$) = \mu\big)$, for $\mu \in \mathcal{M} \cup \{\perp\}$, then $(M,T,i) \models_S \varphi$ iff $S(e,i) = \mu$, and the two inductive cases are as expected.

Also, recall that the switches first forward according to the schedule, and only if a crash occurs, the protocol is used. For convenience, we assume this policy is included as rules in the protocol. For example, in the two-path protocol, we assume that for every $m \in \mathcal{M}$, $v \in \pi_m$, and $e = \langle v, u \rangle \in E$ that is an edge that $\pi_m$ traverses, we have two rules in $\text{PROT}^m_{S,v}$. The first rule indicates that if $m$ is scheduled on $e$ and $e$ has not crashed, it should be forwarded on it: $\big(S(e,\$) = m\big) \wedge \neg e \implies u$. The second rule indicates that if $e$ has not crashed and $m$ is not scheduled on $e$, it should wait for its turn and stay in $v$: $\neg\big(S(e,\$) = m\big) \wedge \neg e \implies v$.

**Example 3.1.** Recall that in the do-nothing protocol, when a message is scheduled on an edge that has crashed, no corrective action is taken and the message stays in the vertex. The rules that describe

[3]The assumption that switches are aware of all the crashes in the network is strong and impractical. It allows, for example, to strengthen the two-path protocol to forward messages on their second path if a crash occurs somewhere on the first path, which is particularly helpful if there are switches with no backup path. We can easily weaken this global assumption to a local one in which switches are only aware of crashes in their outgoing edges. But, we keep the strong global assumption for ease of presentation.

the do nothing protocol are simple. Consider a message $m \in \mathcal{M}$ and a vertex $v \in V$. Let $e \in E$ be the outgoing edge from $v$ on which $S$ forwards $m$ at some time point. The rule for $m$ in $v$ in do nothing protocol states that if $m$ is at $v$ and $e$ crashes, then stay in $v$. Written as a propositional rule it is $m \wedge e \implies v$. □

**Example 3.2.** We describe some of the rules of the two-path protocol. Recall that we assume that each message $m \in \mathcal{M}$ has a first path $\pi_m$ on which it is scheduled, and we allow each vertex on $\pi_m$ to have a second path to $t(m)$, which is taken from $v \in \pi_m$ if the next outgoing edge from $v$ that $\pi_m$ traverses, crashes. We assume the paths are given using partial functions FP, SP : $\mathcal{M} \times V \to V$, where FP$(m, v)$, for $v \in \pi_m$, returns the vertex following $v$ on $\pi_m$, and SP$(m, v)$ returns the fallback vertex from $v$ for message $m$.

Consider a vertex $v \in V$ and a message $m \in \mathcal{M}$ for which $v$ is on one of the second paths of $v$. Let $e \in E$ be the *second choice edge* from $v$, thus $e = \langle v, \text{SP}(m, v)\rangle$. We describe an assertion $free(e, m)$ that gets a true value iff $e$ is free for $m$. Namely, no message is scheduled on it in the schedule $S$, and no message $m' \prec m$ tries to use it. The second restriction is harder to state, and we do it in a few steps.

Let Prc $\subseteq \mathcal{M}$ be the messages that have precedence over $m$ and have the same second choice edge $e$ from $v$, thus Prc $= \{m' \in \mathcal{M} : m' \prec m$ and SP$(m', v) =$ SP$(m, v)\}$. Thus, if no message is scheduled by $S$ on $e$, the edge $e$ is free for $m$ if no message in Prc tries to use $e$. Consider a message $m' \in$ Prc. We distinguish between two cases. If $v$ is on the first path $\pi_{m'}$, then $m'$ attempts to use $e$ if $m'$ is on $v$, and the next edge on the path has crashed. Written as an assertion it is $attempt$-FP$(v, m') = m' \wedge \langle v, \text{FP}(m', v)\rangle$. If $v$ is not on the first path, then $m'$ attempts to use $e$ if it is on $v$ as there is no first choice edge leaving $v$. Written as an assertion it is $attempt$-SP$(v, m') = m'$.

We can now write the full assertion $free(e, m)$. Recall that $e$ is free if there is no message scheduled on it and no message in Prc attempts to use it:

$$free(e,m) = \big(S(e,\$) = \bot\big) \wedge \bigwedge_{m' \in \text{Prc s.t. } v \in \pi_{m'}} \neg attempt\text{-FP}(v,m') \wedge \bigwedge_{m' \in \text{Prc s.t. } v \notin \pi_{m'}} \neg attempt\text{-SP}(v,m').$$

We use $free(e, m)$ in the forwarding rules for $m \in \mathcal{M}$ at a vertex $v \in V$. Again, we distinguish between the case that $v$ is on the first path of $m$ and the case where it is on a second path. For the first, we have $\big(m \wedge \langle v, \text{FP}(m, v)\rangle \wedge free(e, m)\big) \implies \text{SP}(m, v)$. For the second, recall that there is no first-choice edge leaving $v$ for message $m$, so the rule is simpler: $\big(m \wedge free(e, m)\big) \implies$ SP$(m, v)$. □

### *Outcomes*

Given a schedule $S$, a protocol Prot, and a fault sequence $F = T_0, \ldots, T_{t-1}$, there is a unique *outcome* to the network, which we denote by $out(S, \text{Prot}, F)$. Intuitively, the outcome is a sequence of *snapshots* of the system at each time point. Each snapshot includes the positions of all the messages. The value of an outcome $out(S, \text{Prot}, F)$, denoted $val(S, \text{Prot}, F)$, is the number of messages that arrive at their destination by the timeout $t$.

