Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

FAST: Acceleration from theory to practice ⋆

Sébastien Bardin[1], **Alain Finkel**[2], **Jérôme Leroux**[3], **Laure Petrucci**[4]

[1] LSL (LIST, CEA), Saclay
e-mail: `sebastien.bardin@cea.fr`

[2] LSV (UMR CNRS 8643, ENS de Cachan), Cachan
e-mail: `finkel@lsv.ens-cachan.fr`

[3] LaBRI (UMR CNRS 5800, ENSEIRB, Université Bordeaux-1), Bordeaux
e-mail: `leroux@labri.fr`

[4] LIPN (UMR CNRS 7030, Université Paris-13), Villetaneuse
e-mail: `laure.petrucci@lipn.univ-paris13.fr`

The date of receipt and acceptance will be inserted by the editor

Abstract. FAST is a tool for the analysis of systems manipulating unbounded integer variables. We check safety properties by computing the reachability set of the system under study. Even if this reachability set is not necessarily recursive, we use innovative techniques, namely symbolic representation, acceleration and circuit selection, to increase convergence. FAST has proved to perform very well on case studies. This paper describes the tool, from the underlying theory to the architecture choices. Finally, FAST capabilities are compared with those of other tools. A range of case studies from the literature is investigated.

Keywords: counter systems, infinite reachability set, symbolic representation, acceleration

1 Introduction

Automatic verification of reactive systems is a major field of research. A popular way of modeling such systems is by means of concurrent automata with shared variables. The automata represent the control structure of the system, while variables encode data. Many classes of such extended automata have been studied, considering variables ranging over integers (counters), real numbers (time), words (queues, stacks) and so on.

The semantics of such an extended automaton is given by a transition system $(\mathcal{C}, \to)$, defined by a set of configurations $\mathcal{C}$ and a transition relation $\to$. A configuration $c \in \mathcal{C}$ is a tuple of control locations (one for each component) and a valuation for each variable of the system. The transition relation $\to$ is a binary relation over the set of configurations. A configuration c' is reachable from a configuration c if and only if $(c, c') \in \to^*$, where $\to^*$ denotes the reflexive and transitive closure of $\to$. The set of configurations reachable from the configuration c_0 is called the reachability set from c_0.

Safety properties are expressed in terms of "safe reachable configurations". They are the most commonly encountered properties in practice, and allow specification of important properties such as the absence of deadlock, capacity overflow and division by zero.

The class of *counter systems*, where variables range over integers, appears to be interesting. From a practical point of view, these systems allow the modeling of, for example, communication protocols [18], multi-thread programs or programs with pointers [8]. From a theoretical view, many well-known classes appear to be encompassed by counter systems, like Minsky machines, Petri nets extended with reset/ inhibitor/ transfer arcs [32, 39], reversal-bounded counter machines [47] and broadcast protocols [33,34].

The counterpart of the expressiveness of counter systems is that only two counters with increment, decrement and test-to-zero can simulate a Turing machine. Then checking even basic safety properties of counter systems is undecidable. Many works have been conducted on identifying decidable subclasses, like Petri nets [60] and reversal-bounded counter machines [46,47]. However few of these results have been implemented, mainly for two reasons. First since each result applies for a restricted subclass, there is no generic method for a large class of counter systems. Second, these algorithms are often inefficient in practice.

⋆ This paper is mainly based on results presented at CAV 2003, TACAS 2004 and ATVA 2005.

1.1 The tool FAST

In this paper, we present the tool FAST [5, 9], designed to check safety properties on counter systems. We made the choice to consider a very large subclass of counter systems, namely linear counter systems, for which checking safety properties is undecidable.

The safety properties are expressed in terms of Presburger constraints over counters. They strictly include the usual reachability properties, expressed in terms of control location or upward closed / convex sets of configurations.

The tool FAST has four main advantages:

- Since linear counter systems and Presburger constraints are very expressive, FAST can be applied to a large spectrum of applications and the tool is not tied to a particular specific case-study.
- Despite the inherent theoretical limitations, a powerful engine based on recently developed techniques (*acceleration, flattening, reduction*) allows FAST to check the correctness of the system in most practical cases.
- FAST design is fully based on a clear theoretical framework (*flat acceleration*). Abilities and limits of the tool are clearly identified: FAST is complete for the class of flattable systems [7]. Moreover since many decidable subclasses of counter systems are flattable, FAST provides a unified verification algorithm for all these classes [56, 57].
- Finally, in case the automatic verification fails, the user can guide the tool using a script language. We think that this is an important feature since termination cannot be guaranteed.

1.2 Theoretical foundations

Symbolic model checking. FAST follows the model checking approach [13, 26], based on the exhaustive exploration of the reachability set. However, since one manipulates potentially infinite sets of configurations, called *regions*, the model checking must be "symbolic". A symbolic representation must support the following operations: (1) post- and/or pre-image computation, (2) union to collect all reachable configurations, (3) inclusion to test for fixpoint. The most popular symbolic representations are based on regular languages: these are quite expressive and automata-theoretical data structures provide well-known and efficient algorithms performing the previous operations. With these ingredients, it becomes possible to launch a fixpoint computation for forward or backward reachability sets (see for example [51]), as exemplified in procedure 1.

procedure REACH1($\mathbf{x}_0$)
input: symbolic configuration $\mathbf{x}_0$.
$\mathbf{x} \leftarrow \mathbf{x}_0$
while $\text{POST}(\mathbf{x}) \not\sqsubseteq \mathbf{x}$ **do**
  $\mathbf{x} \leftarrow \text{POST}(\mathbf{x}) \sqcup \mathbf{x}$
end while
return $\mathbf{x}$

Procedure 1: standard symbolic procedure

Acceleration. In practice, an iterative symbolic reachability set computation similar to the one of procedure 1 will surely fail. A solution to help convergence is to use *acceleration techniques*. Acceleration consists of computing in one step the effect of the transitive closure of a transition or a sequence of transitions.

First ideas of acceleration can be found in the covering tree of Petri nets by Karp and Miller in 1969 [49], extended by Finkel to well-structured transitions systems [36]. The first paper on the acceleration of counter systems is probably due to Boigelot and Wolper in [21], considering functions with increment/ decrement/ reset and convex guards. Since then, lots of work has been achieved in this area, for example [2, 14, 41, 42, 50]. Results of [20, 37, 66] extend those of Wolper and Boigelot to linear functions with Presburger-definable guards.

Flat acceleration framework. An efficient acceleration algorithm is not sufficient to compute the reachability set. The question is how to find out the circuits (sequences of transitions) of the system, whose acceleration will lead to a successful computation of the reachability set. This issue was not clearly treated until we introduced the flat acceleration framework [7]. We proposed the notion of flattening, and showed that flat acceleration computes the reachability set if and only if the system is flattable. Moreover, we designed a complete heuristic for flattable systems, and generic optimizations called reductions. The framework is articulated around four key points: (1) the system under consideration, (2) the symbolic representation, (3) the acceleration algorithm and (4) a heuristic to select circuits to be accelerated.

The tool FAST follows strictly the flat acceleration framework. The systems analyzed (linear counter systems with finite monoid) and the corresponding acceleration algorithms can be found in [37, 66]. The symbolic representation is based upon the automata representation of semi-linear sets (see [23, 67]). The selection heuristic is the one described in [7] with the reduction presented in [37].

Even though the reachability set of a linear counter system is not Presburger definable in general, in practice the systems manipulated are regular enough to have a Presburger definable reachability set. The techniques presented throughout the paper allow for model checking many counter systems (more than forty tests).

Moreover, Leroux and Sutre have shown in [56, 57] that FAST is guaranteed to terminate for many subclasses of counter systems: 2-counters VASS, reversal-bounded counter machines, lossy VASS, BPP, Cyclic Petri nets and other subclasses.

1.3 Other tools for counter systems

The following approaches and tools have been developed to check correctness of counter systems.

Reachability set computation. Tools ALV [25,68], LASH [55] and TREX [3] implement symbolic methods to compute the forward reachability set of counter systems. ALV provides two different symbolic representations for integer vectors: Presburger formula or automata as in FAST. Acceleration is available for the formula-based representation [50], but not for the automata-based representation. The tool is mostly used in backward computation or in approximated forward computation [12]. LASH [55] foundations are close to those of FAST, with similar symbolic representations and acceleration algorithms. The main difference is that LASH does not implement any circuit search and the user has to provide circuits to the tool. TREX [3] follows the same framework but uses rather different technologies. A comparison of ALV, FAST, LASH and TREX is presented in section 8.

Co-reachability set computation. One of the most interesting results for counter systems verification is the computability of the co-reachability set of monotonic VASS: monotonic VASS is a large subclass, efficient symbolic representations have been developed and interesting case studies have been conducted. We can cite the work on covering sharing trees of Delzanno, Raskin and Van Begin [30] and the tool BRAIN by Voronkov and Rybina [64]. These approaches are more specific than the one of FAST: computation is backward only[1], properties are reduced to upward-closed sets and systems are monotonic. For example case-studies of section 9 and section 10 could not have been handled with these tools.

Reachability set approximation. Finally, some approaches relax the exactness of computation to ensure computation termination or at least simpler computational steps. However the superset obtained in the end may not be tight enough to decide the property. We can cite the classic tool HYTECH [1], as well as the *abstract-check and refine* technique of Raskin et al. [44] to compute iteratively covering trees of monotonic Petri nets.

1.4 Contribution

This paper provides both an overview of the main results obtained on FAST and on acceleration of counter systems [4,5,7,9,10,29,37,56,57] as well as some original contributions: an up-to-date description of FAST, an in-depth experimental comparison with similar tools, the verification of the TTP protocol (a short description of this work was proposed in [4]) and the verification of the Capacity Exchange Signaling protocol.

[1] FAST can also be used for backward reachability computations.

1.5 Outline

The sequel of the paper is structured as follows. Each point of the FAST framework is presented in sections 2 to 5: counter systems (section 2), symbolic representation (section 3), acceleration (section 4) and selection heuristic (section 5). After this overview of the theoretical foundations, section 6 presents the tool FAST. Experiments are presented in section 7, comparisons with tools ALV, LASH and TREX can be found in section 8 and two case-studies are developed in sections 9 and 10.

2 Presburger Arithmetic And Counter Systems

2.1 Sets

Given two sets E and F, we denote by $E \cup F$, $E \cap F$, $E \backslash F$ and $E \times F$, the union, the intersection, the difference and the Cartesian product of E and F. The set $E^i, i > 0$, is defined by $E^1 = E$ and $E^{n+1} = E \times E^n$. We write $E \subseteq F$ if E is a subset of F. The empty set is denoted $\emptyset$. Given two sets E, X such that $E \subseteq X$, the complement of E (in X) is denoted by $\bar{E}$ and is defined as $\bar{E} = X \backslash E$. The cardinal of a finite set X is written $|X|$.

2.2 Relations

A *relation* R between E and F is a subset $R \subseteq E \times F$. We write $x \, R \, x'$ whenever $(x, x') \in R$. The inverse relation of R, written $R^{-1} \subseteq F \times E$, is defined by $x' R^{-1} x$ if and only if $x R x'$. The image of $x \in E$ by R is the set $R(x) \subseteq F$ defined by $R(x) = \{x' \in F | x R x'\}$. The definition is extended to a set $X \subseteq E$ by $R(X) = \{x' \in F | \exists x \in X . x R x'\}$. Given two relations $R_1 \subseteq E \times F$ and $R_2 \subseteq F \times G$, the *composition of* R_1 and R_2, written $R_1 \bullet R_2 \subseteq E \times G$, is defined by: $x (R_1 \bullet R_2) x''$ if $x R_1 x'$ and $x' R_2 x''$ for some $x' \in F$.

A *binary relation* R on E is a relation between E and itself. The *identity relation on* E is the binary relation $Id_E = \{(x, x) | \; x \in E\}$. For R on E, we denote by R^i the relation defined inductively by: R^0 is the identity relation on E and $R^{n+1} = R \bullet R^n$. The *reflexive and transitive closure* of R, denoted R^*, is then defined by $R^* = \bigcup_{n \geq 0} R^n$.

2.3 Numbers and matrices of numbers

Let $\mathbb{Z}$ (resp. $\mathbb{N}$) denote the set of *integers* (resp. *non-negative integers*). We denote by $\mathcal{M}_n(\mathbb{Z})$ (resp. $\mathcal{M}_n(\mathbb{N})$)

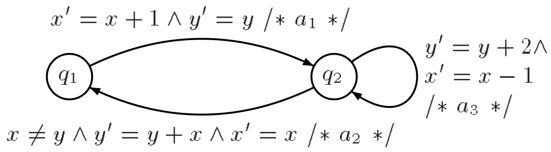


Fig. 1: A simple counter system

the set of square matrices of size n over $\mathbb{Z}$ (resp. $\mathbb{N}$). The identity matrix of size n is denoted 1_n. The max-norm of a matrix (resp. vector), written $||\cdot||_\infty$, is the maximal absolute value appearing in the matrix (resp. vector).

