Discrete Event Dynamic Systems manuscript No.
(will be inserted by the editor)

VeriSiMPL 2: An Open-Source Software for the Verification of Max-Plus-Linear Systems

Dieky Adzkiya · Yining Zhang · Alessandro Abate

Received: date / Accepted: date

Abstract This work presents a technique to generate finite abstractions of autonomous Max-Plus-Linear (MPL) systems, a class of discrete-event systems employed to characterize the dynamics of the timing related to the synchronization of successive events. Abstractions of MPL systems are derived as finite-state transition systems. A transition system is obtained first by partitioning the state space of the MPL system into finitely many regions and then by associating a unique state of the transition system to each partitioning region. Relations among the states of the transition system are then set up based on the underlying dynamical transitions between the corresponding partitioning regions of the MPL state space. In order to establish formal equivalences, the obtained finite abstractions are proven either to simulate or to bisimulate the original MPL system. The approach enables the study of general properties of the original MPL system formalised as logical specifications, by verifying them over the finite abstraction via model checking. The article presents a new, extended and improved implementation of a software tool (available online) for the discussed formal abstraction of MPL systems, and is tested on a numerical benchmark against a previous version.

Keywords Max-plus algebra · Discrete-event systems · Piece-wise affine systems · Transition systems · Model abstractions · Difference-bound matrices · Bisimulations · Model checking

This work has been supported by the European Commission via STREP project MoVeS 257005, Marie Curie grant MANTRAS 249295, and IAPP project AMBI 324432.

D. Adzkiya
Delft Center for Systems and Control, Delft University of Technology, Mekelweg 2, 2628 CD, Delft, The Netherlands
E-mail: d.adzkiya@tudelft.nl

Y. Zhang · A. Abate
Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom
E-mail: zhang.yining@hotmail.com, aabate@cs.ox.ac.uk

1 Introduction

The seminal work in [7, p. ix] characterizes discrete-event dynamic systems as man-made systems consisting of a finite number of resources (processors or memories, communication channels, machines) shared by several users (jobs, packets, manufactured objects), which contribute to the achievement of a common goal (a parallel computation, the end-to-end transmission of a set of packets, the assembly of a product in an automated manufacturing line). The dynamics of such systems have to deal with issues of synchronization and concurrency. Synchronization requires the availability of several resources or users at the same time, whereas concurrency appears when some user must choose among several resources at a particular time instant.

Max-Plus-Linear (MPL) systems are a class of discrete-event dynamic systems [7,15] with a continuous state space characterizing the time of occurrence of the underlying sequential discrete events. MPL systems are naturally predisposed to describe the evolution of timed event graphs in the events domain, under the assumption that timing events are linearly dependent (within the max-plus algebra) on previous event occurrences and possibly (for non-autonomous systems encompassing non-determinism) on external schedules. MPL systems have a wide range of applications: they have been employed in the analysis and scheduling of infrastructure networks, such as communication and railway systems [23], production and manufacturing lines [31,20], as well as of biological systems [12].

Traditionally the dynamical analysis of MPL systems is grounded upon their algebraic features [7]. It allows investigating global system properties such as its transient or asymptotic behaviors, its periodic regimes, or its ultimate dynamical behavior [16]. Those system properties can be studied by using the spectral theory of system matrices in the max-plus algebra. A number of results have appeared on the geometric theory of MPL systems as introduced in [14], such as the computation of different control on invariant sets [18,25] and the feedback controller design [28]. The application of model predictive control in MPL systems has been studied in [17] and in a subsequent line of work.

In this work, we consider the following verification problem: given an autonomous MPL system with a predefined set of initial states and given a specification expressing a property of interest, we determine whether the MPL system satisfies the specification. Specifications are expressed as either a Linear Temporal Logic (LTL) formula or a large fragment of Computation Tree Logic (CTL) formulae [8], and permit much finer description of properties than the classical results in the literature. Further, the approach allows zooming in on properties of (possibly single) traces of the system at hand, rather than on global analysis. In this work we develop an approach based on finite-state abstractions to tackle this problem. With regards to the abstraction procedure (cf. Sections 3 and 4), we put forward a technique that generates, in a finite number of steps, a finite-state transition system. The abstract states are obtained by finite partitioning the state space of the original MPL system: this partitioning is automatic and is based on the underlying dynamics and on the property of interest, and associates a unique abstract state to each partitioning region. The relations between pairs of abstract states are established by checking whether a trajectory of the original MPL system is allowed to transition between the corresponding partitioning regions. Computationally, this characterization is performed by forward-reachability analysis over a Piece-wise Affine (PWA) representation of the MPL dynamics [33], as discussed in [2]. The abstract transition system may contain behaviors that cannot be mimicked by the original MPL system [29]. In order to establish formal relationships between the concrete model and its abstraction, we argue that in general the abstract transition system simulates the original MPL system [30], and derive sufficient conditions on its bisimilarity. The overall approach leverages a representation of the spatial regions and of the dynamics based on Difference-Bound Matrices (DBM) [19]. This representation allows for compact and computationally fast operations on regions of the state space, and thus for fast computation of the quantities of interest. In the literature, finite abstractions have been applied to models such as hybrid [6,26, 30] and PWA systems [35,36], however with tolling computational costs. While we leverage a PWA representation of the given MPL dynamics [22] – a particular case of the PWA system used in [35,36] – to build the abstract transition system, techniques for abstractions of PWA systems developed in [36] do not appear to be applicable in the context of the models derived from MPL systems, since the abstraction depends on a specific LTL formula. Furthermore, the new approach in this work, hinging on DBM manipulations and on a partitioning that is tailored to the underlying dynamics, appears to be drastically more scalable.

Theoretical contributions This manuscript represents an extension of the results in [2]. The abstraction procedure in [2] assumes that the partition associated with atomic propositions is coarser than the partition associated with the affine dynamics: this work relaxes this assumption. A numerical study in [2] shows that the bottleneck of the abstraction procedure resides in the generation of transitions: in this paper, we attempt overcoming this bottleneck by using a tree structure for representing the partition of the state space (cf. Section 4): this significantly decreases the time to generate the transition relation (cf. Section 5.2). On the other hand, in the new implementation the drawback related to the use of the tree structure is the need for a higher amount of memory. Finally in Section 2.3.2, we discuss a new procedure to determine the complement of a DBM, which improves on [2].

Computational contributions The abstraction technique developed in this work has been implemented as a JAVA software tool, VeriSiMPL 2, which is freely available for download at http://sourceforge.net/projects/verisimpl/. According to the numerical benchmark in Section 5.1, a great improvement on the runtime is shown in the JAVA implementation compared with the original MATLAB implementation [1]. Furthermore VeriSiMPL [1] employs the SPIN model checker [24], whereas in VeriSiMPL 2 we use the NuSMV model checker [13]. SPIN is a full LTL model checking tool, whereas NuSMV is a software tool to verify both LTL and CTL formulae. Furthermore NuSMV embeds Binary Decision Diagram (BDD) [34] structures, geared towards time and space efficiency.

Outline The article is structured as follows. Section 2 introduces models and preliminary notions. The abstraction procedure for autonomous MPL systems is discussed in Section 3. Section 4 discusses the use of a tree as a key data structure for the abstraction procedure. Section 5 tests the developed approach over a computational benchmark, whereas a running case study is discussed throughout the manuscript. Furthermore for each procedure described in this paper, we elaborate on the data structure employed in the JAVA implementation. Finally, Section 6 concludes the work.