Formally, we refer to the snapshots as *configurations*, so we have $out(S, \text{Prot}, F) = C_0, C_1, \ldots, C_t$. A configuration includes the positions of all the messages in the network, thus it is a set of the form $C_i = \{\langle m, v\rangle : m \in \mathcal{M}$ and $v \in V\}$. Having $\langle m, v\rangle \in C_i$ means that message $m$ is at vertex $v$ at time $i$. In particular, for every $m \in \mathcal{M}$ there is a unique $v \in V$ such that the pair $\langle m, v\rangle$ is in $C_i$. In the initial configuration, the messages are at their origin, thus $C_0 = \{\langle m, s(m)\rangle : m \in \mathcal{M}\}$. The other configurations are defined inductively as follows. Consider a time point $0 \leqslant i \leqslant t-1$. Recall that the edges that crashed at time $i$ are $T_i$. For $v \in V$, let $M(v, C_i) \subseteq \mathcal{M}$ be the messages that are in $v$'s queue in $C_i$, thus $m \in M(v, C_i)$ iff $\langle m, v\rangle \in C_i$. Consider a message $m \in M(v, C_i)$. Recall that $\text{Prot}^m_{S,v}$ is a collection of forwarding rules for $m$ at $v$. Let $\varphi^m_i \implies u$ be the unique rule that has $(M(v, C_i), T_i, i) \models_S \varphi^m_i$. Then, $m$ is forwarded from $v$ to $u$, and we have $\langle u, m\rangle \in C_{i+1}$. We use $out^m(S, \text{Prot}, F) = v_0, v_1, \ldots, v_t$ to denote the *positions* of a message $m \in \mathcal{M}$ throughout the outcome $out(S, \text{Prot}, F)$, thus, for $0 \leqslant i \leqslant t$, we have $\langle v_i, m\rangle \in C_i$. The positions $out^m(S, \text{Prot}, F)$ correspond to a sequence $\varphi^m_1, \ldots, \varphi^m_t$ of assertions. For $1 \leqslant i \leqslant t$, the assertion $\varphi^m_i$ is the unique assertion that is satisfied for $m$ at time $i$. So, message $m$ is forwarded according to the rule $\varphi^m_i \implies u$. The value of $out(S, \text{Prot}, F)$ is the number of messages that arrive at their destination on time, thus $val(S, \text{Prot}, F) = |\{\langle m, t(m)\rangle \in C_t\}|$.

We are interested in finding schedules that have a guarantee on the number of messages that reach their destinations before the timeout, no matter what sequence of faults occur. We formalize this notion as follows.

**Definition 3.1.** *Assuming the switches run the protocol* Prot*, a schedule $S$ is $(k, \ell)$-resistant, if for every fault sequence $F$ with at most $k$ faults, at least $\ell$ messages arrive at their destination by time $t$, thus $val(S, \text{Prot}, F) \geqslant \ell$.*

## 4. DECIDING RESISTANCE

In this section we study the problem of deciding whether a given schedule is $(k, \ell)$-resistant. Apart from its practical importance, it is an important ingredient in the solution of the more interesting problem of synthesizing a resistant schedule, which we study in the next section. Recall that given a schedule, a protocol, and a fault sequence, there is a unique outcome for the network. Its value is the number of messages that arrive before the timeout. In order to decide whether a given schedule is $(k, \ell)$-resistant, we search for a fault sequence that *witnesses* its non-resistance. We construct an SMT program whose solutions correspond to such witnesses, thus the program is satisfiable iff the given schedule is not $(k, \ell)$-resistant.

Before solving the problem, we illustrate its intricacy in the following example.

**Example 4.1.** One might suspect that when using a simple protocol as the ones described in the introduction, it suffices to consider *immediate* crashes, namely fault sequences in which edges only crash at time 0. In the do-nothing protocol, this is indeed the case: if there is a fault sequence with $k$ crashes that causes $\gamma < \ell$ messages to arrive on time, then having the crashes happen at time 0, causes at most $\gamma$ messages to arrive on time. However, we show that for the two-path protocol this is not the case. That is, we describe a setting in which $k$ crashes occur. If they all occur at time 0, at least $\ell$ messages arrive on time. However, if we allow crashes at a later time, less than $\ell$ messages arrive on time.

Consider the network that is depicted in Figure 1. Let $\mathcal{M} = \{m_1, m_2, m_3, m_4\}$, where the source of all messages is $s$ and the target for message $m_i$ is $u_i$, for $1 \leqslant i \leqslant 4$. The order of the messages is $m_1 \prec m_2 \prec m_3 \prec m_4$. The first path of $m_i$, for $i = 1, 2, 3$, is $s, a, b, u_i$, and for $m_4$ it is $s, a, u_4$. Thus, all messages use the edge $e$ and the paths are disjoint after using it. We consider a schedule that schedules $m_i$ on $e$ at time $i - 1$, for $1 \leqslant i \leqslant 4$, and

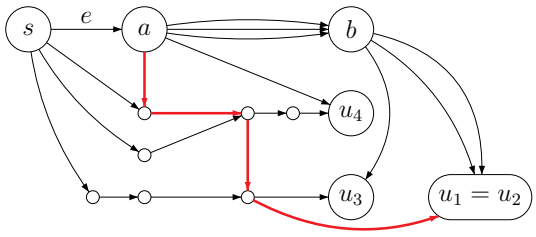


**Figure 1: A network in which the switches run the two-path protocol and immediate crashes are less damaging than non-immediate crashes.**

the other edges on the first path at the immediate time afterwards. So, if there are no crashes, message $m_1$ traverses $e$ at time 0, the edge $\langle a, b\rangle$ at time 1, and the edge $\langle b, u_1\rangle$ at time 2. We set the timeout at 5, and clearly all messages meet it if there are no crashes.

We describe the second paths. From $a$, only message $m_1$ has a second path and it is depicted in red. The other messages have a second path of length 4 from $s$ to their target vertex. Each second path traverses one red edge. Let $\ell = 2$ and $k = 2$. It is not hard to see that every fault sequence with immediate crashes makes at most 2 messages miss the timeout. However, consider the fault sequence in which $\langle a, b\rangle$ and $e$ crash at time 1. At time 0, message $m_1$ traverses $e$ and the other messages wait in $s$. Then, the crashes cause $m_1$ to use the red path and the other messages to use their second path from $s$. Let $i = 2, 3, 4$. Message $m_i$ reaches the red edge on its second path at the same time as message $m_1$. Message $m_4$ reaches the red edge on its path at time 2, message $m_3$ at time 3, and message $m_2$ at time 4. Since $m_1 \prec m_i$, it has precedence over the other message and it traverses the red edge before them. Thus, messages $m_2, m_3$, and $m_4$ reach their target at time 6, thereby missing the timeout. □

We proceed to describe our solution for the problem.