2.4 Presburger arithmetic

Presburger arithmetic [63] is the first order additive theory over the integers $\langle \mathbb{Z}, \leq, + \rangle$. Satisfiability and validity of Presburger arithmetic are both decidable. A *Presburger formula* is denoted by $\phi(\overrightarrow{x})$ where $\overrightarrow{x}$ is a n-dim vector of *free variables* ($\overrightarrow{x}[i]$ is the i-th component of $\overrightarrow{x}$). The set of vectors *defined* by such a formula $\phi(\overrightarrow{x})$, i.e, the set of vectors satisfying ϕ, is denoted by $[\![\phi(\overrightarrow{x})]\!] \subseteq \mathbb{Z}^n$. A set $X \subseteq \mathbb{Z}^n$ is said to be *Presburger-definable* if there exists a Presburger formula $\phi(\overrightarrow{x})$ such that $X = [\![\phi(\overrightarrow{x})]\!]$.

2.5 Counter systems

A *counter system* is a finite control structure (automaton[2]) extended with m integer variables whose values can be modified by actions denoted by a Presburger formula. Fig. 1 gives an example of a counter system.

Definition 1 (Counter system). Let m be a non-negative integer. A *m-dim counter system* S is a tuple $S = (Q, T, m)$, where Q is a finite non empty set of *locations*, and T is a finite set of transitions (q, ϕ, q') where $q, q' \in Q$ and $\phi(\overrightarrow{x}, \overrightarrow{x}')$ is a Presburger formula over $2m$ variables.

Given a transition $t = (q, \phi, q') \in T$, we define the functions α, β and l by $\alpha(t) = q$, $\beta(t) = q'$ and $l(t) = \phi$.

Semantics. As previously mentioned, the semantics of a counter system is given by a transition system $(\mathcal{C}_S, \xrightarrow{T})$. The *set of configurations* $\mathcal{C}_S$ of a counter system S is $Q \times \mathbb{Z}^m$. The transition relation $\xrightarrow{T}$ is defined as follows. The semantics of a transition $t \in T$ is given by the relation $\xrightarrow{t}$ over $\mathcal{C}_S$ defined by: $(q, \overrightarrow{x}) \xrightarrow{t} (q', \overrightarrow{x}')$ if $q = \alpha(t), q' = \beta(t)$ and $(\overrightarrow{x}, \overrightarrow{x}') \in [\![l(t)]\!]$. This definition can be extended to the set T^* of sequences of transitions. Let us denote by ε the empty word. Then $\xrightarrow{\varepsilon} \stackrel{\text{def}}{=} Id_{\mathcal{C}_S}$ and $\xrightarrow{t \cdot \pi} \stackrel{\text{def}}{=} \xrightarrow{t} \bullet \xrightarrow{\pi}$. We also extend $\longrightarrow$ to any language $\mathcal{L} \subseteq T^*$ by $\xrightarrow{\mathcal{L}} \stackrel{\text{def}}{=} \bigcup_{\pi \in \mathcal{L}} \xrightarrow{\pi}$. The definition of $\xrightarrow{T}$ follows directly. The relation $\xrightarrow{T^*}$ is called the *reachability relation.*

[2] In case of multi-component systems, we consider the synchronized product automaton.

Remark 1. The analysis of a counter system with $|Q|$ locations and m counters can always be reduced to the analysis of a system $S' = (\{q'\}, T', m+1)$ with only one location and $m + 1$ variables, by encoding the control structure in a new counter x_Q.

Notation. Whenever S is implicitly known, it is omitted in the notation.

2.6 Reachability problems

For any $X \subseteq \mathcal{C}$ and any $\mathcal{L} \subseteq T^*$, the set $\mathsf{post}(\mathcal{L}, X)$ of configurations reachable from X following sequences of transitions in $\mathcal{L}$ is defined by $\mathsf{post}(\mathcal{L}, X) = (\xrightarrow{\mathcal{L}})(X) = \{x' \in \mathcal{C} |\ \exists x \in X;\ (x, x') \in \xrightarrow{\mathcal{L}}\}$. We focus on two particular sets: the set $\mathsf{post}(T, X)$ of all configurations reachable in one step from X, also denoted by $\mathsf{post}(X)$; and the set $\mathsf{post}(T^*, X)$ of all configurations reachable from X (*the reachability set of* X), also denoted by $\mathsf{post}^*(X)$.

Given an initial set of configurations X_0, checking a safety property P can be done by: (1) computing $\mathsf{post}^*(X_0)$, (2) deciding whether $\mathsf{post}^*(X_0) \subseteq P$ holds or not. We focus here on the reachability set computation, which is the central issue. Since counter systems generalize Minsky machines (counters with increment, decrement and test-to-zero), their reachability sets are not recursive in general. Then the best we can hope for are correct procedures, with no theoretical guarantee of termination but efficient on large subclasses and practical case-studies.

A symmetrical approach is to compute in a backward manner the co-reachability set $\mathsf{pre}^*(\bar{P}) = (\xrightarrow{\mathcal{L}})^{-1}(\bar{P})$ and check that $X_0 \cap \mathsf{pre}^*(\bar{P})$ is empty. In the following we always consider forward computation, but our results can be straightforwardly adapted to backward computation.

3 Automata-Based Symbolic Representation

The symbolic model checking approach was first developed to verify large but still finite-state systems. The key idea is to manipulate sets of states directly through a concise symbolic representation (such as BDDs) rather than manipulate enumerations of concrete states. The approach naturally extends to infinite state verification using more complex symbolic representations, such as automata.

In the case of counter systems, the class of Presburger-definable sets is naturally used as the symbolic representation, since: (1) the union of two Presburger-definable sets is effectively Presburger-definable, (2) assuming the set X is Presburger-definable and S is a counter system, then $\mathsf{post}_S(X)$ is Presburger-definable, (3) we can check if $X \subseteq X'$ for any two Presburger-definable sets X and X'.

3.1 Number Decision Diagrams (ndd)

The efficiency of the algorithms based on Presburger definable sets depends strongly on the symbolic representation used for manipulating these sets. Different techniques [43] and tools have been developed for manipulating *Presburger-definable sets*: by working directly on the Presburger formulas [52] (implemented in Omega [62]), by using semi-linear sets [45] (implemented in Brain [64]), or by using Number Decision Diagrams [23,65] (ndd, implemented in Fast [5], Lash [55] and Mona [53]).

The ndd representation is obtained by remarking that given a *basis* $r \geq 2$ *of decomposition*, an integer, or more generally an integer vector in $\mathbb{N}^m$, can be decomposed into a word over the alphabet $\Sigma_{r,m} = \{0 \ldots r-1\}^m$. Then a "regular" set of integer vectors can be decomposed into a regular language $\mathcal{L} \subseteq \Sigma^*_{r,m}$, and it can be naturally represented by an automaton over $\Sigma_{r,m}$. Such an automaton is called a ndd [23,65]. An example is presented in figure 2. For more detailed information on ndd and automata-theoretic representations of Presburger sets, the reader is referred to [23,65,67].

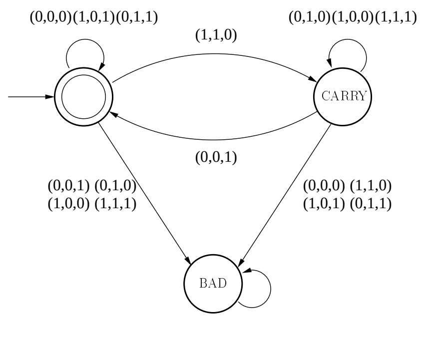


Fig. 2: An automaton to represent $\{(x, y, z)|x + y = z\}$ (least digit first)

This approach is very fruitful since set-theoretical operations correspond to well-known operations on automata (intersection for conjunction, complementation for negation, projection for quantification and so on). Presburger-definable sets and Presburger-definable relations (on $2m$ variables $(\overrightarrow{x}, \overrightarrow{x}'))$ are canonically represented (uniqueness of the minimal form w.r.t. the number of nodes of the ndd). The post- and pre-operations for arbitrary ndd-definable relations are also quite straightforward, since ndd-definable sets are closed by image of such relations.

Automata representations are well-suited for applications that require a lot of boolean manipulations such as model-checking. For these applications, ndd have two crucial advantages over Presburger formula and semi-linear sets. First, a minimization procedure for automata provides a canonical representation for *ndd-definable sets* (a set represented by a ndd). This means that the ndd representing a given set only depends on this set and not on the way it is computed. On the other hand, Presburger formulas and semi-linear sets lack canonicity. As a direct consequence, a set that possesses a simple representation could unfortunately be represented in an unduly complicated way. Second, deciding if a given vector of integers is in a given set can be performed in linear time with ndd, while it is at least *NP-hard* [15,45] with Presburger formula and semi-linear sets.

Remark 2. In practice, in order to decrease the number of output transitions in the ndd, the basis r is set to 2 (binary decomposition) and the alphabet $\{0,1\}^m$ is reduced to $\{0,1\}$ thanks to a serialization [67]. This can be done without any loss of expressivity since Presburger-definable sets are exactly the sets that can be represented by ndd in any basis r of decomposition. People interested in the expressive power of ndd should consult [24].

4 Reachability Computation For Flat Counter Systems

Procedure 1 only terminates on *bounded systems* [7]: any configuration $x \in \mathsf{post}^*(X)$ must be reachable from a configuration $x_0 \in X$ by a path $x_0 \xrightarrow{\pi} x$ where $|\pi|$ is bounded independently of x and x_0. In practice systems are rarely bounded. For example any linear counter system (S, X_0) such that $\mathsf{post}^*(X_0)$ is infinite while X_0 is finite (e.g. a system with a circuit adding one to a counter) is not bounded. A notable exception is the class of monotonic Petri nets, considering backward computation from upward-closed sets of configurations.

Procedure 1 can be improved by using an algorithm POST_STAR that computes from a symbolic representation of X and a regular language $\mathcal{L} \subseteq T^*$, a symbolic representation of $\mathsf{post}(\mathcal{L}, X)$. We are interested in infinite regular languages $\mathcal{L}$, simple enough so that $\mathsf{post}(\mathcal{L}, X)$ can be effectively computed from any set X. Indeed, if $\mathcal{L}$ is finite, then $\mathsf{post}(\mathcal{L}, X)$ can be computed by procedure 1 and POST_STAR does not add any power to procedure 1. On the other hand, $\mathsf{post}(\mathcal{L}, X)$ cannot be computed for an arbitrary $\mathcal{L}$, since $\mathsf{post}(T^*, X)$ is equal to the reachability set which is not recursive. In the sequel, we consider the special case $\mathcal{L} = \sigma^*$ where $\sigma \in T^*$.

It is easy to define a first syntactic restriction to counter systems such that the reachability set can be computed with an improved version of procedure 1 using $\mathsf{post}(\sigma^*, X)$ for some cycles $\sigma \in T^*$. We call *flat counter* *system* [27,38,40] a counter system where, for each location q, there exists at most one elementary circuit in the control graph containing q (see figure 3). Intuitively a flat system has no nested loop. For example, to compute the reachability set of the flat counter system of figure 3, we first iterate t_1, then fire t_3 and finally iterate t_2. Note that such an algorithm (if it exists) goes beyond the standard symbolic procedure because it can discover the set of configurations that are not necessarily reachable by paths of a bounded length.

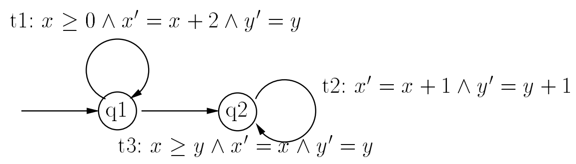


Fig. 3: A flat counter system

4.1 *Presburger linear functions*

The set $\mathsf{post}(\sigma^*, X)$ is not Presburger-definable in general even if X is Presburger-definable. Indeed, this set can be non-recursive since the one-step reachability relation R of a Minsky machine is Presburger-definable and can be encoded by a single loop t such that $[\![l(t)]\!] = R$. Nevertheless, we say that a Presburger-definable binary relation $R \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$ can be *accelerated* if the binary relation R^* is Presburger-definable. In this section, we define a subclass of Presburger-definable binary relations R, both encompassing most of the usually used binary relations $R = [\![l(\sigma)]\!]$ and supporting the effective computation of a Presburger formula encoding R^*.