2 Models and Preliminaries

2.1 Max-Plus-Linear Systems

Define $\mathbb{N}$, $\mathbb{R}$, $\mathbb{R}_\varepsilon$, and ε respectively as the set of natural numbers $\{1,2,\ldots\}$, the set of real numbers, $\mathbb{R}\cup\{\varepsilon\}$, and $-\infty$. For $\alpha,\beta\in\mathbb{R}_\varepsilon$, introduce the two operations

$$\alpha\oplus\beta=\max\{\alpha,\beta\} \qquad \text{and} \qquad \alpha\otimes\beta=\alpha+\beta,$$

where the element ε is considered to be absorbing w.r.t. $\otimes$ [7, Def. 3.4], i.e. $\alpha\otimes\varepsilon=\varepsilon$ for all $\alpha\in\mathbb{R}_\varepsilon$. Given $\beta\in\mathbb{R}$, the max-algebraic power of $\alpha\in\mathbb{R}$ is denoted by $\alpha^{\otimes\beta}$ and corresponds to $\alpha\times\beta$ in the conventional algebra. The usual multiplication symbol $\times$ is usually omitted, whereas the max-algebraic multiplication symbol $\otimes$ is always written explicitly. The rules for the order of evaluation of the max-algebraic operators correspond to those of conventional algebra: max-algebraic power has the highest priority, and max-algebraic multiplication has a higher precedence than max-algebraic addition [7, Sec. 3.1]. The basic max-algebraic operations are extended to matrices as discussed next. Consider matrices $A,B\in\mathbb{R}_\varepsilon^{m\times n}$, $C\in\mathbb{R}_\varepsilon^{m\times p}$, $D\in\mathbb{R}_\varepsilon^{p\times n}$, and scalar $\alpha\in\mathbb{R}_\varepsilon$; we have that

$$\begin{aligned}
[\alpha\oplus A](i,j) &= \alpha\oplus A(i,j)=\max\{\alpha,A(i,j)\},\\
[A\oplus B](i,j) &= A(i,j)\oplus B(i,j)=\max\{A(i,j),B(i,j)\},\\
[C\otimes D](i,j) &= \bigoplus_{k=1}^{p} C(i,k)\otimes D(k,j)=\max_{k\in\{1,\ldots,p\}}\{C(i,k)+D(k,j)\},
\end{aligned}$$

for all $i\in\{1,\ldots,m\}$ and $j\in\{1,\ldots,n\}$. The notation $A(i,j)$ represents the scalar entry of matrix A corresponding to the i-th row and j-th column. Notice the analogy between $\oplus$, $\otimes$ and $+$, $\times$ for matrix and vector operations in the conventional algebra. Given $m\in\mathbb{N}$, the m-th max-algebraic power of $A\in\mathbb{R}_\varepsilon^{n\times n}$ is denoted by $A^{\otimes m}$ and corresponds to $A\otimes\cdots\otimes A$ (m times). Notice that $A^{\otimes 0}$ is an n-dimensional max-plus identity matrix, i.e. the diagonal and non-diagonal elements are 0 and ε, respectively. The following notation is adopted for reasons of convenience: a vector with all the components equal to 0 (resp. $-\infty$) is also denoted by 0 (resp. ε).

An (autonomous) Max-Plus-Linear (MPL) system [7, Rem. 2.75] is defined as:

$$\mathbf{x}(k)=A\otimes\mathbf{x}(k-1), \tag{1}$$

where $A\in\mathbb{R}_\varepsilon^{n\times n}$, $\mathbf{x}(k-1)=[x_1(k-1)\ldots x_n(k-1)]^T\in\mathbb{R}^n$ for $k\in\mathbb{N}$. Furthermore the state space is taken to be $\mathbb{R}^n$ (rather than $\mathbb{R}_\varepsilon^n$), which also implies that the state matrix A has to be row-finite (cf. Definition 1). We use the bold typeset for vectors and tuples, whereas the entries are denoted by the normal typeset with the same name and index. The independent variable k denotes an increasing occurrence index, whereas the state variable $\mathbf{x}(k)$ defines the (continuous) time of k-th occurrence of all events. In particular, the state component $x_i(k)$ denotes the (continuous) time of k-th occurrence of i-th event. MPL systems are characterized by deterministic dynamics, namely they are unaffected by exogenous inputs in the form of control signals or of environmental non-determinism. Since this article is based exclusively on autonomous (that is, not non-deterministic) MPL systems, the adjective will be dropped for simplicity. Finite abstractions of nonautonomous MPL systems [7, Cor. 2.82] are not discussed in this paper. The interested reader is referred to [2, Sec. III].

Definition 1 (Regular (Row-Finite) Matrix [23, Sec. 1.2]) A matrix $A \in \mathbb{R}_\varepsilon^{n\times n}$ is called regular (or row-finite) if A contains at least one element different from ε in each row. □

Example 1 Consider the following two-dimensional MPL system that models a simple railway network between two cities [23, Sec. 0.1]. Here $x_i(k)$ denotes the time of the k-th departure at station i for $i \in \{1,2\}$, and is updated as:

$$\begin{aligned} \mathbf{x}(k) &= \begin{bmatrix} 2 & 5 \\ 3 & 3 \end{bmatrix} \otimes \mathbf{x}(k-1), \quad \text{or equivalently,} \\ \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} &= \begin{bmatrix} \max\{2+x_1(k-1), 5+x_2(k-1)\} \\ \max\{3+x_1(k-1), 3+x_2(k-1)\} \end{bmatrix}. \end{aligned} \tag{2}$$

Notice that in this example A is a row-finite matrix. □

2.2 Piecewise Affine Systems

Piece-wise Affine (PWA) dynamical systems are characterized by a cover (possibly a partition) of the state space and by affine (linear, plus a constant) dynamics that are active within each set of the cover [27,33]. PWA systems are well-posed if the next state is unique once the current state is specified. PWA systems are sufficiently expressive to model a large number of physical processes, and they can approximate nonlinear dynamics with arbitrary accuracy via local linearizations at different operating points [10, p. 1864]. PWA systems have been studied by several authors.

This section discusses PWA systems generated by MPL systems. The obtained PWA systems are well-posed since MPL systems are also well-posed. The construction of a PWA system from an MPL one has combinatorial complexity: in order to improve the performance, we propose to use a backtracking approach.

An MPL system characterized by a row-finite state matrix $A \in \mathbb{R}_\varepsilon^{n\times n}$ can be expressed as a PWA system in the event domain [22, Sec. 3]. The affine dynamics, along with the corresponding region on the state space, can be constructed from coefficients $\mathbf{g} = (g_1,\ldots,g_n) \in \{1,\ldots,n\}^n$. For each i, the coefficient g_i characterizes the maximal term in the i-th state equation $x_i(k) = \max\{A(i,1)+x_1,\ldots,A(i,n)+x_n\}$, that is $A(i,j)+x_j \leq A(i,g_i)+x_{g_i}$, for all $j \in \{1,\dots,n\}$. It follows that the set of states corresponding to $\mathbf{g}$, denoted by $R_{\mathbf{g}}$, is

$$R_{\mathbf{g}} = \bigcap_{i=1}^{n} \bigcap_{j=1}^{n} \{\mathbf{x} \in \mathbb{R}^n : A(i,j)+x_j \leq A(i,g_i)+x_{g_i}\}. \tag{3}$$

Alternatively, a point or a state $\mathbf{x} \in \mathbb{R}^n$ is in $R_{\mathbf{g}}$ if $\max_{j \in \{1,\dots,n\}}\{A(i,j)+x_j\} = A(i,g_i)+x_{g_i}$, for all $i \in \{1,\dots,n\}$.