**Theorem 4.1.** *Consider a network $\mathcal{N} = \langle V, E\rangle$, a set of messages $\mathcal{M}$, a schedule $S$, a protocol* Prot*, a timeout $t$, and values $k$ and $\ell$. We construct an SMT program that has a satisfying assignment iff $S$ is not $(k, \ell)$-resistant, thus there is a fault sequence $F$ that makes at most $k$ crashes and for which $val(S, \text{Prot}, F) < \ell$.*

**Proof:** Recall that a configuration is a snapshot of the network at a certain time point, and that $out(S, \text{Prot}, F)$ is a sequence of $t + 1$ configurations. The variables in the SMT program consist of two types of variables. The first type are *crash variables* of the form $x_{e,i}$, for $e \in E$ and $0 \leqslant i < t$. We add constraints so that a truth assignment $f$ to the variables that satisfies these constraints, naturally corresponds to a fault sequence $F_f = T_0, \ldots, T_{t-1}$, which is defined by $T_i = \{e \in E : f(x_{e,i}) = \texttt{tt}\}$, for $0 \leqslant i < t$. The second type of variables are *configuration variables* of the form $x_{m,v,i}$, for $m \in \mathcal{M}$, $v \in V$ and $0 \leqslant i \leqslant t$. Again, we add constraints, so that a truth assignment $f$ to these variables simulates $out(S, \text{Prot}, F_f)$. Namely, we have $f(x_{m,v,i}) = \texttt{tt}$ iff the position of $m$ at time $i$ is $v$ in the outcome $out(S, \text{Prot}, F_f)$. This allows us to count the number of messages that arrive on time in $out(S, \text{Prot}, F_f)$ and require that it less than $\ell$ (recall that we want to show that $S$ is not $(k, \ell)$-resistant).

We describe the program in more detail. There are two constraints on the crash variables: (1) If an edge crashes, it stays down: $\forall e \in E$ and $1 \leqslant i < t$, we have $x_{e,i} \implies x_{e,i+1}$, and (2) there are at most $k$ faults: $\sum_{e \in E} x_{e,t} \leqslant k$.

We describe the constraints on the configuration variables. Recall that they allow us to simulate the outcome of the network, given the fault sequence that the crash variables correspond to. For a message $m \in \mathcal{M}$, a vertex $v \in V$, and a time $0 \leqslant i \leqslant t$, we use the proposition $Uniq(m, v, i)$ to state that $m$ is on $v$ and on no other vertex at time $i$, thus $Uniq(m, v, i) = x_{m,v,i} \wedge \bigwedge_{u \neq v} \neg x_{m,u,i}$. We have constraints that ensure that messages start at their origin: for every $m \in \mathcal{M}$, we have $Uniq(m, s(m), 0)$.

Next, we add constraints to match the forwarding rules in Prot. Consider a vertex $v \in V$ and a message $m \in \mathcal{M}$. Recall that a forwarding rule is of the form $\varphi \implies u$, where $\varphi$ is a propositional assertion and $u \in V$ is the target vertex of the rule. If $\varphi$ is satisfied and $m$ is at $v$, then it is forwarded to $u$ in the next time point. We have $t$ constraints that correspond to a rule $\varphi \implies u$. Consider $0 \leqslant i \leqslant t - 1$. The $i$-th constraint is of the form $\psi_i \rightarrow Uniq(m, u, i+1)$, thus if $f$ satisfies $\psi_i$, then $f(x_{m,u,i+1}) = \texttt{tt}$, meaning that $m$ is on $u$ at time $i + 1$. We define $\psi_i$ inductively on the structure of $\varphi$. The base cases are: If $\varphi = m'$, then $\psi_i = x_{m',v,i}$, if $\varphi = e$, then $\psi_i = x_{e,i}$, and if $\varphi = \big(S(e, \$) = \mu\big)$, for $\mu \in \mathcal{M} \cup \{\perp\}$, then $\psi_i = \texttt{tt}$ if $S(e, i) = \mu$, and $\psi_i = \texttt{ff}$ otherwise. The two inductive cases are as expected. To conclude the description of the program, we require that less than $\ell$ messages arrive at their destination before the timeout, thus we add the constraint $\sum_{m \in \mathcal{M}} x_{m,t(m),t} < \ell$.

The following observation connects the program we describe above with outcomes of the network. The correctness of the program immediately follows from it.

**Observation 4.2.** *Consider a truth assignment $f$ to the variables that satisfies all the constraints, and let $F_f$ be the corresponding fault sequence. Consider the sequence of configurations $C_f = C_0, C_1, \ldots, C_t$ that is defined by $\langle m, v\rangle \in C_i$ iff $f(x_{m,v,i}) = \texttt{tt}$. Then, $out(S, \text{Prot}, F) = C_f$.* ■

**Example 4.2.** Consider a network with two messages $m_1$ and $m_2$ having $m_1 \prec m_2$. The network includes a vertex $v$ that has three outgoing edges $e_1, e_2$, and $e_3$. Let $e_3 = \langle v, u\rangle$. The first path for $m_1$ traverses $e_1$ and the first path for $m_2$ traverses $e_2$. Both their second paths from $v$ traverse $e_3$. Thus, there is a rule at $v$ for $m_1$ that states that if $m_1$ is on $v$ and $e_1$ crashes, forward $m_1$ to $u$ if it is free: $m_1 \wedge e_1 \wedge free(e_3, m_3) \implies u$. A similar rule is at $v$ for $m_2$. We write these rules as constraints at time $0 \leqslant i < t$. Recall that an edge is free for a message $m$ if no message is scheduled on it, and if no message $m' \prec m$ tries to use it. Since $e_3$ is not on a first path, a message is never scheduled on it. For $m_1$, there is no preceding message, so $e_3$ is always free for it, and the constraint that matches the rule is: $x_{m_1,v,i} \wedge x_{e_1,i} \rightarrow x_{m_1,u,i+1}$. For $m_2$, the edge $e_3$ is free when $m_1$ does not try to use it. First, $m_1$ uses $e_3$ when $m_1$ is on $v$ and its first choice edge $e_1$ has crashed: $x_{m_1,v,i} \wedge x_{e_1,i}$. Thus, the forwarding rule for $m_2$ is $x_{m_2,v,i} \wedge x_{e_3,i} \wedge \neg(x_{m_1,v,i} \wedge x_{e_1,i}) \implies x_{m_2,u,i+1}$. □

**Remark 4.3.** It is possible to devise a different program than the one in Theorem 4.1 by defining a variable for every configuration. On one hand, this allows us to settle for a SAT program rather than an SMT one. But, on the other hand, it results in an explosion in the number of variables, which makes the approach impractical even for very small examples.