Definition 2 (Presburger linear function [37]). A *Presburger linear function* is a function $f : \mathbb{Z}^n \to \mathbb{Z}^n$ such that there exists a tuple $\bar{f} = (\varphi, M, \overrightarrow{v})$, where $\varphi(\overrightarrow{x})$ is a Presburger formula over n variables, $M \in \mathcal{M}_n(\mathbb{Z})$ is a square matrix and $\overrightarrow{v} \in \mathbb{Z}^n$ is a vector, such that f is defined over $[\![\varphi]\!]$ by $f(\overrightarrow{x}) = M.\overrightarrow{x} + \overrightarrow{v}$.

Such a tuple $\bar{f} = (\varphi, M, \overrightarrow{v})$ is called a *Presburger linear presentation* of f. The formula φ is the *guard* of $\bar{f}$.

Note that the binary relation $x' = f(x)$ where f is the Presburger linear function defined by $f(x) = 2x$ for any x is not accelerable. In fact, the binary relation $x' \in f^*(x)$ is not Presburger-definable. We introduce the class of Presburger linear presentations with a finite monoid to enforce the accelerability property. The monoid of a Presburger linear presentation $\bar{f} = (\varphi, M, \overrightarrow{v})$ is the multiplicative monoid M^* of the matrix M, i.e. $M^* = \{1_m, M, M^2, \ldots, M^n, \ldots\}$.

Note that when $M = Id_m$ the Presburger presentation $\bar{f} = (\varphi, M, \overrightarrow{v})$ has a finite monoid. In this case, the binary relation $\overrightarrow{x'} \in f^*(\overrightarrow{x})$ is encoded by the following Presburger formula:

$$\exists k \geq 0,\ \overrightarrow{x'} = \overrightarrow{x} + k\overrightarrow{v} \ \wedge\ \forall 0 \leq l < k,\ \varphi(\overrightarrow{x} + l\overrightarrow{v}) \quad (1)$$

In fact, the following theorem holds.

Theorem 1 ([20, 37]). *The binary relation f^* is effectively Presburger-definable for any Presburger linear presentation $\bar{f}$ with a finite monoid.*

Proof. (Sketch) We reduce the proof to the straightforward case $M = Id_m$. As $\bar{f} = (\varphi, M, \overrightarrow{v})$ has a finite monoid, there exists an integer $n \geq 0$ and an integer $p \geq 1$ such that $M^{n+p} = M^n$. For any integer $k \geq 0$ we denote by $(\varphi_k, M^k, \overrightarrow{v}_k)$ a presentation of f^k. Let us consider the Presburger linear function g defined by the presentation $\bar{g} = (f^n(\varphi_{n+p}), Id_m, (M^p - Id_m)\overrightarrow{\mathsf{v_n}} + \overrightarrow{\mathsf{v_p}})$. We observe that $f^{p+n} = g \circ f^n$ and the following equality provides the reduction:

$$f^* = \bigcup_{r=0}^{n-1} f^r \bigcup_{r=0}^{p-1} g^* \circ f^{n+r}$$

□

Remark 3. The finiteness of the monoid of a Presburger linear presentation is decidable in polynomial time [19].

We have proved that for a Presburger linear presentation $\bar{f} = (\varphi, M, \overrightarrow{v})$ with a finite monoid, the transition relation f^* can be expressed as a Presburger formula. Then computing a ndd representing f^* can be achieved. An upper bound for the construction of this ndd is 3-EXPTIME in the size of the ndd encoding φ (denoted $|\mathcal{A}(\varphi)|$), the values $\left|\left|\overrightarrow{v}\right|\right|_\infty$ and $||M||_\infty$, and the number of counters m. The algorithm is implemented in FAST and the upper bound has never been reached on case-studies, except for the TTP system (with two faults), for which we have designed a special acceleration algorithm that takes into account the particular form of the functions f manipulated.

For some subclasses of Presburger linear functions f with a finite monoid, a more efficient algorithm for computing f^* can be expected.

Definition 3 (Convex translation [4]). A convex translation f is a Presburger linear function $f = (\varphi, 1_m, \overrightarrow{v})$ where 1_m is the identity matrix and φ is a convex polyhedron.

Convex translations are a subclass of Presburger linear functions with finite monoid. The class encompasses for example Petri nets and Minsky machines. Actually, we can use geometrical properties of convex sets to alleviate the transitive closure construction. In fact, in this case the Presburger formula (1) can be replaced by the following one:

$$\exists k \geq 0,\ \overrightarrow{x'} = \overrightarrow{x} + k\overrightarrow{v} \wedge \varphi(\overrightarrow{x}) \wedge (k = 0 \vee \varphi(\overrightarrow{x'} - \overrightarrow{v})) \quad (2)$$

parameter	(*magnitude*)	standard algo.	convex algo.
$\lvert\mathcal{A}(\varphi)\rvert$	(10^5)	3-EXP	quadratic
m	$(< 10^2)$	3-EXP	EXP
$\lvert\lvert v\rvert\rvert_\infty$	(≤ 10)	3-EXP	poly. in m
$\lvert\lvert M\rvert\rvert_\infty$	(≤ 10)	3-EXP	$= 1$

Table 1. Complexity of the acceleration algorithms (upper bounds)

f taken from protocol	$\lvert\mathcal{A}(f^*)\rvert$	Time (sec.) Standard/Convex	Memory (MB) Standard/Convex
Dekker, 22 var	1,536	0.7/0.8	4.6/4
Mesh32, 52 var	1,614	2.1/2.5	8/7.8
Mesh32, 52 var	16,766	10.3/7.4	31/13
TTP2, 19 var	26,409	5.6/2.3	17/18
Dekker, 22 var	41,950	18/10.2	52/30
TTP2, 19 var	190,986	50/9	400/140
TTP2, 19 var	380,332	↑↑↑/34	↑↑↑/534

Table 2. Practical comparison of acceleration algorithms

The relation f^* is proved in [4] to be computable in time bounded by: $|\mathcal{A}(f^*)| \leq |\mathcal{A}(\varphi)|^2.4.(4.m.\,||\overrightarrow{v}||_\infty + 1)^{3.m}$. The main reason for this improvement w.r.t. theorem 1 is that formula (2) has less quantifiers than formula (1), while each quantifier may introduce an exponential blow-up in both time and space. We call convex acceleration the algorithm described in [4] to compute the transitive closure of convex translations.

Remark 4. Since the ndd representation is canonical, the resulting ndd is the same as the one obtained with standard acceleration. The difference is in the intermediate ndd reached during the computation.

The complexity of the convex acceleration is quadratic in $|\mathcal{A}(\varphi)|$, polynomial in $||\overrightarrow{v}||_\infty$ and exponential in the number of counters m. This is a major improvement compared to the standard acceleration algorithm, especially when considering this parameter can take values greater than 10^5. Table 1 recalls the upper bounds for each acceleration algorithm. These results are proved in [4]. Table 1 also provides the typical orders of magnitude of each parameter, based on our experiments on a set of some 40 counter systems taken from the literature. See section 7 for more details about the experiments.

In practice, convex acceleration allows to compute some f^* that cannot be computed by the standard acceleration algorithm (see [4] and section 9). Table 2 shows a comparison of both algorithms on different transitions. The convex algorithm performs better (in both time and space) than the standard algorithm as soon as the resulting automaton (of the computation) has approximately $10,000$ nodes. When $|\mathcal{A}(f^*)| \geq 100,000$ nodes, the convex algorithm is clearly more efficient, and it can be the case that it succeeds in computing $\mathcal{A}(f^*)$ while the standard algorithm fails.

4.2 Linear counter systems with a finite monoid

Definition 4 (Linear counter system [37]). A m-dim counter system $S = (Q, T, m)$ is a m-dim linear counter system if each transition $t \in T$ is labeled by a Presburger linear presentation $\bar{f}_t = (\varphi_t, M_t, \overrightarrow{v_t})$ such that $[\![l(t)]\!] = \{(\overrightarrow{x}, \overrightarrow{x}') \in \mathbb{Z}^{2m};\ \overrightarrow{x}' = f_t(\overrightarrow{x})\}$.

Notation. In the following, we do not distinguish anymore a Presburger linear function f and its presentation $\bar{f}$. There are many presentations for a single function, however $\bar{f}$ is unambiguously given by the linear counter system to be analyzed.

A key notion for linear counter systems is the *finiteness of the monoid of the system.* We now define the monoid of a linear counter system. We denote by $\mathcal{M}$ the set $\mathcal{M} = \{M_t | t \in T\}$.

Definition 5. The monoid $\mathcal{M}^*$ of a linear counter system S is the multiplicative monoid generated by the set of matrices $\mathcal{M}$, i.e. $\mathcal{M}^* = \bigcup_{n\geq 0} \bigcup_{M_1,\ldots,M_n \in \mathcal{M}} M_1 \ldots M_n$.

Remark 5. The finiteness of the monoid of a linear system is decidable in exponential time [59].

Many well-known subclasses of counter systems appear to be linear counter systems with finite monoid. Minsky machines, Petri nets extended with reset/ inhibitor/ transfer arcs [32,39], Ibarra's reversal-bounded counter machines [47] and broadcast protocols of Emerson et Namjoshi [33,34] are linear counter systems with finite monoids.

4.3 Flat linear counter systems with a finite monoid

Theorem 2 ([37]). *The reachability binary relation of a flat linear counter system with a finite monoid is effectively Presburger-definable.*

Linear counter systems with finite monoid satisfy two properties crucial for our approach. First, they encompass most of the interesting subclasses of counter systems. As a consequence, verification techniques for linear counter systems are very generic and can be applied to a large range of systems. Second, the reachability set of flat linear counter systems with finite monoid is effectively computable. Compared for example to Minsky machines, they have three specific advantages:

- Transitions are stable under composition, which simplifies the acceleration computation since a sequence of transitions σ behaves as a single transition.
- Transitions are more expressive than those of a Minsky machine. Even if any linear counter system is equivalent w.r.t. reachability to a Minsky machine, the corresponding control structure is much more difficult to handle because of the nested loops induced by the simulation.

The language of guards in transitions is closed by disjunction and this is a central requirement for the *reduction by union* described in section 5. This technique is intensively used in FAST and experiments prove it is a key feature of the tool.

5 Application to Flattable Counter Systems

An efficient acceleration algorithm is not sufficient to compute reachability sets. We need a way to select which circuits must be used to achieve the computation. In [7], we identify the cornerstone notions of flattable systems and flattenings of systems. We then deduce a procedure for reachability set computation, maximal in the sense that it is complete relatively to flattable systems. This procedure is generic and schematic. Generic, in the sense that it does not depend of the particular data types manipulated by the system; and schematic, since one must implement two abstract sub-procedures (Choose and Watchdog) to obtain an effective (and maximal) procedure. In this section we recall some results from [7] and present the reachability set computation procedure. We then discuss the implementations of the Choose and Watchdog procedures in FAST, and introduce specific optimizations for circuit selection. Finally, we discuss some questions about flattable systems.

5.1 Flattenings and flattable systems

Since most systems of interest are not flat, the issue is to deal with non-flat systems. A way to do it is to consider *flattenings* [7] of the system under study. A flattening S' of a system S (see figure 4) is a flat system simulated by S. Note that flattening is a generalization of unfolding, where one elementary cycle (and only one) is allowed on each location.

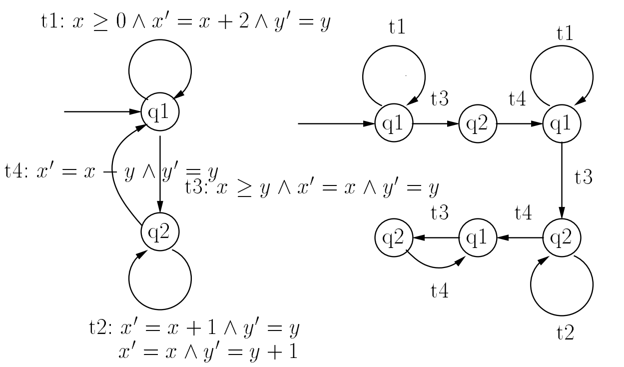


Fig. 4: A system (left) and one of its flattenings (right)

A flattening S' of a system S defines a subreachability set $\mathsf{post}^*_{S'}(X')$ included in $\mathsf{post}^*_S(X)$ (for some X' derived from X, see [7]). A system S is flattable [7] when at least one of its flattenings S' is equivalent to S w.r.t. reachability, i.e. $\mathsf{post}^*_{S'}(X') = \mathsf{post}^*_S(X)$. Since S' is flat, the set $\mathsf{post}^*_{S'}(X')$ is computable. Then using enumeration of flattenings and circuit acceleration, the set $\mathsf{post}^*(X_0)$ can be computed if (S, X_0) is flattable. Actually, the reverse implication also holds [7]. Flattable systems appear to be a maximal class for reachability set computation by circuit accelerations.