The affine dynamics that are active in $R_{\mathbf{g}}$ follow directly from the definition of $\mathbf{g}$ (see previous paragraph) as

$$x_i(k) = x_{g_i}(k-1) + A(i,g_i), \qquad i \in \{1,\dots,n\}. \tag{4}$$

Given a row-finite state matrix A, Algorithm 1 describes a general procedure to construct a PWA system corresponding to an MPL system. On the side, notice that the affine dynamics associated with a dynamical system generated by Algorithm 1 are a special case of the general PWA dynamics as defined in [33, Sec. 1].

Algorithm 1 (Generation of a PWA system from an MPL system)
Input: $A \in \mathbb{R}_{\varepsilon}^{n \times n}$, a row-finite state matrix
Output: $\mathbf{R}, \mathbf{A}, \mathbf{B}$, a PWA system over $\mathbb{R}^n$,
where $\mathbf{R}$ is a set of regions and $\mathbf{A}, \mathbf{B}$ represent a set of affine dynamics

```
initialize $\mathbf{R}, \mathbf{A}, \mathbf{B}$ with empty sets
for all $\mathbf{g} \in \{1,\dots,n\}^n$ do
  generate region $R_{\mathbf{g}}$ according to (3)
  if $R_{\mathbf{g}}$ is not empty then
    generate matrices $A_{\mathbf{g}}, B_{\mathbf{g}}$ s.t. $\mathbf{x}(k) = A_{\mathbf{g}}\mathbf{x}(k-1) + B_{\mathbf{g}}$, corresponding to (4)
    save the outcomes, $\mathbf{R} := \mathbf{R} \cup \{R_{\mathbf{g}}\}$, $\mathbf{A} := \mathbf{A} \cup \{A_{\mathbf{g}}\}$, $\mathbf{B} := \mathbf{B} \cup \{B_{\mathbf{g}}\}$
  end if
end for
```
□

The crucial observation leading to an improvement of the complexity is that it is not necessary to iterate over all possible coefficients in $\mathbf{g}$, as suggested in Algorithm 1. Instead, we can apply a backtracking technique: with this approach, we introduce partial coefficients $(g_1,\dots,g_k)$ for $k \in \{1,\dots,n\}$ and the corresponding region

$$R_{(g_1,\dots,g_k)} = \bigcap_{i=1}^{k} \bigcap_{j=1}^{n} \{\mathbf{x} \in \mathbb{R}^n : A(i,g_i)+x_{g_i} \geq A(i,j)+x_j\}. \tag{5}$$

Notice that if the region associated with some partial coefficient $(g_1,\dots,g_k)$ is empty, then the regions corresponding to the coefficients $(g_1,\dots,g_n)$ are also empty, for all $g_{k+1},\dots,g_n$. The set of all coefficients can be represented as a potential search tree. For a 2-dimensional MPL system, the potential search tree is given in Fig. 1 (left). The backtracking algorithm traverses the tree recursively, starting from the root, in a depth-first order. At each node, the algorithm checks whether the corresponding region is empty. If a region is empty, the computations over the whole sub-tree rooted at the node are skipped (pruning step).

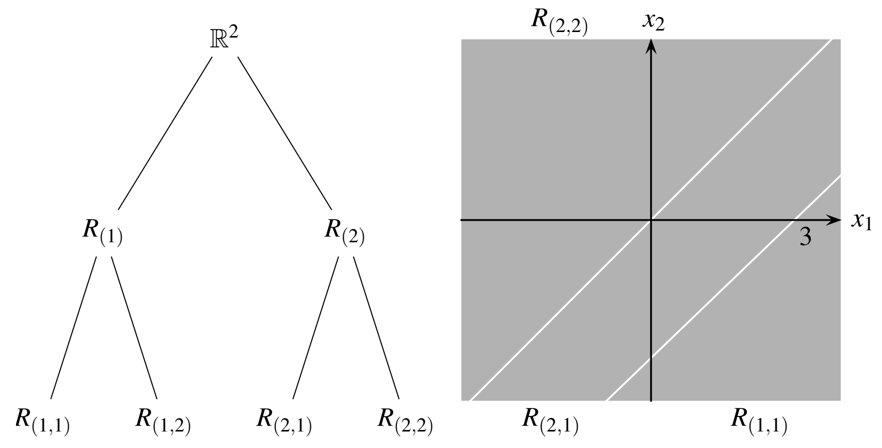

Fig. 1 (Left plot) Potential search tree for a 2-dimensional MPL system. (Right plot) Regions associated with the PWA system generated by the MPL system in (2).

Example 2 With reference to the MPL system in (2), the obtained PWA system is

$$\mathbf{x}(k) = \begin{cases} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \mathbf{x}(k-1) + \begin{bmatrix} 2 \\ 3 \end{bmatrix}, & \text{if } \mathbf{x}(k-1) \in R_{(1,1)}, \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \mathbf{x}(k-1) + \begin{bmatrix} 5 \\ 3 \end{bmatrix}, & \text{if } \mathbf{x}(k-1) \in R_{(2,1)}, \\ \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{x}(k-1) + \begin{bmatrix} 5 \\ 3 \end{bmatrix}, & \text{if } \mathbf{x}(k-1) \in R_{(2,2)}, \end{cases}$$

where $R_{(1,1)} = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \geq 3\}$, $R_{(2,1)} = \{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 \leq 3\}$, and $R_{(2,2)} = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \leq 0\}$, as depicted in Fig. 1 (right). Region $R_{(1,2)}$ does not appear since it corresponds to an empty set. As explained above, the affine dynamics corresponding to a region are characterized by set $\mathbf{g}$: for example, the affine dynamics of $R_{(2,1)}$ are given by $x_1(k) = x_2(k-1) + 5$, $x_2(k) = x_1(k-1) + 3$. □

Implementation In VeriSiMPL 2, the procedure to construct a PWA system from an MPL system is implemented in the JAVA class *Maxpl2pwa*. This class has three public members A, B and D. For initialization, this class requires a row-finite state matrix (*Ampl*) as parameter, then generates a PWA system characterized by a set of affine dynamics (in public members A,B) and a collection of regions (in the public member D). The affine dynamics that are active in the j-th region are characterized by the j-th column of both A and B. Each column of A and the corresponding column of B contain respectively the coefficients $[g_1, \ldots, g_n]^T$ and the constants $[A(1,g_1), \ldots, A(n,g_n)]^T$. The data structure for D will be discussed in Section 2.3. □

2.3 Difference Bound Matrices

This section introduces the definition of a Difference-Bound Matrix (DBM) [19, Sec. 4.1] and some of the essential operations on DBM of use in this work. More precisely, we discuss: the intersection of two DBM, the complement of a DBM, the canonical-form representation of a DBM, the orthogonal projection of a DBM, the emptiness checking on a DBM, the image of a DBM w.r.t. affine dynamics, and the inverse image of a DBM w.r.t. affine dynamics. DBM will be used extensively in the abstraction procedure of MPL systems (cf. Sections 3 and 4).

Definition 2 (Difference-Bound Matrix) A DBM in $\mathbb{R}^n$ is the intersection of finitely many sets defined as $x_j - x_i \bowtie_{i,j} \alpha_{i,j}$, where $\bowtie_{i,j} \in \{<, \leq\}$ denotes an inequality sign, the specified number $\alpha_{i,j} \in \mathbb{R} \cup \{+\infty\}$ represents the upper bound, for $i, j \in \{0, \ldots, n\}$, and the value of the special variable x_0 is always equal to 0. The sets are subsets of $\mathbb{R}^n$ that are characterized by the values of variables $x_1, \ldots, x_n$. $\square$

Implementation Inspired by VeriSiMPL in [2], VeriSiMPL 2 represents a DBM in $\mathbb{R}^n$ as a two-element array of matrices: the first element of the array is an $(n+1)$-dimensional[1] real-valued matrix representing the upper bound α, and the second element is an $(n+1)$-dimensional 0-1 matrix representing the value of $\bowtie$. More precisely, the $(i+1, j+1)$-th element of the first and second matrix represents respectively the upper bound and the strictness of the sign of $x_j - x_i$[2], for $i \in \{0, \ldots, n\}$ and $j \in \{0, \ldots, n\}$ (the nonstrict sign $\leq$ corresponds to 1 and the strict sign $<$ corresponds to 0). Furthermore, a collection of DBM is represented as a list of two-element arrays. As we discussed in Section 2.2, the implementation of the procedure to construct a PWA system from an MPL system, JAVA class *Maxpl2pwa*, has a public member D. The variable D corresponds to a collection of DBM, which means, in VeriSiMPL 2, that D is implemented as an arraylist of DBM. $\square$

Next, we discuss the implementation of a number of operations over DBM that are key for the abstraction procedure.