**Remark 4.4.** The program we devise in Theorem 4.1 simulates the forwarding of messages throughout the network. This simulation enables us to verify other forms of resistance of schedules. For example, assigning different importance to messages (e.g., critical vs. non-critical) is common in practice [14]. We can easily adapt the constraint that at least $\ell$ messages arrive, to a (SAT) constraint that requires all critical messages to arrive and at least $\ell$ non-critical messages arrive.

# 5. FINDING A RESISTANT SCHEDULE

In this section we study the problem of finding a $(k, \ell)$-resistant schedule, which is the main problem that we intend to solve in this paper. Recall that such a schedule guarantees that if at most $k$ edges crash, then, assuming that the switches run a fixed protocol, the number of messages that arrive on time is at least $\ell$.

We devise a CEGAR-like procedure to find a $(k, \ell)$-resistant schedule. Let $P$ be the program that is described in Theorem 3.1 to find *a* schedule. We run this program to find an initial schedule $S$. We then check if $S$ is $(k, \ell)$-resistant by running the SMT program that is described in Theorem 4.1. If the second program returns UNSAT, then $S$ is a $(k, \ell)$-resistant schedule and we return it. Otherwise, the program returns a "counterexample", which is a fault sequence $F$ that witnesses the fact that $S$ is not resistant, thus less than $\ell$ message arrive when the faults $F$ occur, and we have $val(S, \text{PROT}, F) < \ell$. We generate a constraint using $F$, add it to $P$, and repeat the process. The algorithm is described in detail below, where the crux of the approach, namely the method GENERATECONSTRAINT, is described in the next section.

**Input:** A network $\mathcal{N}$, a set of messages $\mathcal{M}$, a protocol PROT, a timeout $t$, and $\ell$ and $k$.

**Output:** A $(k, \ell)$-resistant schedule or "False" if no such schedule exists.

Let $P$ be the program described in Theorem 3.1 to find *a* schedule for $\mathcal{M}$ in $\mathcal{N}$.
**while** true **do**
  **if** UNSAT($P$) **then**
    Return *False*
  Let $S$ be the schedule that corresponds to a model of $P$.
  **if** $S$ is $(k, \ell)$-resistant **then**
    Return $S$.
  Let $T_1, \ldots, T_t$ be a sequence of edge crashes for which less than $\ell$ messages arrive.
  $\psi \leftarrow$ GENERATECONSTRAINT$(T_1, \ldots, T_t)$
  Add $\neg\psi$ to $P$ and solve $P$.
**end while**

## 5.1 Generating a Constraint

We describe how to generate a constraint from a given fault sequence. Note that by adding a constraint to the program we eliminate the schedules that do not satisfy it. We maintain soundness in that we add a constraint that only eliminates schedules that are not resistant.

Recall that the SMT program we construct in Theorem 3.1 finds a schedule. We refer to the variables in this program as *schedule variables* and they are of the form $x_{e,m,i}$, for $e \in E$, $m \in \mathcal{M}$, and $1 \leqslant i \leqslant t$. Let $X$ be the set of all schedule variables. The constraints in the program guarantee that there is a one-to-one correspondence between schedules and truth assignments to $X$ that satisfy the constraints. Such an assignment $f : X \to \{\texttt{tt}, \texttt{ff}\}$ corresponds to the schedule that schedules a message $m$ on $e$ at time $i$ iff $f(x_{e,m,i}) = \texttt{tt}$. We sometimes abuse notation and refer to a schedule that satisfies constraints over $X$.

Consider a schedule $S$ that is not $(k, \ell)$-resistant, and let $F = T_0, \ldots, T_{t-1}$ be a fault sequence that witnesses this fact. Recall that $out(S, \text{PROT}, F)$ is a sequence $C_0, C_1, \ldots, C_t$ of configurations. Consider a message $m \in \mathcal{M}$. Recall that we denote by $out^m(S, \text{PROT}, F)$ the sequence of positions of $m$ in each configuration $C_i$, where $0 \leqslant i \leqslant t$. A sequence of assertions $\varphi_1^m, \ldots, \varphi_t^m$ corresponds to $out^m(S, \text{PROT}, F)$, where $\varphi_i^m$ is the assertion of the rule according to which $m$ is forwarded at time $i$, for $0 \leqslant i < t$.

We describe the intuition of the construction. For $m \in \mathcal{M}$ and $1 \leqslant i \leqslant t$, we construct a constraint $\psi_i^m$ from $\varphi_i^m$, where $\psi_i^m$ is over schedule variables, thus it constraints schedules. Consider a schedule $S'$ that satisfies all of these constraints. We define the constraints so that, assuming the sequence of crashes that occur are $F$, then the outcome of the schedule $S'$ matches that of the non-resistant schedule $S$. It follows that $S'$ is not resistant. The argument is inductive. For the base case, recall that all outcomes start from the same initial configuration. Assume the two outcomes match upto time $0 \leqslant i < t$. Forwarding depends on three parameters: the crashes, the protocol, and the schedule. The first two match in the two outcomes and the fact that $S'$ satisfies the $\psi_i^m$ constraints, for every $m \in \mathcal{M}$, implies that the schedules $S$ and $S'$ forward in the same manner. Thus, the $(i+1)$-th configuration in the two outcomes also matches. It follows that every schedule that satisfies all the $\psi_i^m$ constraints is not resistant, thus we add to the SMT program the constraint $\neg\psi = \neg \bigwedge_{m \in \mathcal{M},\ 0 \leqslant i < t} \psi_i^m$.

We define $\psi_i^m$ formally. Recall that $m$ is at vertex $v_i$ at time $i$, and $\varphi_i^m$ is a rule at that vertex. We define $\psi_i^m$ inductively on the structure of $\varphi_i^m$. If $\varphi_i^m = m'$, then $\psi_i^m = \texttt{tt}$ if message $m' \in \mathcal{M}$ is in $v_i$'s queue at time $i$, thus $\langle m', v_i \rangle \in C_i$, and otherwise $\psi_i^m = \texttt{ff}$. If $\varphi_i^m = e$, then $\psi_i^m = \texttt{tt}$ if edge $e \in E$ has crashed at time $i$, thus $e \in T_i$, and otherwise $\psi_i^m = \texttt{ff}$. If $\varphi_i^m = \big(S(e, \$) = m'\big)$, then we define $\psi_i^m = x_{e,m',i}$. The two inductive cases are as expected.