5.2 A complete procedure for flattable systems

The following procedure is complete (w.r.t flattable systems) for reachability set computation of (S, X_0): enumerate a flattening S' of S, compute $\mathsf{post}^*_{S'}(X'_0)$, test if it is a fixpoint of S: if so, return, otherwise iterate.

However, such a procedure will surely consume too many resources in practice. We proposed in [7] an approach which proves to be efficient in practice. A *restricted regular linear expression* (rlre) over an alphabet σ is a regular expression of the form $w_1^* \ldots w_n^*$ where $w_i \in \Sigma^*$. The fixpoint computation for flattable systems reduces to exploring the set of rlre over T. This can be achieved by building iteratively an increasing sequence of rlre such that each $w \in T^*$ is present infinitely often in the sequence.

The key issue is to select the $w \in T^*$ to be added to the sequence at each step, such that the fixpoint is reached quickly. Procedure 2 presents our complete heuristic. Instead of considering all sequences in T^*, we consider only sequences of length less than or equal to some bound k. This set of sequences is denoted $T^{\leq k}$, and a circuit selection where length of circuits is limited to k is said to be *k-flattable*. If the search fails, it is eventually stopped, k is incremented and the *k-flattable* macro is launched again. Procedure Watchdog decides when *k-flattable* should be aborted, and procedure Choose selects at each step a sequence $w \in T^{\leq k}$.

The procedure is schematic: Choose and Watchdog are abstract. Assuming their implementations respect the fairness conditions listed in procedure 2, the procedure obtained is *correct and complete* for flattable linear counter systems with finite monoid [7].

5.3 Implementation of procedure REACH2

We describe the implementations of Choose and Watchdog in FAST. We believe that these solutions are generic enough to be used with other data types than counters.

Procedure Choose. There is no monotonic relationship between the size of a region and the size of its concretization (w.r.t. $\subseteq$). Then regions reached during intermediate steps of computation may have a size much larger than the one of the final region representing the fixpoint.

procedure REACH2($\mathbf{x}_0$)
input: a ndd $\mathbf{x}_0$
1: $\mathbf{x} \leftarrow \mathbf{x}_0$; $k \leftarrow 0$
2: $k \leftarrow k+1$
3: **start**
4: **while** POST($\mathbf{x}$) $\not\subseteq \mathbf{x}$ **do** /* k-flattable */
5: Choose fairly $w \in T^{\leq k}$
6: $\mathbf{x} \leftarrow$ POST_STAR($w, \mathbf{x}$)
7: **end while** /* end k-flattable */
8: **with**
9: **when** Watchdog stops **goto** 2
10: **return** $\mathbf{x}$

Fairness: we assume that along an infinite execution path of REACH*2, procedure* Watchdog *is called infinitely often. Moreover, between two calls to* Watchdog*, each $w \in T^{\leq k}$ is selected at least once.*

Procedure 2: Procedure REACH2

Such large intermediate regions must be avoided as much as possible. Choose selects the next $w \in T^{\leq k}$, such that $|\text{POST\_STAR}(w, \mathbf{x})| \leq |\mathbf{x}|$. If there is no such w, then the next one is selected.

Procedure Watchdog*.* On the one hand, the procedure should detect as early as possible that the length of circuits is not sufficient to compute the reachability set, in order to avoid useless computations. On the other hand it should keep the length of circuits tight enough to prevent $|T^{\leq k}|$ from becoming intractable. Let us denote by *depth* the number of iterations of line 4 of Procedure 2 (macro k-flattable, *depth* is reset to 0 when exiting the macro). Our stop criterion for Watchdog is a maximal limit on depth. In practice, with a value of k large enough, the fixpoint is computed within a few iterations.

Completeness? These implementations of Choose and Watchdog do not fully respect the fairness conditions defined in procedure 2, thus termination is no longer guaranteed in theory. However, in practice, FAST terminates on many examples, as reported in section 7.

5.4 Reduction of the number of cycles

A remaining issue in procedure REACH2 is the cardinal of $T^{\leq k}$ exponential in k. We use *reduction techniques* [7, 37] to decrease dramatically the number of *useful* sequences, so that the enumeration becomes tractable in practice. The idea underlying reduction is that all sequences are not needed to compute the reachability set, and moreover in some cases some finite sets of sequences can safely be replaced by a single transition keeping the same reachability set.

We mainly use two reduction techniques: reduction by union [37] and reduction by commutation [7].

- Reduction by union consists in merging two transitions with same Presburger linear functions and different guards $f_1 = (\varphi_1, M, \vec{v})$ and $f_2 = (\varphi_2, M, \vec{v})$ into a unique affine function $f_1 + f_2 = (\varphi_1 \vee \varphi_2, M, \vec{v})$.
- Reduction by commutation consists in removing transitions $g \cdot f$ and $f \cdot g$, where $f, g \in T^{\leq k}$ for some k, whenever f and g satisfy $\xrightarrow{f \cdot g} = \xrightarrow{g \cdot f}$. This is sound w.r.t. reachability since $(\xrightarrow{f \cdot g})^*$ and $(\xrightarrow{g \cdot f})^*$ are then equal to $(\xrightarrow{f})^* \bullet (\xrightarrow{g})^*$.

system	$\lvert T\rvert$	k	$\lvert C^{\leq k}\rvert$	U	C	U+C
csm	13	1	14	14	14	14
		2	**183**	103	57	**35**
consistency	8	1	9	9	9	9
		2	68	45	44	30
		3	**484**	172	299	**98**
swimming pool	6	1	7	7	7	7
		2	43	21	24	16
		3	259	56	114	28
		4	**1555**	126	614	**47**
		5	9331	252	3591	86

$|C^{\leq k}|$: number of valid circuits of length $\leq k$
U: number of valid circuits after the reduction by union
C: number of valid circuits after the reduction by commutation
U+C: number of valid circuits after the reduction by union and commutation

Table 3. Effect of circuit reductions on case-studies

In [37], it is proven that reduction by union reduces the number of cycles of length $\leq k$ of linear counter systems with finite monoid to a *polynomial number in* k. Table 3 shows the effect of these reductions on a few examples. The bold value of k indicates the length of circuits used by FAST to compute the fixpoint.

Both reduction techniques appear to perform well in practice, and their combination leads to impressive cutoffs: $|C^{\leq k}|$ is divided by 5 in the first two examples, and by 30 in the last one. Reductions are definitely a key feature in FAST performances, allowing the tool to consider circuits of length 4 or 5 in some examples.

Beyond flat acceleration. The reduction by union allows to compute some particular kinds of nested loops. Actually, when considering linear counter systems with guards defined over the full binary automata logic, the union reduction allows to compute the reachability set of non-flattable counter systems. The question is still open for standard linear counter systems.

5.5 Flattable systems almost everywhere!

The question "Given a linear counter system with finite monoid S, is S flattable?" is undecidable, since the reachability problem reduces to this question [7]. However many interesting subclasses of counter systems have been shown to be flattable. It is the case for 2-dim VASS [56], k-reversal counter machines, lossy VASS, Cyclic VASS and other subclasses [57].

This is interesting for at least two reasons. From a practical point of view, procedure 2 provides a unified and efficient algorithm to decide reachability on all these subclasses of counter systems. This is an important step, since even though most of these subclasses were known to be decidable, their algorithms were totally different and very difficult to extend. From a theoretical point of view, it is interesting to note that some of the previous proofs of reachability used specific cases of circuit acceleration and flattening. These proofs are easier to write once these concepts are clearly identified.

6 FAST: Tool description

FAST [5,9] is a tool for checking safety properties of linear counter systems. The tool is designed according to the flat acceleration framework.

6.1 Computational framework

FAST is organized through a client-server architecture. *The server* is the computation engine as described in section 6.1. It contains a Presburger library, the acceleration algorithm and the search heuristics. *The client* is a front-end which allows the user to interact with the server through a graphical user interface (GUI, figure 5). The server can also be used as a standalone tool. The server is written in C++ (7,400 lines) while the client is written in Java. The MONA library [53,61] provides basis for automata manipulations.

6.1.1 Software architecture

FAST engine is structured according to the flat acceleration framework. The program is organized around four main classes: Presburger-affine functions, ndd, acceleration algorithms and a flattening heuristic.

ndd are encoded in basis 2, least-digit first. The class provides standard set operations like union, intersection, complementation and projection, as well as the synthesis of a ndd from a Presburger formula. This implementation is built on the MONA package. Note that for efficiency purposes, MONA restricts automata to 2^{24} nodes.

Standard acceleration and convex acceleration algorithms are implemented, which can be used for both forward and backward computation.

The flattening heuristic follows procedure 2. Reduction by commutation and reduction by union are both available.

6.1.2 Technical issues

Procedure REACH2, data structures and algorithms presented in sections 3 and 4 provide the backbone of FAST. However, several practical problems are not covered by these results. For example, locations can be encoded explicitly or by counters, circuits can be computed statically or on-the-fly. Here, we describe some implementation choices made in FAST. Currently there is no known best solution for each of the problems mentioned hereafter.

Variables in $\mathbb{N}$. All the results of sections 3 and 4 hold for variables ranging over $\mathbb{Z}$. However in FAST counters range over $\mathbb{N}$. First, the corresponding ndd are smaller thanks to a simpler encoding, which leads to better performance. Second, this is not a strict restriction since we did not find any example where negative counters were required and moreover a variable $x \in \mathbb{Z}$ can always be encoded by two positive variables $x^+, x^- \in \mathbb{N}$ such that $x = x^+ - x^-$ and $(x^+ = 0 \vee x^- = 0)$.

Location encoding. A stated in remark 1 (page 4), bounded variables (control, boolean, bounded integer variables) are encoded as counter variables. On the one hand it allows for a better sharing of the reachability set structure and avoids an explicit product of control structures for systems composed of many components. On the other hand, we do not take any advantage of the boundedness of these variables. A solution may be to extend ndd with a bdd-like structure for bounded variables, following the work done in [11].

Static computation of circuits. We compute statically circuits of length k. Practical case studies show that this approach is tractable thanks to reductions. However discovering circuits on-the-fly, or at least a dynamic slicing of potential circuits, would probably be useful.

6.2 Input/Output

FAST takes as input a description of the system to be analyzed and a strategy specifying what to compute. Outputs are textual messages stating if the system is safe or not. Finally, a graphical user interface is also available.

6.2.1 The input system

The linear counter system can be described directly in the FAST formalism. However since many of FAST's case studies were extended Petri nets, we developed a tool [10] to transform a Petri net in PNML format into a FAST model. The language PNML [16] describes various extensions of Petri nets and is being standardized (ISO/IEC 15909-2).

6.2.2 The strategy

The strategy is a script specifying the sequence of computations to perform in order to prove the correctness of the system. This script language manipulates sets of configurations (`region`), sets of transitions (`transition`) and booleans. All basic set-operations are available. The user can define finite sets of transitions $T' \subseteq T^*$ and primitives to compute $\mathsf{post}^*(T', X_0)$ and $\mathsf{pre}^*(T', X_0)$ are provided. A standard forward analysis is specified using only four instructions: declare the initial region X_0, compute the reachability set $\mathsf{post}^*(T', X_0)$, declare the region P describing the property to check and finally test whether $\mathsf{post}^*(T', X_0) \subseteq P$.

The language also allows the user to guide the tool more precisely. For example a system can be analyzed in an incremental way, dividing the whole system into smaller parts (cf. section 9); the user can indicate circuits to be used; choose the acceleration algorithm; or set up parameters of the heuristics.

The script language gives the user control over the sequence of computations performed. This can prove useful when the fully automatic approach fails. Thus, FAST stands between a fully automatic approach, justified when termination is guaranteed but restrictive otherwise, and computer-aided verification.

6.2.3 User Interface

A graphical user interface [6] is available (see figure 5). It provides aided editing of systems and strategies, pretty printing, and predefined strategies. Once the computation starts, the interface supplies the user with feedback on a number of parameters (memory consumption, time elapsed, etc.).

6.3 FAST Extended Release

An extended version of FAST has been presented in [9]. This new release offers mainly an open architecture allowing to plug easily any Presburger package to the tool.

Open architecture. The architecture has been slightly redesigned and is now divided into two parts: on the one side, a counter system analysis engine built upon a generic Presburger API (instead of a ndd package); on the other side, various implementations of this API. The generic Presburger programming interface (GENEPI) requires only basic set-theoretic operations on Presburger-definable sets. We provide three implementations of the API based on standard packages LASH [55], MONA [61] and OMEGA [62]. The first two packages are automata-based while OMEGA is formula-based. The MONA implementation corresponds to the original version of FAST. All experiments carried out in this paper use the MONA implementation.