2.3.1 Intersection of DBM

We describe a procedure to compute the intersection of two DBM, which is again a DBM. We denote the two DBM that are going to be intersected as D and E. The intersection is denoted by F, i.e. $F = D \cap E$. The set associated with $x_j - x_i$ in F is the intersection of the set associated with $x_j - x_i$ in D and in E, for every i, j. Recall that each set is characterized by the upper bound and the strictness of the sign (cf. Definition 2). The bound used for F is simply the minimum of the bounds in D and E. In the case of strictness of the inequality signs in F, we need to consider all possible combinations of bounds and strictness in D and E. More precisely, if the upper bound of D and E is different, the strictness of F equals the strictness of the one with smaller bound. If the upper bound (of D and E) is the same and the sign of D and E is not

[1] The matrices are $(n+1)$-dimensional rather than n-dimensional because we need to store x_0 as well.

[2] Notice that, the $(i+1, j+1)$-th element corresponds to $x_j - x_i$ (not $x_i - x_j$).

strict, then the sign of F is not strict. Finally if the upper bound (of D and E) is the same and either D or E has a strict sign, then the sign of F is strict. In order to derive an expression for the maximum number of operations, let us assume that both D and E are DBM in $\mathbb{R}^n$. Then the maximum number of operations is $\mathscr{O}(n^2)$, where the notation $\mathscr{O}$ describes an upper bound on the limiting behaviour of a function when its argument tends towards infinity [32, Sec. 3.4.1].

Example 3 Consider $D = \{\mathbf{x} \in \mathbb{R}^3 : x_1 - x_2 \leq 5, x_2 - x_3 < 3\}$ and $E = \{\mathbf{x} \in \mathbb{R}^3 : x_1 - x_3 < 7, x_2 - x_3 \leq 3\}$. The intersection of D and E is given by $F = \{\mathbf{x} \in \mathbb{R}^3 : x_1 - x_2 \leq 5, x_1 - x_3 < 7, x_2 - x_3 < 3\}$. □

Implementation In VeriSiMPL 2, the procedure to compute the intersection of two DBM is implemented in the JAVA class *Dbm_and*. This class has one public member *myF* corresponding to the intersection of the two DBM. For initialization, this class requires two DBM as parameters and then computes the intersection of the given two DBM. □

2.3.2 Complement of a DBM

We describe a procedure to compute the complement of a DBM w.r.t. the state space $\mathbb{R}^n$. The complement of a DBM is obtained as a union of finitely many DBM, which we want to be pairwise disjoint. The general procedure is presented after an example.

Example 4 Let us determine the complement of $D = \{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_2 \leq 5, x_2 - x_3 < 3, x_3 - x_4 < 1\}$. The DBM D is an intersection of 3 sets, i.e. $set_1 = \{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_2 \leq 5\}$, $set_2 = \{\mathbf{x} \in \mathbb{R}^4 : x_2 - x_3 < 3\}$, $set_3 = \{\mathbf{x} \in \mathbb{R}^4 : x_3 - x_4 < 1\}$. The first step is the computation of the complement of each set w.r.t. $\mathbb{R}^4$, i.e. $set_1^c = \{\mathbf{x} \in \mathbb{R}^4 : x_2 - x_1 < -5\}$, $set_2^c = \{\mathbf{x} \in \mathbb{R}^4 : x_3 - x_2 \leq -3\}$, $set_3^c = \{\mathbf{x} \in \mathbb{R}^4 : x_4 - x_3 \leq -1\}$. The complement of D is obtained as $set_1^c \cup (set_1 \cap set_2^c) \cup (set_1 \cap set_2 \cap set_3^c)$, i.e. $\{\mathbf{x} \in \mathbb{R}^4 : x_2 - x_1 < -5\} \cup \{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_2 \leq 5, x_3 - x_2 \leq -3\} \cup \{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_2 \leq 5, x_2 - x_3 < 3, x_4 - x_3 \leq -1\}$. □

The general procedure to determine the complement of a DBM D in $\mathbb{R}^n$ is as follows. Suppose that the DBM D is an intersection of m sets, denoted by $set_1, \ldots, set_m$. First we compute the complement of each set w.r.t. $\mathbb{R}^n$. We denote the complement of set_i as set_i^c for $i \in \{1, \ldots, m\}$. The complement of D is given by $\cup_{i=1}^m ((\cap_{j=1}^{i-1} set_j) \cap set_i^c)$, where $\cap_{j=1}^0 set_j$ is set to $\mathbb{R}^n$. The maximum number of operations is in the order of $\mathscr{O}(n^3)$.

Implementation In VeriSiMPL 2, the procedure to determine the complement of a DBM is implemented in the JAVA class *Dbm_complement*. This class has one public member *D_comp* corresponding to the complement of the DBM. The initialization of this class requires a DBM as the parameter, and then it computes the complement of the given DBM. □

2.3.3 Canonical-Form Representation and Orthogonal Projection of a DBM

Each DBM admits an equivalent and unique canonical-form representation, which is a DBM with the tightest possible bounds [19, Sec. 4.1]. Since obtaining the canonical-form representation of a DBM is equivalent to the all-pairs shortest path problem over the corresponding potential graph [19, Sec. 4.1], the Floyd-Warshall algorithm [21] can be used over the graph, with a complexity that is cubic w.r.t. its dimension.

One advantage of the canonical-form representation is that it is straightforward to compute orthogonal projections w.r.t. a subset of its variables: this is simply performed by deleting rows and columns corresponding to the complementary variables [19, Sec. 4.1]. The orthogonal projection of a DBM in canonical form is again in canonical form [19, Obs. 1].

Example 5 Let us consider the following DBM: $\{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_4 \le -3, x_2 - x_1 \le -3, x_2 - x_4 \le -3, x_3 - x_1 \le 2\}$. In the description of this set we have omitted the inequalities that are unbounded, say e.g. $x_3 - x_4 < +\infty$. One can show that the canonical-form representation is given by $\{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_4 \le -3, x_2 - x_1 \le -3, x_2 - x_4 \le -6, x_3 - x_1 \le 2, x_3 - x_4 \le -1\}$. Notice that the upper bound of $x_2 - x_4$ and $x_3 - x_4$ is tighter. Moreover, the orthogonal projection of the DBM w.r.t. $\{x_1, x_2\}$ is $\{\mathbf{x} \in \mathbb{R}^2 : x_2 - x_1 \le -3\}$. □

Implementation In VeriSiMPL 2, the Floyd-Warshall algorithm [21] has been implemented in the JAVA class *Floyd_warshall*. This class has one public member *myD* corresponding to the DBM in the canonical form. For initialization, this class requires a DBM as parameter and then generates the canonical-form representation of the given DBM. □

2.3.4 Checking Emptiness of a DBM

We describe a procedure to check whether a DBM is empty. By using the potential graph representation [19, Sec. 4.1], the unfeasible sets of constraints are only those which form a circuit with a strictly negative weight in the graph. In other words, this circuit corresponds to a constraint $x_i - x_i \bowtie_{i,i} \alpha_{i,i}$, with $\bowtie_{i,i} \in \{<, \le\}$ and $\alpha_{i,i} < 0$, which is not feasible. As a consequence, in order to test whether a DBM is empty or not, we simply have to check for the existence of such a circuit: this can be achieved by the Bellman-Ford algorithm [9, Sec. 5], which is cubic w.r.t. the dimension of its input. Whenever a DBM is in canonical form, testing for strictly negative cycles can be reduced to checking whether there is an i such that $\bowtie_{i,i}$ is $<$ or $\alpha_{i,i} < 0$. In this instance, the complexity of emptiness checking is linear w.r.t. dimension of the DBM.