**Example 5.1.** Consider the setting that is described in Example 3.2: the network that is depicted in Figure 1, there are four messages $m_1, m_2, m_3$, and $m_4$, and the first edge on their first path is $e$. Consider the schedule $S$ that is described in Example 3.2, which has $S(e, i-1) = m_i$, for $i = 1, 2, 3, 4$. For the timeout 5, recall that $S$ is not $(2, 2)$-resistant. The witnessing fault sequence has two crashes at time 1: the edge $e$ and the edge $\langle a, b \rangle$ that $m_1$ uses. In the first configuration of the counterexample, all the messages are at their origin. Since no edges crash at time 0, forwarding is done according to the schedule, thus message $m_1$ traverses $e$, and the other messages wait for their turn at $s$. Thus, $m_1$ follows the rule $\varphi_0^{m_1} \implies a$, where $\varphi_0^{m_1} = \big(S(e, \$) = m_1\big) \wedge \neg e$. Since $e \notin T_0$, we have $\psi_0^{m_1} = x_{m_1,e,0} \wedge \texttt{tt}$, which we shorten to $\psi_0^{m_1} = x_{m_1,e,0}$. Similarly, message $m_i$, for $i = 2, 3, 4$, stays in $s$ according to the rule $\neg\big(S(e, \$) = m_i\big) \wedge \neg e \implies s$, thus, after shortening, we have $\psi_0^{m_i} = \neg x_{m_i,e,0}$. At time 1, the edges crash and the messages enter their second paths. Since the edges on the second path do not participate in first choice paths, there are no constraints on the schedule, and all the constraints can be shortened to $\texttt{tt}$. Thus, the constraint we add to the program is essentially $\neg x_{m_1,e,0}$. The meaning of this constraint is that all the schedules in which $m_1$ is scheduled at time 0 on $e$ are not $(2, 2)$-resistant, and we eliminate them all in the first iteration of the CEGAR loop. □

We formally prove that the constraint we add is sound.

**Lemma 5.1.** *Let $S$ be a schedule that is not $(k, \ell)$-resistant. Let $F$ be a fault sequence that witnesses its non-resistance, and let $\psi = \bigwedge_{m \in \mathcal{M},\ 0 \leqslant i < t} \psi_i^m$ be the generated constraint from $F$. Let $S'$ be a schedule that satisfies $\psi$. Then, $out(S, \text{PROT}, F) = out(S', \text{PROT}, F)$, and in particular $S'$ is not $(k, \ell)$-resistant.*

**Proof:** Let $out(S, \text{PROT}, F) = C_0, \ldots C_t$ and $out(S', \text{PROT}, F) = C'_0, \ldots, C'_t$. We prove by induction on time $0 \leqslant i \leqslant t$ that $C_i = C'_i$. For the base case, all messages start from their origin, thus $C_0 = C'_0$. For the inductive step, we assume that $C_i = C'_i$, and we prove that $C_{i+1} = C'_{i+1}$. Consider a message $m \in \mathcal{M}$ and let its position in the $i$-th configuration be $v \in V$, thus $\langle m, v \rangle \in C_i$. Recall that $M(v, C_i) \subseteq \mathcal{M}$ is the set of messages in $v$'s queue in $C_i$. Let $\varphi_i^m$ be the assertion according to which $m$ is forwarded in $out(S, \text{PROT}, F)$, thus $\big(M(v, C_i), T_i, i\big) \models_S \varphi_i^m$. We prove

that $m$ is also forwarded according to $\varphi_i^m$ in $out(S', \text{PROT}, F)$, thus $\big(M(v, C'_i), T_i, i\big) \models_{S'} \varphi_i^m$. The proof is by induction on the structure of $\varphi_i^m$. The base cases in which $\varphi_i^m = e$ and $\varphi_i^m = m$ are immediate as we have $M(v, C_i) = M(v, C'_i)$. The last case is $\varphi_i^m = \big(S(e, \$) = m\big)$. Thus, we have $\psi_i^m = x_{e,m,i}$. Now, $S$ models $\varphi_i^m$ iff $S' \models \psi_i^m$. The inductive cases are immediate, and we are done. ∎

Completeness of the CEGAR algorithm follows from the fact that all correct schedules are solutions for the initial program $P$. Thus, we have the following.

**Theorem 5.2.** *The CEGAR algorithm is sound and complete.*

## 5.2 Improving the constraint

In the previous section we describe how to generate a constraint $\psi$ from a given fault sequence $F$ that witnesses the non-resistance of a schedule $S$. We showed that if a schedule $S'$ satisfies $\psi$, then the fault sequence $F$ that witnesses the non-resistance of $S$ also witnesses the non-resistance of $S'$. Specifically, we showed that assuming the sequence of crashes $F$ occurs, then the outcomes of $S$ and $S'$ coincide. We added $\neg\psi$ to our program, thus we eliminate all the schedules that do not satisfy it. Since we want to eliminate as many schedules as possible in an iteration, we would like the constraint we add to be as strong as possible. We suggest an optimization that attempts to weaken $\psi$, thus strengthening $\neg\psi$.

We describe the approach. Let $out(S, \text{PROT}, F) = C_0, \ldots, C_t$. Recall that $\psi$ is of the form $\psi = \bigwedge_{1 \leqslant j \leqslant t} \psi_j$. Let $\psi^i = \bigwedge_{1 \leqslant j \leqslant i} \psi_j$, for $1 \leqslant i \leqslant t$, and $\psi^0 = \texttt{tt}$. Using the same argument as in Lemma 5.1, we can show that if a schedule $S'$ satisfies $\psi^i$ and the fault sequence $F$ occurs, then the outcome following $S'$ *shares* a prefix of length $i$ with the outcome of $S$, thus $out(S', \text{PROT}, F) = C_0, C_1, \ldots, C_i, C'_{i+1}, \ldots, C'_t$.

Intuitively, we check if it is possible to "recover" from the $i$-th configuration, for $0 \leqslant i \leqslant t$. We fix the fault sequence $F$, which can be thought of as knowing the sequence of failures in advance. We check whether there is a schedule $S'$ whose outcome coincides with the outcome of $S$ in the first $i$ configurations, and is "resistant" to the specific sequence $F$. That is, even though its outcome reaches the configuration $C_i$, it still manages to deliver at least $\ell$ messages on time, thus $val(S', \text{PROT}, F) \geqslant \ell$. Note that for $i = t$, there is no such schedule. On the other hand, if it is not possible to recover from configuration $C_0$, then there is no $(k, \ell)$-resistant schedule[4]. Formally, we have the following.