The shared automata package. An implementation of the API using shared automata introduced by Couvreur in [28] has been developped by Jérôme Leroux and Gérald Point. These automata share their strongly connected

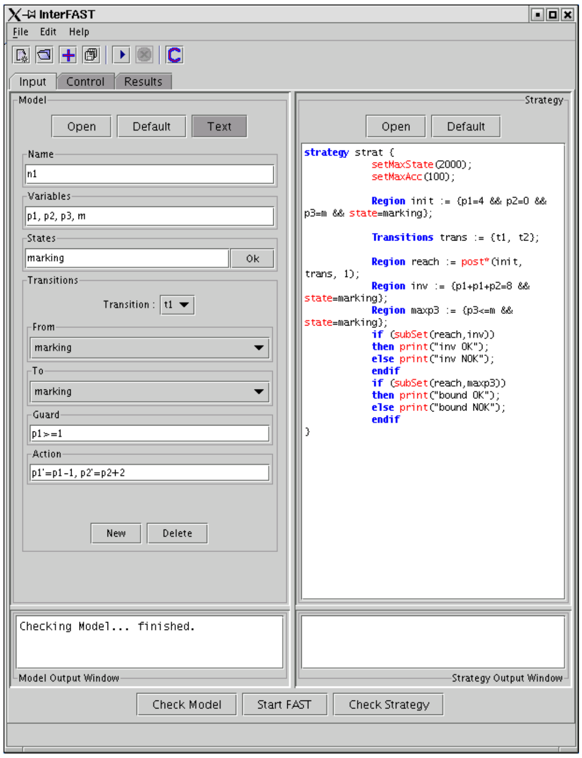


Fig. 5: FAST graphical user interface

components in a bdd-like manner, allowing to implement important features for intensive computation, such as cache computation and constant-time equality testing. The library is functional, but the computation cache is not sufficiently optimized yet. The shared automata package is called PRESTAF.

Experimental comparisons performed in [9] demonstrate that the three automata-based implementations of the generic Presburger API largely outperform the formula-based implementation. Indeed OMEGA appears to compute unduly complicated Presburger formulas (even with the simplification method provided by the package), while LASH, MONA and PRESTAF benefit from canonical representations of automata.

7 Experiments

This section reports some experiments made with FAST.

7.1 About tests

We use a large pool of counter systems and case studies analyzed by tools ALV, BABYLON[3], BRAIN, LASH and TREX to evaluate FAST. These 37 systems are available on FAST web pages [35].

[3] http://www.ulb.ac.be/di/ssd/lvanbegin/CST

Symbol	Meaning
m	number of counters
T	set of transitions
k	length of circuits used by FAST
$C^{\leq k}$	set of circuits of length $\leq k$
$\lvert\mathcal{A}\rvert$	number of nodes of the ndd
$\lvert\rho\rvert$	length of the rlre built by REACH2
↑↑↑	memory exceeded
$\geq c$	time elapsed (memory consumed) greater than c seconds (Mb)
?	unknown value
-	computation does not apply

Table 4. Symbols used in test reports

These systems range from tricky academic puzzles like the swimming pool protocol [42] to industrial case studies like the cache coherence protocol for the Futurebus+. We distinguish three categories of systems: counter systems with a finite reachability set, monotonic counter systems with an infinite reachability set and linear counter systems with an infinite reachability set.

All experiments have been performed on an Intel Pentium III 933Mhz equipped with 512 Mbytes of memory. Time is in seconds and memory in Mbytes. FAST is used with the following settings: standard acceleration, basic strategy (no human guidance), MONA-based implementation of the Presburger API.

7.2 Results

Table 5 reports FAST behavior on the examples, using forward computation. The number of cycles $|C^{\leq k}|$ is given after reductions (union and commutation).

FAST computes successfully the reachability set of 78% of the systems considered. This ratio is 74% when considering only unbounded systems. We show in section 8 that FAST performs better than similar tools.

These good results validate the design of FAST. First, all examples are expressed straightforwardly by means of counter systems. Second, the monoid is always finite. At least 78% of the systems are flattable and have a Presburger-definable reachability set. Finally in 19% of the tests, the length of circuits used is strictly greater than 1. This number increases to 22% when considering only unbounded counter systems. This proves that considering circuits and not only loops is a major feature.

FAST limitations are likely to be more practical (memory consumption, time elapsed) than theoretical. Crucial points are not only the number of variables, but also (and mainly!) the structure of the reachability set and the length k of the circuits used. Indeed, when k is too large, the static computation of circuits consumes too many resources.

8 Comparison with other tools

In this section, we compare FAST with other tools, namely ALV, LASH and TREX, to evaluate their performance on exact forward reachability set computation. Let us pinpoint that the goal here is not to find out which tool is the best for counter system validation. Actually, it would be unfair for TREX which is mostly designed for timed automata extended with integer variables, and for ALV which offers full CTL model checking, backward computation, over-approximation and different symbolic representations. These experiments are rather used to evaluate the contribution of each particular feature of FAST.

8.1 The tools

First, we present the tools ALV, LASH and TREX, and compare them with FAST through the flat acceleration framework.

ALV [25,68] is designed to check any CTL formula on full counter systems. ALV also offers different symbolic representations for integer vectors (automata or Presburger formula) and a wide range of options, like backward computation, over-approximation [12] for the automata-based representation and acceleration [50] for the formula-based representation. This acceleration algorithm is designed for the following class of operations: there is no guard and actions are mostly relations of the form $x'_i \# x_i + c$ where $\# \in \{\leq, =, \geq\}$ and x'_i is the value of variable x_i after the transition occurs. Typically ALV uses approximate forward fixpoint computation to prune the state space during the backward fixpoint computations.
In the rest of the paper, we use the following configuration: automata-based representation (then no acceleration), forward computation, no over-approximation. In this configuration, the main differences with FAST are that no acceleration algorithm is available, the heuristic is similar to REACH1 and bounded variables are encoded by bdd [11].

LASH [55] works on linear counter systems. Regions are encoded by automata and standard acceleration is implemented for functions with a finite monoid. Without user guidance, LASH is restricted to loop acceleration (i.e. the heuristic considers only words $w \in T$ instead of sequences in T^*) because no circuit search is supplied.

TREX [3] manipulates counter systems restricted to timed automata-like operations[4]: guards are conjunctions of constraints $x_i - x_j \leq c$ and actions are of the form $x'_i = x_j + c$ where x_i is a variable, c is a constant and x_j is a variable or the constant 0. Regions are encoded by pdbm, an extension of dbm with additional

[4] Actually TREX is designed to check systems with clocks and counters. We consider here the restriction to counter systems.

System	m	$\|T\|$	sec	Mb	$\|\rho\|$	k	$\|C^{\leq k}\|$
Bounded counter systems							
Producer/Consumer	5	3	0,41	2,37	3	1	3
RTP	9	12	2,24	2,76	8	1	12
Lamport ME	11	9	2,70	2,88	11	1	9
Reader/Writer	13	9	9,68	23,14	23	1	9
Peterson ME	14	12	4,97	3,78	12	1	12
Dekker ME	22	22	21,72	5,48	36	1	22
Monotonic unbounded counter systems							
Manufacturing	7	6	$\geq$ 1800	?	?	?	?
swimming pool	9	6	111	29,06	9	4	47
CSM	13	13	45,57	6,31	32	2	35
Kanban	16	16	10,43	6,54	2	1	16
Multipoll	17	20	22,96	5,13	13	1	20
FMS	22	20	157,48	8,02	23	2	46
extended ReaderWriter	24	22	$\geq$ 1800	?	?	?	?
pncsa	31	38	$\geq$ 1800	?	?	?	?
Mesh2x2	32	32	$\geq$ 1800	?	?	?	?
Mesh3x2	52	54	$\geq$ 1800	?	?	?	?
Unbounded counter systems							
Synapse Cache Coherence	3	3	0,30	2,23	2	1	3
Berkeley Cache Coherence	4	3	0,49	2,75	2	1	3
M.E.S.I. Cache Coherence	4	4	0,42	2,44	3	1	4
M.O.E.S.I. Cache Coherence	4	5	0,56	2,49	3	1	5
lift controller - N	4	5	4,56	2,90	4	3	20
Illinois Cache Coherence	4	6	0,97	2,64	4	1	6
Firefly Cache Coherence	4	8	0,86	2,59	3	1	8
Dragon Cache Coherence	5	8	1,42	2,72	5	1	8
Esparza-Finkel-Mayr	6	5	0,79	2,55	2	1	5
ticket 2i	6	6	0,88	2,54	5	1	6
ticket 3i	8	9	3,77	3,08	10	1	9
barber m4	8	12	1,92	2,68	8	1	12
bakery	8	20	$\geq$ 1800	?	?	?	?
Futurebus+ Cache Coherence	9	10	2,19	3,38	8	1	10
Consistency	12	8	200	7,35	9	3	98
Central Server	13	8	20,82	6,83	11	2	25
Last-in First-served	17	10	1,89	2,74	12	1	10
Producer/Consumer Java - 2	18	14	13,27	3,81	53	1	14
Producer/Consumer Java - N	18	14	401,5	12,46	86	2	75
Inc/Dec	32	28	$\geq$ 1800	?	?	?	?
2-Producer/2-Consumer Java	44	38	$\geq$ 1800	?	?	?	?

Table 5. Fast in practice

parameters constrained by an arithmetic formula. An acceleration procedure is implemented, which allows at least all accelerations of Fast and Lash. However this procedure produces unrestricted arithmetic formulas and then inclusion becomes undecidable. The heuristic is restricted to $C^{\leq k}$, for a value of k statically defined by the user. Finally, TReX does not compute circuits statically but discovers them on-the-fly. A more in-depth comparison of Fast and TReX is presented in [29].

Table 6 compares the different tools through the flat acceleration framework. Column "termination" indicates the class of systems for which the tool terminates (F: flattable, k-F: k-flattable, Unif-b: uniformly bounded).

8.2 Comparison on forward computation

We now compare the capabilities of Alv, Lash, Fast and TReX in exact forward computation of reachability sets. The counter systems chosen for tests all have an infinite reachability set, except systems RTP, Lamport and Dekker. Results are summarized in table 7.

	system	symb. rep.	acceleration	termination
Alv	full	ndd	no	Unif-b
Fast	linear	ndd	flat	F
Lash	linear	ndd	loop	1-F
TReX	restricted	pdbm	interpolation	k-F (*)

(*) Termination modulo an oracle to decide inclusion.

Table 6. Different tools for the verification of counter systems.

Experimental results show a drop in performance of Alv and Lash when k increases. Fast completely supports the flat acceleration framework and obtains the best results. On the other side, Alv does not supply any acceleration mechanism and the tool does not succeed in computing these complex reachability sets. Between Alv and Fast, the tool Lash is restricted to loop acceleration and it terminates only on simple examples ($k \leq 1$). Note that when Lash is provided with the circuits to use, its performance is similar to that of Fast. The difference between Fast and Lash is primarily the length of circuits, not the ndd implementation. Finally, TReX performance is less correlated with k, since the tool ter-

System	ALV(*)	LASH	FAST	k	TREX
RTP (bounded)	T	T	T	1	T
Lamport (bounded)	T	T	T	1	T
Dekker (bounded)	T	T	T	1	T
ticket 2	T	T	T	1	T
kanban	↑	T	T	1	T
multipoll	↑	T	T	1	↑
prod/cons java (2)	↑	T	T	1	-
prod/cons java (N)	↑	↑	T	2	-
lift control, N	↑	↑	T	2	T
train	↑	↑	T	2	T
csm, N	↑	↑	T	2	↑
consistency	↑	↑	T	3	-
swimming pool	↑	↑	T	4	↑
pncsa	↑	↑	↑	?	↑
incdec	↑	↑	↑	?	↑
bigjava	↑	↑	↑	?	↑

T: computation of the reachability set in less than 20 minutes
↑: no termination in less than 20 minutes
- : the systems cannot be modeled in TREX

(*) These results are consistent with those reported by Bultan and Bartzis in [12].

Table 7. Comparison of different tools

minates for the lift system ($k = 2$) and fails on multipoll ($k = 1$).

These results demonstrate a strong correlation between the flat acceleration framework and practical termination. Comparison between ALV and LASH shows the benefits of acceleration, while comparison between LASH and FAST highlights the necessity of selecting circuits and not only loops.