Example 6 Let us consider the following DBM $\{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_4 \le -3, x_2 - x_1 \le -3, x_2 - x_4 \le -3, x_3 - x_1 \le 2, x_0 - x_0 = 0, x_1 - x_1 = 0, x_2 - x_2 = 0, x_3 - x_3 = 0, x_4 - x_4 = 0\}$. Using the procedure discussed in Section 2.3.3, one can show that the canonical-form representation is given by $\{\mathbf{x} \in \mathbb{R}^4 : x_1 - x_4 \le -3, x_2 - x_1 \le -3, x_2 - x_4 \le -6, x_3 - x_1 \le 2, x_3 - x_4 \le -1, x_0 - x_0 = 0, x_1 - x_1 = 0, x_2 - x_2 = 0, x_3 - x_3 = 0, x_4 - x_4 = 0\}$. Since the upper bounds of $x_i - x_i$ for $i \in \{0, 1, 2, 3, 4\}$ are 0 and the corresponding signs are not strict, the DBM is not empty. □

Implementation In VeriSiMPL 2, the procedure to check emptiness of a DBM has been implemented in the JAVA class *Dbm_isempty*. The procedure consists of two steps: first we generate the canonical-form representation and then check whether the DBM is empty. For initialization, this class requires a DBM as the parameter. This class stores the canonical-form representation of the DBM in the public member *myD*. The public member *b* equals *true* if the DBM is empty and equals *false* if the DBM is not empty. □

2.3.5 Image and Inverse Image of a DBM

We discuss procedures to compute the image and the inverse image of a DBM w.r.t. affine dynamics. In the case of a union of finitely many DBM, the image and the inverse image w.r.t. affine dynamics can be obtained by applying the preceding procedures to each DBM.

Each region and the corresponding affine dynamics of the PWA system generated by an MPL system (cf. Algorithm 1) can be characterized by DBM. From (3), each region of the PWA system generated by a row-finite max-plus matrix is a DBM in $\mathbb{R}^n$. Each affine dynamics (4) can generate a DBM in $\mathbb{R}^{2n}$, which comprises points $(\mathbf{x}(k-1),\mathbf{x}(k)) \in \mathbb{R}^n \times \mathbb{R}^n$ such that $\mathbf{x}(k)$ is the image of $\mathbf{x}(k-1)$, i.e. $\mathbf{x}(k) = A \otimes \mathbf{x}(k-1)$. More precisely, the DBM is obtained by rewriting the expression of the affine dynamics as $\bigcap_{i=1}^{n}\{(\mathbf{x}(k-1),\mathbf{x}(k)) : x_i(k) - x_{g_i}(k-1) \leq A(i,g_i)\} \cap \bigcap_{i=1}^{n}\{(\mathbf{x}(k-1),\mathbf{x}(k)) : x_i(k) - x_{g_i}(k-1) \geq A(i,g_i)\}$.

Proposition 1 ([2, Th. 1]) *The image and the inverse image of a DBM with respect to affine dynamics (in particular the PWA expressions in* (3)-(4) *generated by an MPL system) is a DBM.*

The general procedure to compute the image of a DBM in $\mathbb{R}^n$ w.r.t. affine dynamics mapping $\mathbb{R}^n \to \mathbb{R}^n$ involves: 1) computing the cross product of the DBM and $\mathbb{R}^n$; then 2) intersecting the cross product with the DBM generated by the expression of the affine dynamics; 3) calculating the canonical form of the obtained intersection; and finally 4) projecting the canonical-form representation over $\{x_1(k),\ldots,x_n(k)\}$. The maximum number of operations for the procedure depends critically on the third step and is $\mathcal{O}(n^3)$ [2, p. 3043]. The illustration of the procedure to compute the image for $n = 1$ is depicted in Fig. 2 (left).

Example 7 Let us compute the image of $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, x_1 - x_2 \leq 0\}$ w.r.t. $x_1' = x_2 + 5, x_2' = x_2 + 3$ by using the above procedure. The cross product of the DBM and $\mathbb{R}^2$ is $\{(\mathbf{x},\mathbf{x}') \in \mathbb{R}^4 : 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, x_1 - x_2 \leq 0\}$. The intersection of the cross product and the DBM generated by the expression of the affine dynamics is $\{(\mathbf{x},\mathbf{x}') \in \mathbb{R}^4 : 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, x_1 - x_2 \leq 0, x_1' - x_2 = 5, x_2' - x_2 = 3\}$. The canonical form of the obtained intersection is $\{(\mathbf{x},\mathbf{x}') \in \mathbb{R}^4 : 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1, 5 \leq x_1' \leq 6, 3 \leq x_2' \leq 4, 0 \leq x_2 - x_1 \leq 1, 5 \leq x_1' - x_1 \leq 6, 3 \leq x_2' - x_1 \leq 4, x_1' - x_2 = 5, x_2' - x_2 = 3, x_2' - x_1' = -2\}$. The projection w.r.t. $\{x_1', x_2'\}$ is computed by removing all inequalities containing x_1 or x_2, which yields $\{\mathbf{x}' \in \mathbb{R}^2 : 5 \leq x_1' \leq 6, 3 \leq x_2' \leq 4, x_2' - x_1' = -2\}$. □

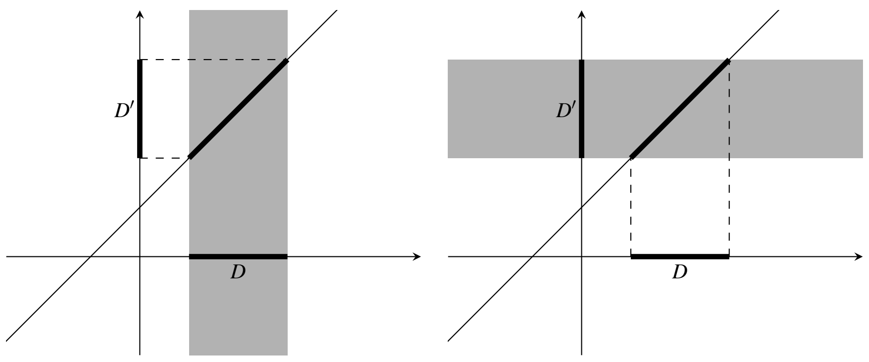

Fig. 2 The left and right plots illustrate the algorithms to determine the image and inverse image of a DBM w.r.t. affine dynamics, respectively. In the left plot D' is the image of D, whereas in the right plot D is the inverse image of D'.