**Observation 5.3.** *If it is not possible to recover from $C_i$ and a schedule $S'$ satisfies $\psi^i$, then $S'$ is not a $(k, \ell)$-resistant schedule and $F$ witnesses this fact.*

So, rather than adding the constraint $\neg\psi$, we look for the minimal $i$ such that it is not possible to recover from $C_i$, and add $\neg\psi^i$ to the program. To find this minimal $i$, we construct the following program.

**Theorem 5.4.** *Consider a schedule $S$ that is not $(k, \ell)$-resistant, a witnessing fault sequence $F$, and an index $0 \leqslant i \leqslant t$. We construct an SMT program that is satisfiable iff it is possible to recover from the $i$-th configuation, thus there is a schedule $S'$ that satisfies $\psi^i$ and has $val(S', \text{PROT}, F) \geqslant \ell$.*

**Proof:** The program is a combination of the program to find a schedule in Theorem 3.1 and the program that finds a fault sequence in Theorem 4.1. The variables of the program include schedule variables of the form $x_{e,m,j}$, for $e \in E$, $m \in \mathcal{M}$, and $i \leqslant j \leqslant t-1$, and configuration variables of the form $x_{m,v,i}$, for $m \in \mathcal{M}$, $v \in V$ and $i \leqslant j \leqslant t$. The constraints on the schedule variables are as in Theorem 3.1 apart from the constraint that requires that all messages arrive on time, which we relax to require that at least $\ell$ messages arrive on time.

As in Theorem 4.1, we use the configuration variables to simulate the outcome. For a message $m \in \mathcal{M}$, a vertex $v \in V$, and a time $i \leqslant j \leqslant t$, recall that the proposition $Uniq(m, v, i)$ states that $m$ is on $v$ and on no other vertex at time $j$. We add constraints that ensure that messages start at their position in $C_i$: for every $m \in \mathcal{M}$ and $v \in V$ such that $\langle m, v\rangle \in C_i$, we have $Uniq(m, v, i)$.

Next, we add constraints to match the forwarding rules in PROT. Consider a vertex $v \in V$ and a message $m \in \mathcal{M}$. Recall that a forwarding rule is of the form $\varphi \implies u$, where $\varphi$ is a propositional assertion and $u \in V$ is the target vertex of the rule. If $\varphi$ is satisfied and $m$ is at $v$, then it is forwarded to $u$ in the next time point. We have $t - (i + 1)$ constraints that correspond to a rule $\varphi \implies u$. Consider $i \leqslant j \leqslant t - 1$. The $j$-th constraint is of the form $\psi_j \rightarrow Uniq(m, u, j + 1)$, thus if $f$ satisfies $\psi_j$, then $f(x_{m,u,j+1}) = \texttt{tt}$, meaning that $m$ is on $u$ at time $j + 1$. The difference between the definition of the constraint here and that in Theorem 4.1 is in the inductive definition of $\psi_j$. There, the "unknown" is the fault sequence whereas the schedule was known, and here the situation is opposite. The base cases are: If $\varphi = m'$, then $\psi_j = x_{m',v,j}$, if $\varphi = e$, then $\psi_i = \texttt{tt}$ iff $e \in T_j$, and if $\varphi = \big(S(e, \$) = m\big)$, for $m \in \mathcal{M}$, then $\psi_j = x_{e,m,i}$, and if $S(e, i) = \perp$, then $\psi_j = \neg \bigwedge_m x_{e,m,j}$. The two inductive cases are as expected. To conclude the description of the program, recall that we check whether it is possible to recover from $C_i$, thus we require that at least $\ell$ messages arrive at their destination before the timeout, and we add the constraint $\sum_{m \in \mathcal{M}} x_{m,t(m),t} \geqslant \ell$. ∎

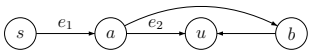


**Figure 2: A network in which the optimization can strengthen the added constraint.**

**Example 5.2.** Consider the network that is depicted in Figure 2. There are two messages $m_1$ and $m_2$ with source $s$ and target $u$. Only $m_2$ has a second choice path from $a$, and the path proceeds through $b$. Consider the schedule $S$ in which $m_1$ is scheduled on $e$ at time 0 followed by $m_2$ at time 1. Let $\ell = 1$, $k = 1$, and $t = 3$. Consider the fault sequence in which $e_2$ crashes at time 0. We claim that $F$ witnesses the non-resistance of $S$. Indeed, since $m_1$ has no second path from $a$, it gets stuck there, thus the positions of $m_1$ in $out(S, \text{PROT}, F)$ are $s, a, a, a$. Recall that $m_1$ uses $e_1$ first, so the positions of $m_2$ are $s, s, a, b$, and it also misses the timeout. The constraint corresponding to the rule according to which $m_1$ is forwarded is $\psi_0^{m_1} = x_{e_1,m_1,0}$, and the constraint according to which $m_2$ is forwarded at time 1 is $\psi_1^{m_2} = x_{e_1,m_2,1}$. Thus, $\psi$ is essentially $\psi_0^{m_1} \wedge \psi_1^{m_2}$. Note that it is not possible to recover from the configuration $C_1$. On the other hand, it is possible to recover from the configuration $C_0$. Indeed, if $m_2$ is scheduled on $e_1$ first, then if $e_2$ crashes at time 0, $m_2$ will still reach $u$ by time 3. Thus, we add to the program the constraint $\neg\psi^1 = \neg\psi_0^{m_1}$. □

# 6. EVALUATION

We have implemented the CEGAR loop for finding a $(k, \ell)$-resistant schedule, which is described in Section 5, assuming that the switches run the two-path protocol. Our implementation is in

[4]In the sabotage-game terminology, the fault sequence $F$ is a winning strategy for the saboteur.

Python and relies on Z3 [13] as an SMT solver. We ran our experiments on a personal computer, Intel Core i5 quad core 1.75 GHz processor, with 8 GB memory.

**Network and path generation** We evaluate the algorithm on networks that were generated randomly by the Python library Networkx [15]. The networks are generated in the following manner. We fix the number of vertices, edges, and messages. We generate a random directed graph. Once we have a graph, we randomly select a source and a target for each message, while guaranteeing that these are different nodes. We refer to a graph and messages with the source and the target vertices, as the *setting*. The computations on a setting are all deterministic.