TREX results show that pdbm is not a good symbolic framework for counter systems, since many systems cannot be modeled this way and moreover, despite acceleration, termination occurs less frequently. Again, recall that TREX is primarily designed to handle parametric timed systems.

8.3 Comments

FAST appears to be a very efficient tool for the forward computation of reachability sets of counter systems. In experiments, FAST performance is clearly superior to that of similar tools ALV, LASH and TREX.

Again, recall that it does not necessarily imply that FAST is better then the other tools for counter systems validation since we restricted the experiments to exact forward computation while other approaches exist. Moreover, recall that we use restrictions of ALV and TREX which are primarily designed to handle different systems (TREX) or richer properties (ALV). Yet, we believe that the computation of the exact reachability set of a linear counter system is an important issue and in this setting, technologies implemented in FAST are clearly superior.

9 The TTP protocol

This section describes the verification of the TTP protocol with the tool FAST. In prior work the protocol was verified correct by hand (for an arbitrary number of faults) or in a computer-aided manner (for one fault) with LASH and ALV. These tools could not verify correctness for two faults. FAST checks automatically the correctness of the protocol for one fault, and correctness is proved for two faults, using abstractions.

9.1 Protocol description

The TTP protocol [54] is supported by the transport industry (Airbus, Audi, EADS, PSA and others) and aims at managing embedded microprocessors. We focus here on the *group membership algorithm* of the TTP. It is a fault-tolerant algorithm, preventing the partitioning of valid microprocessors (stations) after a failure.

A clique is a subset of stations communicating only with stations of the same clique. In normal behavior, there is only one clique containing all the valid stations. The protocol ensures that when a fault occurs and creates different cliques among the stations, after a while valid stations belong to a unique clique.

Description. Time[5] is divided into rounds. Each round is divided into as many slots as stations. The protocol behaves as follows (a more complete description can be found in [54, 22]):

1. Each station s_i keeps the following information: a list l_i of boolean values stating, for each station s_j, whether s_i considers s_j as valid or not; two counters C^i_{Ack} and C^i_{Fail}.
2. During a slot, only one station broadcasts a message and the others receive it. The message is the list l_i.
3. When a station s_j receives a message from a station s_i: if $l_i \neq l_j$, or if no message is received, then s_j considers s_i as faulty; l_j is updated and C^j_{Fail} is incremented. Otherwise C^j_{Ack} is incremented.
4. When a station s_i is about to broadcast a message: if $C^i_{Ack} \leq C^i_{Fail}$ then s_i considers itself as invalid and becomes inactive (no emission). Otherwise C^i_{Ack} and C^i_{Fail} are reset to 0, and l_i is broadcasted to all other stations.

9.2 Modeling

We use the modeling proposed by Merceron and Bouajjani in [22]. This modeling is based on counter systems. It captures an arbitrary number N of stations but only

[5] Clocks are synchronized by other mechanisms of the TTP protocol.

a fixed number of faults. Merceron and Bouajjani actually provide an infinite family of counter systems, each modeling the behavior of the protocol for some number f of faults.

The counter system for $f = 1$ is given in figure 6. Variable C_w (resp. C_F) denotes the number of active stations (resp. inactive stations). Variable C_p counts the number of slots elapsed during the round. Since a round is divided into N slots, when $C_p = N$, variable C_p is reset to 0 and a new round begins. Location `normal` models the normal behavior of the protocol. When a fault occurs, the protocol enters abnormal behavior. Location `Round1` is the first round following the error. Location `later` represents the other rounds. A fault divides arbitrarily active stations into two cliques C_1 and C_0. We denote by C_1 and C_0 the number of stations of cliques C_1 and C_0. Variable d (resp. d_0, d_1, d_F) counts the number of active stations (resp. from C_0, from C_1, inactive) which have emitted during the round.

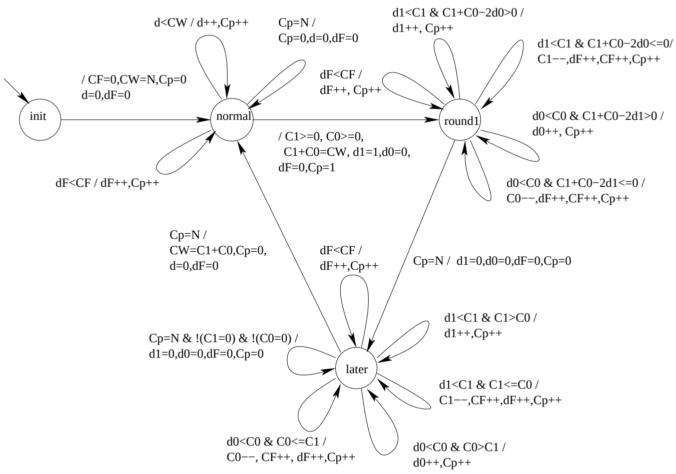


Fig. 6: Model for the TTP, 1 fault

The safety property to check is that, at most two rounds after the fault occurs, there is only one clique left. It is expressed in this model by the following property (P_1) : location = later $\wedge\ C_p = N \Rightarrow (C_1 = 0 \vee C_0 = 0)$.

Remark 6. This specification is actually incomplete, and we should check: *For all paths, if a fault occurs then location* `later` *is reached and property P_1 holds.* However this property is not a safety property. Model checking tools for infinite state systems such as ALV can handle these type of properties which cannot be handled by reachability tools such as FAST.

Interests of this case study. The number of counters is not large (9), and actions are standard. However guards are complex linear inequalities involving many variables. The counter system is not an extended VASS nor a restricted counter system manipulated by TReX. Moreover, because of the strong connection between variables, the reachability set has a very complex structure. However, it is Presburger-definable.

9.3 Automatic verification for 1 fault

The counter system of figure 6 is not linear because of the non-deterministic assignment of the transition between location `normal` and location `Round1`. Hopefully, since the transition between `later` and `normal` models returning back to the normal mode, property P_1 is only concerned with what happens in `later`, and variables affected non deterministically are not used in `normal`, we can remove this transition. Then the non deterministic assignment can be encoded into the initial region.

The resulting linear counter system is presented in figure 7. This system has a finite monoid.

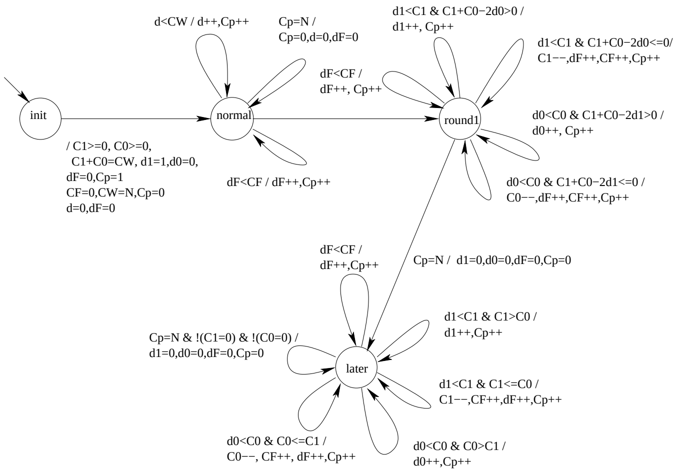


Fig. 7: Linear counter system for the TTP, 1 fault

Results. FAST checks *automatically* that property P_1 is satisfied. The computation of the reachability set requires only cycles of length 1, and the minimal `ndd` computed has 27, 932 nodes. Computation takes 1, 880 seconds and 73 Mb of memory.

Incremental analysis. The computation time can be reduced via a better script strategy. Indeed, FAST heuristic does not take into account the particular aspect of the control graph of the protocol. Since there is no returning back in the graph, we can first compute the set of all configurations reachable on location `normal`, then fire the transition to reach location `Round1`, and iterate the process. This decomposition is made explicit in figure 8.

With this method, computation time drops to 203 seconds for a memory consumption of 55 Mb [6].

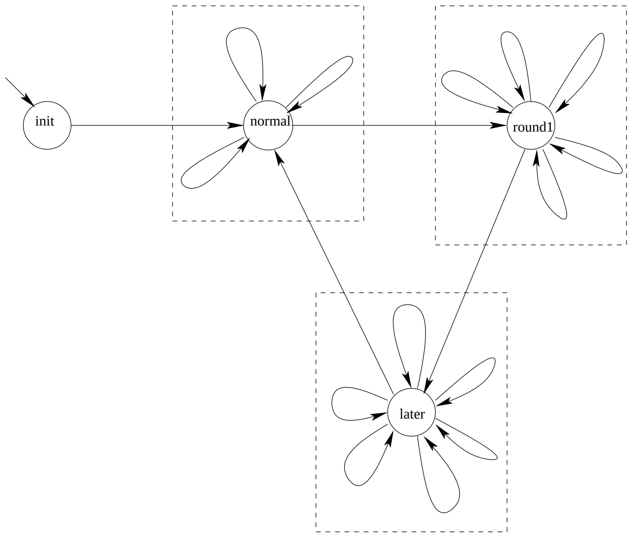


Fig. 8: Modular decomposition of the TTP

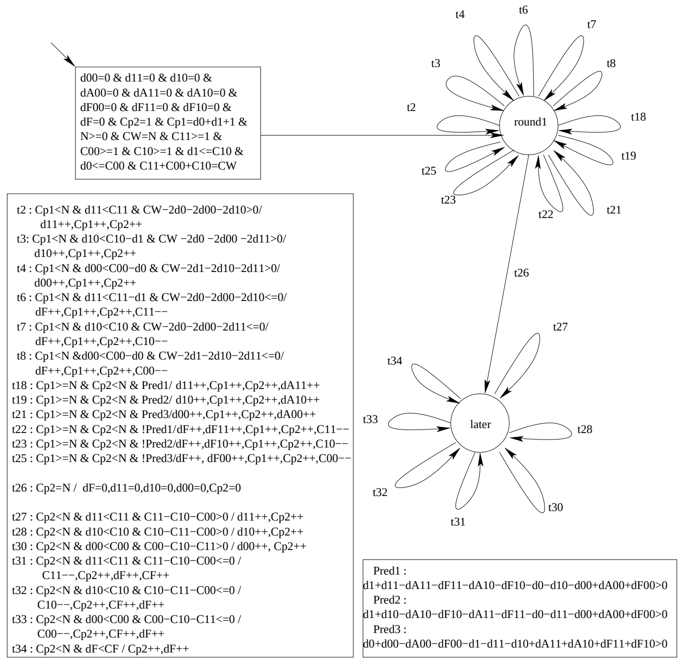


Fig. 9: Counter system for the TTP, 2 faults

9.4 Verification for 2 faults

The linear counter system for two faults is presented in figure 9. The normal behavior is not described in the figure. The system is much larger than the one for one fault, with 18 variables and guards involving up to 14 variables. There are now three different cliques. The absence of clique is expressed here by:

(P_2) : location = later $\wedge\, C_{p2} = N \Rightarrow (C_{11} \neq 0 \wedge C_{10} = C_{00} = 0) \vee (C_{10} \neq 0 \wedge C_{11} = C_{00} = 0) \wedge (C_{00} \neq 0 \wedge C_{10} = C_{11} = 0)$

Need for convex acceleration. When computing the transitive closure of transitions, the size of the automata computed becomes too large and exceeds FAST limitation of 2^{24} nodes, causing the program to crash. Luckily all transitions are convex translations, except t_{26} which does not need to be accelerated since it does not belong to any circuit. Hence the convex acceleration can be used, and transitive closures of transitions have all been computed.

Fixed number of stations. For a small fixed number of stations, the reachability set is computed. For $N = 5$, the reachability set is computed and P_2 is checked to be true. Computation requires 900 seconds and 588 Mb(!) of memory. The final ndd has 5,684 nodes. Computation succeeds with $N = 10$, but fails for $N = 15$ (the automata are too large).

Arbitrary number of stations. Since automata encountered during the computation are too large, we compute an over-approximation of the reachability set by relaxing some constraints and removing some variables. We hope this approximation has a simpler structure and is still precise enough to conclude. We use the following tricks:

Reduction of the number of variables, by using straightforward invariants like $C_W = C_{11} + C_{10} + C_{00}$.
Over-Approximation of the behavior, by removing some complex terms in the guards. Moreover some variables are removed in this process.
Modular computation, as described for one fault, to speed up computation.

The abstraction of the system is presented in figure 10. FAST checks that P_2 holds on this system, which proves the correctness of the TTP for 2 faults.

9.5 Results

Results are summarized in table 8. Convex acceleration always performs better than standard acceleration, in both time and space.

9.6 Verification with ALV*,* LASH *and* TREX

Here we report the tests we carried out to verify the TTP with the tools ALV, LASH and TREX.