Implementation In VeriSiMPL 2, the procedure to determine the image of a DBM w.r.t. affine dynamics has been implemented in the JAVA class *Dbm_image*. This class has one public member *Dy* corresponding to the image of the given DBM w.r.t. the given affine dynamics. Furthermore this class requires affine dynamics and a DBM as inputs for initialization. The affine dynamics are characterized by two column vectors, representing the indices of state variables and the constants (cf. Section 2.2). □

The procedure to compute the inverse image of a DBM in $\mathbb{R}^n$ w.r.t. affine dynamics mapping $\mathbb{R}^n \to \mathbb{R}^n$ is similar with the procedure for computing the image. The difference lies in the last step: the canonical-form representation is projected over $\{x_1(k-1),\ldots,x_n(k-1)\}$. The maximum number of operations for computing the inverse image is the same as that for computing the image [2, p. 3043].

Implementation In VeriSiMPL 2, the procedure to determine the inverse image of a DBM w.r.t. affine dynamics has been implemented in the JAVA class *Dbm_invimage*. This class has one public member *Dx* corresponding to the inverse image of the given DBM w.r.t. the given affine dynamics. The initialization of this class requires affine dynamics (as before), a DBM, and the dimension of the state space. □

Let us mention that the procedure discussed in this section has been extended to the forward and backward reachability analysis of MPL systems [4,3,5]. Next, we discuss the modeling framework that is going to be used in the abstraction procedure (cf. Sections 3 and 4).

2.4 Transition Systems

This section introduces the notion of transition systems, a standard class of models to represent hardware and software systems [8, Sec. 2.1].

Definition 3 (Transition System [8, Def. 2.1]) A transition system *TS* is characterized by a quintuple $(S, \longrightarrow, I, AP, L)$ where

– S is a set of states,
– $\longrightarrow\ \subseteq S \times S$ is a transition relation,
– $I \subseteq S$ is a set of initial states,
– AP[3] is a set of atomic propositions, and
– $L : S \to 2^{AP}$ is a labeling function.

TS is called finite if the cardinality of S and AP is finite. □

For convenience, we write $s \longrightarrow s'$ instead of $(s,s') \in \longrightarrow$. The behavior of a transition system can be described as follows. The transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\longrightarrow$. If a state has more than one outgoing transition, the "next" transition is chosen in a purely nondeterministic fashion. Recall that 2^{AP} denotes the power set of AP. The labeling function relates each state to set of atomic propositions that are satisfied over the state.

Definition 4 (Direct Predecessors and Direct Successors [8, Def. 2.3]) Let $TS = (S, \longrightarrow, I, AP, L)$ be a transition system. For $s \in S$, the set of direct successors and direct predecessors of s are defined respectively as

$$Post(s) = \left\{s' \in S : s \longrightarrow s'\right\} \qquad \text{and} \qquad Pre(s) = \left\{s' \in S : s' \longrightarrow s\right\}. \qquad \square$$

The notations for the sets of direct successors and direct predecessors are expanded to subsets of S in the obvious way (i.e. pointwise extension): for $C \subseteq S$, let

$$Post(C) = \bigcup_{s \in C} Post(s) \qquad \text{and} \qquad Pre(C) = \bigcup_{s \in C} Pre(s).$$

A transition system $TS = (S, \longrightarrow, I, AP, L)$ is called deterministic if $|I| \leq 1$ and $|Post(s)| \leq 1$ for all states s (this is called action deterministic in [8, Def. 2.5]). A path of a transition system TS is a sequence of states starting from some initial state; it evolves according to the transition relation; and it cannot be prolonged, i.e. either the path is infinite or the path is finite and ends in a terminal state [8, Defs. 2.4 and 3.6]. A trace of a path is defined as the finite or infinite word over the alphabet 2^{AP} obtained by applying the labeling function to all the states in the path. The set of traces of a transition system TS is defined as set of traces associated to all paths in TS [8, p. 98].

Example 8 Consider the example of a transition system in Fig. 3. The set of states is $S = \{s_0, s_1, s_2, s_3\}$. The set of initial states is $I = \{s_0, s_2\}$. The set of atomic propositions is $AP = \{a, b\}$, with the labeling function: $L(s_0) = \emptyset$, $L(s_1) = \emptyset$, $L(s_2) = \{a\}$, $L(s_3) = \{b\}$. □

2.5 Specifications (or Formal Properties)

This section introduces (propositional) Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), which are logical formalisms that are suited for specifying properties [8, Chs. 5 and 6]. The syntax and semantics of LTL and CTL are discussed.

[3] The notation AP does not represent the multiplication of matrix A by matrix P.

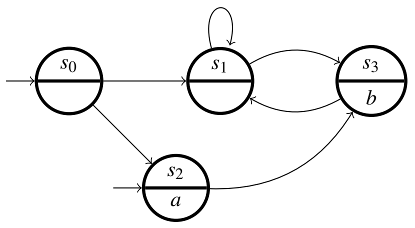

Fig. 3 Graphical representation of the transition system described in Example 8.

2.5.1 Linear Temporal Logic

LTL formulae are recursively defined over a set of atomic propositions, by Boolean and temporal operators. More formally, the syntax of LTL formulae is defined as follows:

Definition 5 (Syntax of Linear Temporal Logic [8, Def. 5.1]) LTL formulae over the set AP of atomic propositions are formed according to the following grammar:

$$\varphi ::= \mathit{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

where $a \in AP$. □

Boolean operators are $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction), whereas temporal operators are $\bigcirc$ (next), U (until), $\Box$ (always), and $\Diamond$ (eventually). The until operator allows to derive the temporal modalities $\Diamond$ and $\Box$ [8, p. 232]. The $\bigcirc$-modality is a unary prefix operator and requires a single LTL formula as its argument. Formula $\bigcirc\varphi$ holds at the current moment, if φ holds in the next "step". The U-modality is a binary infix operator and requires two LTL formulae as argument. Formula $\varphi_1 \mathsf{U} \varphi_2$ holds at the current moment, if there is some future moment for which φ_2 holds and φ_1 holds at all moments until that future moment. The $\Diamond$-modality and $\Box$-modality are unary prefix operators and require a single LTL formula as argument: formula $\Diamond\varphi$ is satisfied if φ will be true eventually in the future, whereas formula $\Box\varphi$ is satisfied if φ holds from now on forever.

LTL formulae stand for properties of paths (in fact, the associated trace). This means that a path can either fulfill an LTL-formula or not. An infinite path satisfies an LTL formula φ if the trace of the path satisfies φ [8, p. 236]. Notice that the trace of an infinite path is an infinite word over the alphabet 2^{AP}. A transition system satisfies an LTL formula if all paths of the transition system satisfy the LTL formula [8, p. 237].

2.5.2 Computation Tree Logic

CTL formulae are classified as state and path formulae. State formulae are assertions about atomic propositions over the states. Path formulae express temporal properties of paths. Path formulae in CTL are built by the next-step and until operators, however no nesting of temporal modalities is allowed.

Definition 6 (Syntax of Computation Tree Logic [8, Def. 6.1]) CTL state formulae over the set AP of atomic propositions are formed according to the following grammar:

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL path formulae are formed according to the following grammar:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathsf{U} \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae. □

CTL distinguishes between state formulae and path formulae. State formulae express a property of a state, while path formulae express a property of a path, i.e. a sequence of states. The temporal operators $\bigcirc$ and U have the same meaning as in LTL and are path operators. Path formulae can be turned into state formulae by prefixing them with either the path quantifier $\exists$ (pronounced "for some path") or the path quantifier $\forall$ (pronounced "for all paths"). Formula $\exists\varphi$ holds in a state if there exists some path satisfying φ that starts in that state. Dually, $\forall\varphi$ holds in a state if all paths that start in that state satisfy φ.

The universal (resp. existential) fragment of CTL is a CTL formula where the state formulae are required to be in positive normal form, (negations may only occur adjacent to atomic propositions) and do not contain existential (resp. universal) path quantifiers.