As a last step before finding a schedule, we find first- and second-paths for each message. We note that the problem of finding "good" paths is a crucial component of the protocol. It is a challenging problem both theoretically and practically, and it is out of the scope of this paper. We now describe the path-finding algorithm we use. We start with a simple approach. Consider a message $m \in \mathcal{M}$. Recall that $s(m)$ and $t(m)$ denote the source vertex and the target vertex of message $m$, respectively. The first path is the shortest path between $s(m)$ and $t(m)$. To find the second path, we remove the edges that the first path traverses. For each vertex $v$ on the first path, we set the second path from $v$ to be the shortest path from $v$ to $t(m)$ in the new graph, if it exists. The problem with this simple approach is that the second paths are of similar lengths for the different messages, making it hard to evaluate the CEGAR loop. Thus, we improve the approach by, intuitively, having messages avoid heavily loaded edges when selecting the two paths. Technically, the graph is now weighted. The weight of an edge refers to the number of times it appears in a first or second path. We construct the weights in the graph incrementally. The weights start from 0. We order the messages arbitrarily and choose them one by one. When a message is chosen, it selects its first and second paths similarly to the above (only running Dijkstra's algorithm rather than a BFS for finding shortest paths). The weight of every edge on the first and second path is incremented by 1.

**Execution time measurements** We elaborate on the pseudo-code that is described in Section 5. An execution starts with an initialization phase (or *Setup*), in which the method $\exists$SCHED, implementing the program described in Theorem 3.1, is called; followed by the iterative phase, which we refer to as the CEGAR loop. Method $\exists$SCHED returns the initial schedule to be used in the CEGAR loop. An iteration starts with a call to the method ISRESIST, which is an implementation of the program that is described in Theorem 4.1. Method ISRESIST$(S, k, \ell)$ returns $\texttt{tt}$ and concludes the iterations if schedule $S$ is $(k, \ell)$-resistant, or otherwise it returns a fault sequence that witnesses the non-resistance of $S$. If the schedule is not $(k, \ell)$-resistant, the second method that is run in the CEGAR iteration is $\exists$SCHEDGIVCONF, which implements the optimization that is described in Theorem 5.4. A call to $\exists$SCHEDGIVCONF$(F, C_i)$ returns $\texttt{tt}$ if it is possible to recover from the configuration $C_i$, assuming the sequence of faults $F$ occurs. This check is performed for every time point $0 \leqslant i \leqslant t-1$ such that the $i$-constraint is not $\texttt{tt}$. If for some time $i$, it finds that it is not possible to recover from $C_i$ then $\neg\psi^i$ is added to the SMT program. The iteration ends with a call to the SMT solver. If it is SAT, then we generate a schedule for the next iteration, whereas if it is UNSAT, we terminate and return that no $(k, \ell)$-resistant schedule exists.

Our goal in the implementation is to evaluate the performance of our CEGAR loop rather than focusing on scalability. In our experiments we considered small settings (S) with 30 vertices, 50 messages, 40 edges, a timeout of 10, and $\ell = 45$ while considering different values for $k$, and a larger setting (L) with 100 vertices, 150 messages, 200 edges, a timeout of 12, and $\ell = 150$. For each setting, we measured three different aspects: the time needed for the Setup phase, the average time needed to execute one iteration and the average number of iterations required before finding a $(k, \ell)$-resistant schedule. The iteration execution times were measured first without applying the optimization, in what we call the Basic loop, and then with the optimization. Our intention was understanding the contributions in terms of execution time of the different methods of the approach and assess the improvement caused by the optimization.

Table 1 shows the execution time of the Setup phase and of the basic iteration (I-Bas) and the optimized iteration (I-Opt). The values have been averaged over 3-5 executions.

| Size: $k$ | Setup (sec) | I-Bas (sec) | I-Opt (sec) |
|---|---|---|---|
| S: $k = 1$ | 2.03 | 11.3 | 27.17 |
| S: $k = 2$ | 2.04 | 17.64 | 31.5 |
| S: $k = 3$ | 2.07 | 17.42 | 31.08 |
| L: $k = 1$ | 11.92 | 142.18 | 261.91 |

**Table 1: Execution times of the Setup phase and the iterations (without and with optimization) of the CEGAR loop.**

It is interesting to observe that the execution time of one iteration with optimization is sensibly longer than the one of a basic iteration. Nevertheless, considering the overall time needed for finding a $(k, \ell)$-resistant schedule, the optimization yields much better results than the basic loop. In Table 2 we show the average number of iterations till the CEGAR loop terminates when the optimization is used: typically, the CEGAR loop terminates after one or two iterations. If the optimization finds that it is not possible to recover from the first configuration $C_0$, it adds the constraint $\texttt{ff}$ to the program, thereby indicating that no $(k, \ell)$-resistant schedule exists and terminating the CEGAR loop. We find that this generally occurs in the first few iterations. In contrast, when running the CEGAR loop with no optimization, the performance is poor. The run either finds the first schedule to be $(k, \ell)$-resistant or it does not terminate.

| Size: $k$ | Number of iterations |
|---|---|
| S: $k = 1$ | 1.4 |
| S: $k = 2$ | 1.2 |
| S: $k = 3$ | 1.4 |
| S: $k = 4$ | 1 |
| L: $k = 1$ | 1.3 |

**Table 2: The average number of iterations till the CEGAR loop terminates.**

**Additional results** We found the following workflow convenient and useful in practice. We assume the network is typically fixed as well as the number of faults $k$. For a fixed setting, and fixed $k$, we refer to the *best* $\ell$ as the highest $\ell$ such that a $(k, \ell)$-resistant schedule exists and no $(k, \ell+1)$-resistant schedule exists. If the best $\ell$ is too low, the designer can use redundancy and increase $m$. We observe that it is convenient to run the CEGAR loop with $\ell = m$. Since we assume $k > 0$, it is rare that a $(k, m)$-resistant schedule exists. But, an initial schedule $S$ is typically found.

We implemented a method, which we call CALC-$\ell$, that finds the largest $\ell_S$ such that $S$ is $(k, \ell_S)$-resistant. This $\ell_S$ gives us a guess on the best $\ell$. The method we implement is a binary search that performs calls to the method ISRESIST. That is, for $0 \leqslant c \leqslant \ell$, we call ISRESIST$(S, k, c)$. If it returns $\texttt{tt}$, then $S$ is $(k, \ell)$-resistant and we increase $c$, and otherwise we decrease $c$. As mentioned above,

the programs in the binary search differ in exactly one constraint; the constraint that guarantees that less than $\ell$ messages arrive on time. Amortizing the construction helps, but the running time of this method is significant ($\sim 823s$ for S:$k = 3$ and $\sim 9386s$ for L:$k = 1$) mostly due to calls to IsResist that are UNSAT.