With ALV[7], the reachability set computation does not terminate for one fault and an arbitrary number of sta-

[6] The resulting minimal ndd is the same than the one previously computed, but intermediate computations are less expensive.

[7] The settings are those considered in section 8.

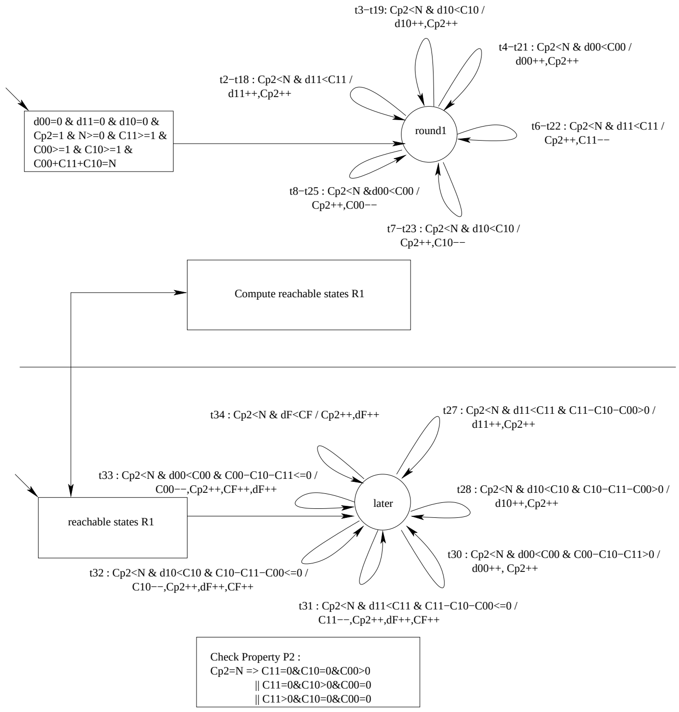


Fig. 10: Abstraction for the TTP, 2 faults

	standard acc.		convex acc.		
faults - stations	time sec.	mem. Mb	time sec.	mem. Mb	n. of nodes
1 - N	1,880	73	1,200	63	27,932
2 - 5	↑↑↑ (*)	↑↑↑ (*)	892	588	5,684
2 - 10	↑↑↑ (*)	↑↑↑ (*)	24,365	588	273,427
2 - 15	↑↑↑ (*)	↑↑↑ (*)	↑↑↑	↑↑↑	↑↑↑
2 - N	↑↑↑ (*)	↑↑↑ (*)	↑↑↑	↑↑↑	↑↑↑
2 - N (abstraction)	420	200	350	200	11,036

(*) The memory is saturated by acceleration computation; the verification process does not go further.

Table 8. Benchmark for the TTP (FAST)

tions. The verification reported in [22] is not fully automatic: ALV is used to check an intermediate invariant guessed by the authors on location `round1`. This invariant is then used to compute an over-approximation of the set of reachable states in location `later`. This is sufficient here to ensure the correctness, but the exact reachability set is never computed and the approach needs to guess the appropriate invariant.

LASH successfully computes the reachability set for one fault and an arbitrary number of stations, since only loop acceleration is required. However for two faults, the acceleration algorithm saturates the memory and the verification fails.

Finally, the TTP cannot be modeled in the (restricted) counter systems manipulated by TREX.

10 The CES service

In this section, we describe the verification of the service expected from the *Capability Exchange Signaling Protocol* (CES). In [58] Billington and Liu prove by hand the structure of the reachability graph. FAST automatically checks the results from [58] describing the nodes in the reachability graph.

10.1 Service description

The protocol aims at one peer informing the other of its multimedia capabilities. The length of communication channels (buffers) intervenes here as a parameter.

Figure 11 presents a colored Petri net modeling the CES service. This net is derived from [58]. Colored Petri nets (CPN) [48] are an extension of Petri nets where tokens are arbitrary typed data values (coloring). Colored tokens consumed and produced by transitions are defined by terms on the arcs. Functions and types are expressed in the ML language.

Places *OutControl* and *InControl* always contain only one token having two possible values. Place *forTransfer* contains a queue with a unique kind of message. Place *revTransfer* contains a queue with two kinds of messages: *transRes* and *rejReq*. Finally, place *dSymbol* uses three tokens, each containing a list of symbols. The transition *forLOST* models the loss of messages.

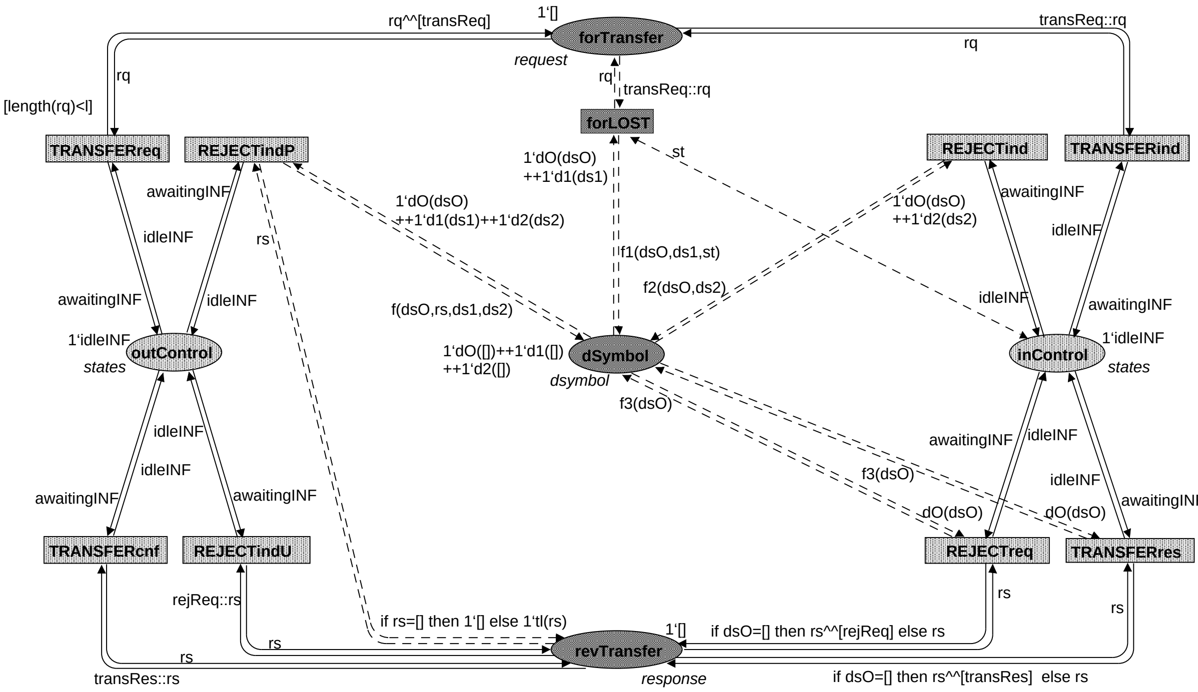
Remark 7. The length of queue *forTransfer* is bounded by l in this system, where l is a parameter of the CPN. When analyzing the CPN, the value of this parameter must be fixed. We want to remove this limitation in our counter system to enable a parametric verification of the protocol.

Billington and Liu [58] study the structure of the reachability graph of the CES service. In particular, they prove that the reachability set contains exactly 12 configurations for a queue of length 1, and for each increment of the queue length, 4 new configurations are added to the reachability set. These additional configurations are completely characterized [58, table 2, page 287].

Interest. The CES is naturally a queue system. Since FAST manipulates counters and not queues, it was not the best suited tool at first sight. We show how to model on this specific case the queues by counters, and how to check with FAST that the translation is sound. Such an approach is further detailed in [18, 17].

10.2 A counter system for the CES

The first step is to transform the CPN into a counter system. Queues are modeled explicitly in the CPN, but




```
color states = with idleINF | awaitingINF;
color req = with transReq;
color request = list req;
color res = with transRes | rejReq;
color response = list res;
color dsym = with d;
color dsymb = list dsym;
color dsymbol = union dO: dsymb + d1: dsymb + d2: dsymb;

var rq: request;
var rs: response;
var dsO,ds1,ds2: dsymb;
var st: states;

fun f1(dsO,ds1,st) = if dsO=[] orelse (st=awaitingINF andalso length(dsO)=1)
 then 1'dO(dsO)++1'd1(ds1^^[d])
    else 1'dO(tl(dsO))++1'd1(ds1);
fun f2(dsO,ds2) = if dsO=[]
    then 1'dO(dsO)++1'd2(ds2^^[d])
    else 1'dO(tl(dsO))++1'd2(ds2);
fun f3(dsO) = if dsO=[]
    then 1'dO(dsO)
    else 1'dO(tl(dsO));
fun f(dsO,[],[],[])=1'dO(dsO^^[d])++1'd1([])++1'd2([])
  | f(dsO,[],[],d::ds2)=1'dO(dsO)++1'd1([])++1'd2(ds2)
  | f(dsO,[],d::ds1,[])=1'dO(dsO)++1'd1(ds1)++1'd2([])
  | f(dsO,r::rs,[],[])=1'dO(dsO)++1'd1([])++1'd2([])
  | f(_,_,_,_) = empty;
```

Fig. 11: CPN modeling the CES service specification, derived from [58]

must be represented by a set of integer variables in the counter system. Since the queue *forTransfer* has only one kind of message, it is straightforwardly replaced by a counter stating the number of messages in the queue. The second queue *revTransfer* is more complex to deal with, since it has two kinds of messages *transRes* and *rejReq*. Following [58], we make the assumption that the two kinds of messages never coexist in the queue. Then the queue is modeled by two counters, one for each kind of message. The validity of this assumption is checked in section 10.3.

The counter system derived from the CPN model is presented in figure 12 (only typical parts are included). Variable *buflg* represents the maximal size of the queue *forTransfer* (it corresponds to parameter l in the CPN). The system has a single location `marking`. Place *dsymbol* in the CPN always contains three tokens, which are queues containing a single type of element. Therefore, in the counter system, it is represented by three different variables $d0, d1, d2$. Since some transitions have complex firing modes, due to the functions on arcs and in guards, they need to be split into different transitions of the counter system (e.g. *REJECTreq*).

10.3 Model correctness

We check the modeling hypothesis about queues. To ensure that two kinds of messages never coexist in the queue *forTransfer*, we show that all reachable configurations satisfy: either *revTrans* has a null value, or *rejTrans* has a null value. This is expressed by the following region:

```
Region bad := {(!(revTrans)=0) && (!(rejTrans)=0)};
```

```
model CES {
 var outControl, inControl, forTrans, revTrans, rejTrans,
     d0, d1, d2, buflg;
 states marking;

 transition TRANSFERreq := {
  from   := marking;
  to     := marking;
  guard  := outControl=0 && forTrans<buflg;
  action := outControl'=1, forTrans'=forTrans+1;
 };
...
 transition REJECTreq1 := {
  from   := marking;
  to     := marking;
  guard  := inControl=1 && d0=0;
  action := inControl'=0, rejTrans'=rejTrans+1;
 };

 transition REJECTreq2 := {
  from   := marking;
  to     := marking;
  guard  := inControl=1 && !d0=0;
  action := inControl'=0, d0'=d0-1;
 };
}
```

Fig. 12: Description in FAST of the CES service

Region *bad* is then intersected with the reachability set. The resulting region is empty, thus the modeling hypothesis is correct. Note that *no value is needed for buflg*. This means that the result is valid for *any value* of *buflg*.

10.4 Verification with FAST

We automatically prove with FAST the characterization of nodes in the reachability graph obtained by Liu and Billington in [58]. To do so, a region is declared for each of the 12 configurations of the system when the buffer length l is 1 (*mark1* to *mark12*). Additional configurations are defined according to the formula in [58, table 2, page 287] (*mark1n* to *mark4n*). The reachability set is supposed to be equal to the union of all these regions (region *fullOG*). The script in figure 13 performs the automatic verification of this result.

The reachability set is computed with cycles of length 2. The result is positive, i.e. the reachability set is as the expected. Since variable *buflg* is never set, the property is automatically proved for any queue length. FAST terminates in less than 30 seconds.