Definition 7 (Universal Fragment of CTL [8, Def. 7.74]) The universal fragment of CTL, denoted $\forall$CTL, consists of the state formulae Φ and path formulae φ given, for $a \in AP$, by

$$\begin{aligned}\Phi &::= true \mid false \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \forall\varphi \\ \varphi &::= \bigcirc\Phi \mid \Phi_1 \mathsf{U} \Phi_2 \mid \Phi_1 \mathsf{R} \Phi_2\end{aligned}$$

□

Definition 8 (Existential Fragment of CTL [8, Def. 7.78]) The existential fragment of CTL, denoted $\exists$CTL, consists of the state formulae Φ and path formulae φ given, for $a \in AP$, by

$$\begin{aligned}\Phi &::= true \mid false \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \exists\varphi \\ \varphi &::= \bigcirc\Phi \mid \Phi_1 \mathsf{U} \Phi_2 \mid \Phi_1 \mathsf{R} \Phi_2\end{aligned}$$

□

The positive normal forms require the use of the release operator R as a basic operator in the logic. The release operator is dual to the until operator: formula $\varphi_1 \mathsf{R} \varphi_2$ holds for a word if φ_2 always holds, a requirement that is released as soon as φ_1 becomes valid.

2.6 Equivalences and Abstractions

Abstraction is a fundamental concept that enables the analysis of large [8, Ex. 7.53] or even infinite [8, Ex. 7.54] transition systems. An abstraction is identified by a set of abstract states $\hat{S}$; an abstraction function f, that associates to each (concrete) state

s of the transition system TS the abstract state $f(s)$ that represents it; and a set AP of atomic propositions labeling the concrete and abstract states. Abstractions differ in the choice of the set $\hat{S}$ of abstract states, the abstraction function f, and the relevant propositions AP.

Typically an abstract transition system simulates the corresponding concrete transition system. Simulation relations are used as a basis for abstraction techniques, where the idea is to replace the model to be verified by a smaller abstract model and to verify the latter instead of the original one. Simulation relations are preorders on the state space requiring that whenever s' simulates s, state s' can mimic all stepwise behavior of s, but the reverse is not guaranteed. The formal definition of the simulation order is given below.

Definition 9 (Simulation Order [8, Def. 7.47]) Let $TS_i = (S_i, \longrightarrow_i, I_i, AP, L_i)$, $i \in \{1,2\}$, be transition systems over AP. A simulation for (TS_1, TS_2) is a binary relation $\mathscr{R} \subseteq S_1 \times S_2$ such that

1. for each $s_1 \in I_1$ there exists $s_2 \in I_2$ such that $(s_1, s_2) \in \mathscr{R}$
2. for all $(s_1, s_2) \in \mathscr{R}$ it holds:
   (a) $L_1(s_1) = L_2(s_2)$
   (b) if $s_1' \in Post(s_1)$ then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathscr{R}$.

A transition system TS_1 is simulated by TS_2 (or equivalently, TS_2 simulates TS_1) if there exists a simulation $\mathscr{R}$ for (TS_1, TS_2). $\square$

We briefly outline the essential ideas of abstractions that are obtained by aggregating disjoint sets of concrete states into single abstract states. Abstraction functions map concrete states onto abstract ones, such that abstract states are associated to equally labeled concrete states [8, Def. 7.50].

The procedure to construct an abstract transition system is as follows [8, Def. 7.51]. The abstract transition system TS_f originates from TS by identifying all states that are represented by the same abstract state under abstraction function f. An abstract state is initial whenever it represents an initial concrete state. Similarly, there is a transition from abstract state $f(s)$ to state $f(s')$ if there is a transition from s to s'.

Proposition 2 ([8, Lem. 7.52]) *Let $TS = (S, \longrightarrow, I, AP, L)$ be a (concrete) transition system, $\hat{S}$ a set of (abstract) states, and $f : S \to \hat{S}$ an abstraction function. Then TS_f simulates TS.*

Proposition 3 ([8, Cor. 7.68 and Th. 7.70]) *Let TS_2 simulates TS_1, assume TS_1 does not have terminal states, let φ be a linear-time property. If TS_2 satisfies φ, then TS_1 also satisfies φ.*

State s in transition system TS is called terminal if and only if $Post(s) = \emptyset$ [8, Def. 2.4]. The result also applies to LTL formulae, since any LTL formula is a linear-time property [8, Defs. 3.10 and 5.6]. In general the reverse of the preceding proposition is not true. More precisely, if TS_2 does not satisfy φ, we cannot deduce that TS_1 does not satisfy φ since the trace of paths that violate φ might be behaviors that TS_1 cannot perform at all. A similar result holds for a (large) fragment of CTL, as shown in the following proposition:

Proposition 4 ([8, Ths. 7.76 and 7.79]) *Let TS_2 simulates TS_1, assume both TS_1 and TS_2 do not have terminal states, let φ be a universal fragment of CTL or an existential fragment of CTL. If TS_2 satisfies φ, then TS_1 also satisfies φ.*

Bisimulation equivalence aims to identify transition systems with the same branching structure, and which thus can simulate each other in a stepwise manner [8, p. 451].

Definition 10 (Bisimulation Equivalence [8, Def. 7.1]) For $i \in \{1,2\}$ let TS_i be transition systems over AP, i.e. $TS_i = (S_i, \longrightarrow_i, I_i, AP, L_i)$. A bisimulation for (TS_1, TS_2) is a binary relation $\mathscr{R} \subseteq S_1 \times S_2$ such that

1. for each $s_1 \in I_1$ there exists $s_2 \in I_2$ such that $(s_1, s_2) \in \mathscr{R}$ and for each $s_2 \in I_2$ there exists $s_1 \in I_1$ such that $(s_1, s_2) \in \mathscr{R}$
2. for all $(s_1, s_2) \in \mathscr{R}$ it holds that
   (a) $L_1(s_1) = L_2(s_2)$
   (b) if $s_1' \in Post(s_1)$ then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathscr{R}$
   (c) if $s_2' \in Post(s_2)$ then there exists $s_1' \in Post(s_1)$ with $(s_1', s_2') \in \mathscr{R}$.

Transition systems TS_1 and TS_2 are bisimulation-equivalent (bisimilar, for short) if there exists a bisimulation $\mathscr{R}$ for (TS_1, TS_2). □

Bisimulation equivalence denotes the possibility of mutual, stepwise simulation. Bisimulation equivalence preserves all formulae that can be expressed in both LTL and CTL [8, p. 469]. This result allows performing model checking on the bisimulation quotient transition system while preserving both affirmative and negative outcomes of the model checking.

3 Finite Abstractions of Max-Plus-Linear Systems

We now develop a framework for the formal verification of MPL systems. Specifically, we check whether an MPL system with a predefined set of initial states $\mathscr{X}_0$ satisfies a specification over a fixed set of finitely many atomic propositions AP. The specification is expressed as an LTL formula or as a universal/existential fragment of CTL formulae.

The abstraction procedure consists of partitioning the state space (cf. Section 3.1) and of computing the transition relation (cf. Section 3.2). As discussed, in general the abstraction is an over-approximation of the original MPL system (cf. Section 2.6): in order to obtain a more precise abstraction, a partition-refinement technique is discussed in Section 3.3. In Section 3.4 we describe a procedure to determine the initial states of the abstract transition system. Finally, the interface with NuSMV model checker is discussed in Section 3.5.

Recall that the idea of abstraction is to replace a model to be verified by a smaller abstract model, and to verify the latter instead of the original one. Let us introduce a characterization of the MPL system as an (infinite-space) transition system.