We observe one phenomenon that is surprising for us. Often the first schedule is the "best" schedule. That is, if $S$ is the first schedule that is found and its guarantee is $\ell_S$, then $\ell_S$ is often the best $\ell$. To verify that $\ell_S$ is indeed the best $\ell$, we run the CEGAR loop with the same setting and set $\ell = \ell_S + 1$. The reason behind this result remains unknown and will require further investigation.

# 7. CONCLUSION

In this paper, we propose a new method for guaranteeing resilience in TT-scheduling switched networks with link failures. Our approach maintains the assumption that the switches have limited computational power while allowing the network some fault tolerance by having the switches run a simple error recovery protocol.

We introduced the definition of a $(k, \ell)$-resistant schedule, which formalizes a notion of a guarantee of a schedule, given an assumption on the degree of failures in the network. In order to find a $(k, \ell)$-resistant schedule we devised a CEGAR-like approach, which we have implemented. We evaluated the approach and found that our algorithm for generating $(k, \ell)$-resistant schedules can be successfully used for small and moderately sized networks with a few hundred nodes. Our results are promising: the CEGAR loop typically terminates within several iterations.

There are many directions for future work and we list some here. First, we considered one type of failures, namely permanent link crashes. Other relevant fault models for switched networks include omission faults and delayed messages. Our definition of a resistant schedule as well as our solution, does not immediately follow to networks suffering from such failures, and both need to be adapted. Second, we assume that the switches run a fixed protocol. An interesting extension would be to synthesize a protocol that is good for a given network as well as an accompanying schedule. Third, the constraints of our scheduler are based on a simple traffic model, and it is interesting to integrate our approach with schedulers that handle more complex constraints, like mixed-criticality or application constraints. Finally, we would like to investigate how our algorithm scales with even bigger networks possibly by adapting techniques such as segmented scheduling [23], which allow finding TT-schedules in extremely large networks.

# 8. REFERENCES


[1] S. Almagor, U. Boker, and O. Kupferman. Formalizing and reasoning about quality. To appear in J.ACM, 2016.

[2] G. Avni, O. Kupferman, and T. Tamir. Congestion games with multisets of resources and applications in synthesis. In *Proc. 35th FSTTCS*, pp. 365–379, 2015.

[3] G. Bauer and H. Kopetz. Transparent redundancy in the time-triggered architecture. In *Proc. 1st DSN*, pp. 5–13. IEEE, 2000.

[4] A. Biewer, J. Gladigau, and C. Haubelt. Towards Tight Interaction of ASP and SMT Solving for System-Level Decision Making. In *Proc. 27th ARCS*, pp. 1–7. VDE, 2014.

[5] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, G. Hofferek, B. Jobstmann, B. Könighofer, and R. Könighofer. Synthesizing robust systems. *Acta Informatica*, 51(3-4):193–220, 2014.

[6] U. Boker, K. Chatterjee, T. A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *Proc. 26th LICS*, pp. 43–52, 2011.

[7] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, and J. Srba. Infinite runs in weighted timed automata with energy constraints. In *Proc. 6th FORMATS*, pp. 33–47, 2008.

[8] T. Brihaye, G. Geeraerts, A. Haddad, B. Monmege, G. A. Pérez, and G. Renault. Quantitative games under failures. In *Proc. 35th FSTTCS*, pp. 293–306, 2015.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems (2nd ed.). chapter The Primary-backup Approach, pp. 199–216, 1993.

[10] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[11] S. S. Craciunas and R. S. Oliver. SMT-based Task-and Network-level Static Schedule Generation for Time-Triggered Networked Systems. In *Proc. 22nd RTNS*, page 45, 2014.

[12] S. S. Craciunas and R. S. Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Real-Time Systems*, 52(2):161–200, 2016.

[13] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. 14th TACAS*, pp. 337–340, 2008.

[14] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-Triggered Implementations of Mixed-Criticality Automotive Software. In *Proc. DATE*, pp. 1227–1232, 2012.

[15] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proc. 7th SciPy*, pp. 11–15, 2008.

[16] T. A. Henzinger. Two challenges in embedded systems design: predictability and robustness. *Phil. Trans. R. Soc. A*, 366(1881):3727–3736, 2008.

[17] T. A. Henzinger. From boolean to quantitative notions of correctness. In *Proc. 37th POPL*, pp. 157–158, 2010.

[18] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. Static Scheduling of a Time-Triggered Network-on-Chip based on SMT Solving. In *Proc. DATE*, pp. 509–514, 2012.

[19] H. Kopetz, M. Braun, C. Ebner, A. Kruger, D. Millinger, R. Nossal, and A. Schedl. The Design of Large Real-Time Systems: the Time-Triggered Approach. In *Proc. 16th RTSS*, pp. 182–187, 1995.

[20] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, 2015.

[21] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *Proc. 10th NSDI*, pp. 399–412, 2013.

[22] C. Löding and P. Rohde. Solving the sabotage game is pspace-hard. In *Proc. 28th MFCS*, pp. 531–540, 2003.

[23] F. Pozo, W. Steiner, G. Rodriguez-Navas, and H. Hansson. A Decomposition Approach for SMT-based Synthesis for Time-Triggered Networks. In *Proc. 20th ETFA*, pp. 1–8. IEEE, 2015.

[24] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: declarative fault tolerance for software-defined networks. In *Proc. 2nd HotSDN*, pp. 109–114, 2013.

[25] K. Schild and J. Würtz. Scheduling of Time-Triggered Real-Time Systems. *Constraints*, 5(4):335–357, 2000.

[26] W. Steiner. An Evaluation of SMT-based Schedule Synthesis for Time-Triggered Multi-Hop Networks. In *Proc. 31st RTSS*, pp. 375–384. IEEE, 2010.

[27] W. Steiner, F. Bonomi, and H. Kopetz. Towards synchronous deterministic channels for the internet of things. In *Proc. WF-IoT*, pp. 433–436. IEEE, 2014.

[28] J. van Benthem. An essay on sabotage and obstruction. In *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, pp. 268–276, 2005.

[29] Y. Wei and D.-S. Kim. Exploiting real-time switched ethernet for enhanced network recovery scheme in naval combat system. In *Proc. ICTC*, pp. 595–600. IEEE, 2014.