10.5 Results

FAST has been used with success to verify the characterization of configurations for a parametric lossy channel system. A major issue is the modeling of the queues by counters, which requires some assumptions on the contents of the queue. FAST proves the correctness of the assumptions, and also the expected property.

```
strategy forward {
Transitions t := {TRANSFERreq, TRANSFERcnf, REJECTindU,
                  TRANSFERind, ...};

Region mark1  := {outControl=0 && inControl=0 && forTrans=0 &&
                 revTrans=0 && rejTrans=0 && d0=0 && d1=0 &&
                 d2=0 && state=marking};
...
Region mark12  := {outControl=1 && inControl=1 && forTrans=0 &&
                  revTrans=0 && rejTrans=0 && d0=1 && d1=1 &&
                  d2=0 && state=marking};

Region marks1n  := {forTrans<=buflg && d0=forTrans-1 &&
                   outControl=1 && inControl=0 && revTrans=0 &&
                   rejTrans=0 && d1=0 && d2=0 && state=marking};
...
Region marks4n  := {forTrans<=buflg && d0=forTrans+1 &&
                   outControl=0 && inControl=1 && revTrans=0 &&
                   rejTrans=0 && d1=0 && d2=0 && state=marking};

Region fullOG := mark1 || ... || mark12 ||
                 marks1n || ... || marks4n;

Region reach := post*(mark1 && {buflg>0}, t, 2);

if (eqSet(fullOG && {buflg>0},reach)) then
   print("fullOG OK");
endif
}
```

Fig. 13: The reachability set

10.6 Verification with ALV *and* LASH

We report our verification of the CES with ALV and LASH. The tool ALV (same settings as in section 8) does not terminate in less than 20 minutes. The tool LASH also does not terminate with only loop acceleration. When specifying cycles of length 2 to be accelerated (these are deduced from the feedback of FAST computation), then LASH terminates quickly. We have not experimented with TReX on this example.

11 Conclusion

In this paper, we have presented FAST, a tool for checking safety properties on counter systems. FAST is an implementation of the flat acceleration framework instantiated to counter systems. The tool implements state-of-the-art technologies such as automata-based representation of Presburger-definable sets, acceleration of linear functions and automatic selection of interesting circuits through dedicated heuristics and reductions. It follows a clear design and each step is justified as rigorously as possible, considering the whole problem is undecidable.

We sketched in the paper all the theoretical foundations of FAST, and described the architecture of the tool. We then described lengthy experiments carried out, and we have compared FAST with other tools having similar goals. The main points of the tool are a very expressive input model allowing many systems to be expressed directly, a powerful engine able to compute the reachability set in most cases, the possibility to guide the tool for complex examples and a clear design.

Many experiments have been successfully carried out. Despite the fact that reachability sets of counter systems are not computable, FAST terminates in about 75% of our experiments. FAST has been the first tool to automatically verify the TTP, a complex fault-tolerant protocol. FAST has also been used to check a parametric property of a lossy channel system, the CES service. These performances are far better than those of similar tools. Actually, comparison with tools like ALV and LASH proves that each mechanism of FAST is of importance. Comparison with ALV demonstrates clearly that circuit acceleration enhances greatly the termination of the reachability set computation, while comparison with LASH shows that considering circuits of arbitrary length (not restricted to loops) is of major importance for many systems. Experiments made with FAST demonstrate that the flat acceleration framework is sound for the verification of counter systems.

Perspectives. FAST has proved to be efficient for counter systems with approximately 20 unbounded variables. The next step is to scale up the techniques to wider systems. We are currently looking towards three directions: (1) improve the ndd representation, for example using cache systems; (2) improve the circuit selection with new reductions and dynamic discovery; (3) relax the exact computation and mix widening and abstraction with acceleration. Another interesting issue is to investigate how to decide richer properties on counter systems, for example liveness. First results have been obtained for LTL [31].

Acknowledgements. We thank Jonathan Billington, Guy Gallasch and Philippe Schnoebelen for their comments on earlier versions of the paper. We are also grateful to Jonathan Billington and Lin Liu for having provided us with the CPN model of the CES service, to Jean-Michel Couvreur for giving us advice for the implementation of shared automata, and to Ales Smrcka for adapting the OMEGA source code to recent compilers.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
2. A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2000.
3. A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Proc. 13th Int. Conf. Computer Aided Verification (CAV'2001), Paris, France, July 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer, 2001.
4. S. Bardin, A. Finkel, and J. Leroux. FASTer acceleration of counter automata. In *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2004) Barcelona, Spain, Mar. 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590. Springer, 2004.
5. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.
6. S. Bardin, A. Finkel, J. Leroux, L. Petrucci, and L. Worobel. *FAST user manual*, 2003. 33 pages. Available at `www.lsv.ens-cachan.fr/fast/`.
7. S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *Proc. of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'2005) , Taipei, Taiwan, October. 2005*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.
8. S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In *Proc. 5th Int. Workshop Automated Verification of Infinite-State Systems (AVIS'2006)*, Vienna, Austria, 2006.
9. S. Bardin, J. Leroux, and G. Point. FAST Extended Release. In *Proc. 18th Int. Conf. Computer Aided Verification (CAV'2006), Seattle, Washington, USA, August 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66. Springer, 2006.
10. S. Bardin and L. Petrucci. From PNML to counter systems for accelerating Petri nets with FAST. In *Proc. of the Workshop on Interchange Formats for Petri Nets (at ICATPN 2004)*, pages 26–40, June 2004.
11. C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. of Foundations of Computer Science*, 14(4):605–624, 2003.
12. C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proc. 16th Int. Conf. Computer Aided Verification (CAV 2004), , Boston, Massachussetts , July 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2004.
13. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.
14. B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, Aug. 1999*, volume 1664 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 1999.
15. L. Berman. Precise bounds for Presburger arithmetic and the reals with addition: Preliminary report. In *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77), Providence, RI, USA, Oct.-Nov. 1977*, pages 95–99. IEEE, 1977.
16. J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and*

Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.

17. J. Billington, G. E. Gallasch, and L. Petrucci. FAST verification of the class of stop-and-wait protocols modelled by coloured Petri nets. *Nordic Journal of Computing*, pages 37–55, 2005. To appear.
18. J. Billington, G. E. Gallasch, and L. Petrucci. Transforming couloured Petri nets to counter systems for parametric verification: A stop-and-wait protocol case study. In *Proc. 2nd workshop on MOdel-based Methodologies for Pervasive and Embedded Software (MOMPES'05, satellite of ACSD'05), Rennes, France*, volume 39, pages 37–55. TUCS general publication, June 2005.
19. B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998.
20. B. Boigelot. On iterating linear transformations over recognizable sets of integers. *Theoretical Computer Science*, 309(2):413–468, 2003.
21. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94), Stanford, CA, USA, June 1994*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.
22. A. Bouajjani and A. Merceron. Parametric verification of a group membership algorithm. In *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'2002), Oldenburg, Germany, Sep. 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2002.
23. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proc. 21st Int. Coll. on Trees in Algebra and Programming (CAAP'96), Linköping, Sweden, Apr. 1996*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 1996.
24. V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc.*, 1(2):191–238, Mar. 1994.
25. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proc. 16th IEEE Int. Conf. Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, pages 382–386. IEEE Computer Society, 2001.
26. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
27. H. Comon and Y. Jurski. Multiple counters automata, safety analysis, and Presburger arithmetic. In *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998.
28. J.-M. Couvreur. A bdd-like implementation of an automata package. In *Proc. 9th Int. Conf. Implementation and Application of Automata (CIAA'2004), Kingston, Canada, July 2004*, volume 3317 of *Lecture Notes in Computer Science*, pages 310–311. Springer, 2004.
29. Ch. Darlot, A. Finkel, and L. Van Begin. About Fast and TReX accelerations. In *Proceedings of the 4th International Workshop on Automated Verification of Critical Systems (AVoCS'04)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, London, UK, Aug.-Sept. 2004. Elsevier Science Publishers.
30. G. Delzanno, J.-F. Raskin, and L. Van Begin. Covering sharing trees: a compact data structure for parameterized verification. *Journal of Software Tools for Technology Transfer*, 5(2–3):268–297, 2004.
31. S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen. Towards a model-checker for counter systems. In *Proc. 4th Int. Symp. Automated Technology for Verification and Analysis (ATVA'2006)*, volume 4218 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2006.
32. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.
33. E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. 13th IEEE Symp. Logic in Computer Science (LICS'98), Indianapolis, IN, USA, June 1998*, pages 70–80. IEEE Comp. Soc. Press, 1998.
34. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99), Trento, Italy, July 1999*, pages 352–359. IEEE Comp. Soc. Press, 1999.
35. FAST homepage. `http://www.lsv.ens-cachan.fr/fast/`.
36. A. Finkel. A generalization of the procedure of karp and miller to well structured transition systems. In *Proc. 14th Int. Coll. Automata, Languages, and Programming (ICALP'87), Karlsruhe, FRG, July 1987*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 1987.
37. A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002), Kanpur, India, Dec. 2002*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.
38. A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1–31, 2003.
39. A. Finkel and G. Sutre. Decidability of reachability problems for classes of two counters automata. In *Proc. 17th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2000), Lille, France, Feb. 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 346–357. Springer, 2000.
40. L. Fribourg. Petri nets, flat languages and linear arithmetic. Invited lecture. In M. Alpuente, editor, *Proc. 9th Int. Workshop. on Functional and Logic Programming (WFLP'2000), Benicassim, Spain, Sept. 2000*, pages 344–365, 2000. Proceedings published as Ref. 2000.2039, Universidad Politécnica de Valencia, Spain.
41. L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints*, 2(3/4):305–335, 1997.
42. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, Jul. 1997*, volume

1243 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 1997.

43. V. Ganesh, S. Berezin, and D. L. Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *Proc. 4th Int. Conf. Formal Methods in Computer Aided Design (FMCAD'02), Portland, OR, USA, nov. 2002*, volume 2517 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2002.
44. G. Geeraerts, J.-F. Raskin, , and L. Van Begin. Expand, enlarge and check... made efficient. In S. K. Rajjamani and K. Etessami, editors, *Proceedings of 17th International Conference on Computer Aided Verification - (CAV 2005)*, number 3576 in Lecture Notes in Computer Science, pages 394–404. Springer, 2005.
45. S. Ginsburg and E. H. Spanier. Semigroups, Presburger formulas and languages. *Pacific J. Math.*, 16(2):285–296, 1966.
46. O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25:116–133, 1978.
47. O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. A. Kemmerer. Counter machines and verification problems. *Theoretical Computer Science*, 289(1):165–189, 2002.
48. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: basic concepts.* Monographs in Theoretical Computer Science. Springer, 1992.
49. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.
50. W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. In *8th Int. Wor. Languages and Compilers for Parallel Computing (LCPC'95), Columbus, Ohio, USA, August 10-12, 1995*, volume 1033 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 1995.
51. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1–2):93–112, 2001.
52. F. Klaedtke. On the automata size for presburger arithmetic. In *Proc. 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04), Turku, Finland July 2004*, pages 110–119. IEEE Comp. Soc. Press, 2004.
53. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *Int. J. of Foundations Computer Science*, 13(4):571–586, 2002.
54. H. Kopetz and G. Grünsteidl. A time trigerred protocol for fault-tolerant real-time systems. In *IEEE computer*, pages 14–23, 1999.
55. Lash homepage. `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.
56. J. Leroux and G. Sutre. On flatness for 2-dimensional vector addition systems with states. In *Proc. 15th Int. Conf. Concurrency Theory (CONCUR'04), London, UK, Aug.-Sep. 2004*, volume 3170 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2004.
57. J. Leroux and G. Sutre. Flat counter automata almost everywhere! In *Proc. of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'2005) , Taipei, Taiwan, October. 2005*, volume 3707 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2005.
58. L. Liu and J. Billington. Tackling the infinite state space of a multimedia control protocol service specification. In *Proc. 23rd Int. Conf. Application and Theory of Petri Nets (ICATPN'2002), Adelaide, Australia, June 2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 273–293. Springer, 2002.
59. A. Mandel and I. Simon. On finite semigroups of matrices. *Theoretical Computer Science*, 5(2):101–111, Oct. 1977.
60. E. W. Mayr. Persitence of Vector Replacement Systems is decidable. *Acta Informatica*, 15:309–318, 1981.
61. Mona homepage. `http://www.brics.dk/mona/index.html`.
62. Omega homepage. `http://www.cs.umd.edu/projects/omega/`.
63. M. Presburger. Uber die volstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *C. R. 1er congrès des Mathématiciens des pays slaves, Varsovie*, pages 92–101, 1929.
64. T. Rybina and A. Voronkov. Brain: Backward reachability analysis with integers. In *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'2002), Saint-Gilles-les-Bains, Reunion Island, France, Sep. 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 489–494. Springer, 2002.
65. P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. 2nd Int. Symp. Static Analysis (SAS'95), Glasgow, UK, Sep. 1995*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 1995.
66. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 1998.
67. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), Berlin, Germany, Mar.-Apr. 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.
68. T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action language verifier, extended. In *Proc. 17th Int. Conf. Computer Aided Verification (CAV 2005), Edinburgh, Scotland, UK, July 6-10, 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 413–417. Springer, 2005.