Definition 11 (Transition System Associated with an MPL System) Consider an MPL system as in (1), with $\mathscr{X}_0$ as the set of its initial conditions, and a set of atomic propositions AP together with the corresponding labeling function L. The associated transition system TS is a tuple $(S, \longrightarrow, I, AP, L)$ where

– set of states S is $\mathbb{R}^n$,
– there exists a transition relation $\mathbf{x} \longrightarrow \mathbf{x}'$ if $\mathbf{x}' = A \otimes \mathbf{x}$, and
– set of initial states I is $\mathscr{X}_0$. $\square$

In this work, we assume the set of states satisfying each atomic proposition is a DBM, i.e. for each $a \in AP$, the set of states $\{\mathbf{x} : a \in L(\mathbf{x})\}$ is a DBM. Furthermore, we assume that the set of initial states $\mathscr{X}_0$ is a union of finitely many DBM. These assumptions allow easily manipulating this structure throughout the whole abstraction procedure.

Next, we introduce the abstraction procedure by discussing how to obtain abstract states and transitions.

3.1 States: Partitioning Procedure

We partition the state space S into finitely many sets, assumed to be non-overlapping and non-empty subsets, which are also called "blocks" [8, Def. 7.29]. An abstraction function f is constructed based on the obtained partition: the abstraction function f maps each state in the same block to a unique abstract state. The partition is denoted by Π_0 that has the following properties: Π_0 is an AP partition, each block is a DBM, and the dynamics within each block are affine. A partition is called an AP partition if the labeling function L maps all states in the same block to the same set of atomic propositions [8, Def. 7.31].

The approach to determine Π_0 is as follows. We first determine an AP partition of S, denoted by Π_{AP}, where each block is a DBM. We then determine a partition Π_{AD} of S (AD stands for "affine dynamics"), where each block is a DBM and the dynamics within each block are affine. Finally, the partition Π_0 is defined as the refinement of Π_{AP} and Π_{AD}, i.e. $\mathscr{R}_{\Pi_0} = \mathscr{R}_{\Pi_{AP}} \cap \mathscr{R}_{\Pi_{AD}}$ [8, Rem. 7.30].

3.1.1 AP Partition

Standard algorithms (cf. [8, Alg. 29]) cannot be used since they require finite cardinality of S. We discuss a procedure to generate an AP partition of S, where each block is a DBM. First we compute the coarsest AP partition, obtained by defining a block for each $\mathbf{a} \in 2^{AP}$, as the inverse image of $\mathbf{a}$ w.r.t. the labeling function L, i.e. $L^{-1}(\mathbf{a}) = \{\mathbf{x} : L(\mathbf{x}) = \mathbf{a}\} = \cap_{a\in\mathbf{a}}\{\mathbf{x} : a \in L(\mathbf{x})\} \setminus \cup_{a\in AP\setminus\mathbf{a}}\{\mathbf{x} : a \in L(\mathbf{x})\}$. Notice that in general each block is a union of finitely many DBM, since the set difference between two DBM is a union of finitely many DBM. Finally the coarsest AP partition is refined such that each block is a DBM. The maximum number of operations depends on the number of atomic propositions and dimension of the state space. If the number of atomic propositions increases, the maximum number of operations rises exponentially. If dimension of the state space increases, the maximum number of operations has a polynomial growth rate.

Example 9 Suppose that $AP = \{a\}$ and the set of states satisfying a is the following stripe $\{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$. The coarsest AP partition contains two blocks, i.e. $\{\mathbf{x} : 0 \le x_1 - x_2 < 3\}$ and $\{\mathbf{x} : x_1 - x_2 < 0\} \cup \{\mathbf{x} : x_1 - x_2 \ge 3\}$. Since the latter

Table 1 Data structure for storing the states that satisfy and that do not satisfy each atomic proposition in the JAVA class *PartAgainstAP*.

a [0]	$\neg a$ [1]	b [2]	$\neg b$ [3]	c [4]	$\neg c$ [5]	…
DBM	(DBM, …, DBM)	DBM	(DBM, …, DBM)	DBM	(DBM, …, DBM)	…

block is a union of two DBM, it is refined into two blocks, i.e. $\{\mathbf{x} : x_1 - x_2 < 0\}$ and $\{\mathbf{x} : x_1 - x_2 \geq 3\}$. The resulting *AP* partition contains three blocks, i.e. $B_1 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$, $B_2 = \{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$, and $B_3 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \geq 3\}$ as shown in Fig. 4. □

Implementation The procedure to generate an *AP* partition is implemented in the JAVA class *PartAgainstAP* in VeriSiMPL 2. This class has one public member *partAP* corresponding to the *AP* partition. The initialization of *PartAgainstAP* requires a list of DBM, where each DBM corresponds to the set of states satisfying an atomic proposition. The procedure to compute an *AP* partition consists of two steps. In the first step, for each atomic proposition we compute the states that do not satisfy the atomic proposition. In the second step, we determine the coarsest *AP* partition.

With focus on the first step, for each atomic proposition we compute the states that do not satisfy the atomic proposition. This can be obtained by determining the complement of the states that satisfy each atomic proposition. Since the set of states satisfying an atomic proposition is a DBM, we can use the JAVA class *DBM_complement* (cf. Section 2.3.2). An array is used to store the states that satisfy and do not satisfy each atomic proposition. Each element in the array is a list of DBM. The number of DBM in each list is either zero or one or more than one. The array looks as in Table 1 (notice that, in JAVA, all indices start from 0, instead of 1).

In the second step, *PartAgainstAP* computes the states for all possible combinations consisting of atomic propositions in *AP* or of their negations. For instance, if there are 3 atomic propositions a, b, c, then we calculate all 8 possible combinations: abc, $ab\neg c$, $a\neg bc$, $a\neg b\neg c$, $\neg abc$, $\neg ab\neg c$, $\neg a\neg bc$, $\neg a\neg b\neg c$. This can be encoded as a binary number: 0 if the atomic proposition is satisfied and 1 if the atomic proposition is not satisfied. By using this encoding, the previous combinations can be interpreted as 000, 001, 010, 011, 100, 101, 110, 111. This encoding helps us to keep track of the atomic propositions that are satisfied based on the binary number expressing the location index in the array. *PartAgainstAP* stores the above-explained combinations in an array. Each element in the array is either an empty list or a list of DBM. Each DBM represents a block and this array represents the final *AP* partition of the state space. The array looks as in Table 2. □

Table 2 Data structure of the final *AP* Partition in the JAVA class *PartAgainstAP*.

abc [0]	$ab\neg c$ [1]	$a\neg bc$ [2]	…	$\neg a\neg b\neg c$ [7]
DBM	(DBM, …, DBM)	(DBM, …, DBM)	…	(DBM, …, DBM)

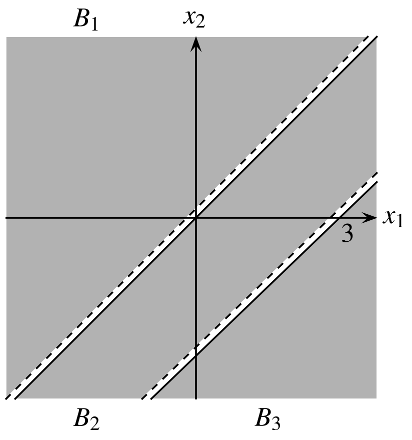

Fig. 4 *AP* partition of $\mathbb{R}^2$ where all blocks are DBM.

3.1.2 AD Partition

We now discuss an approach to construct a partition of S, where each block is a DBM and the dynamics in each block are affine. This approach is first proposed in [2]. The idea is to refine the regions of the PWA system generated by the state matrix.

Definition 12 (Adjacent Regions [2, Def. 9]) Let $R_{\mathbf{g}}$ and $R_{\mathbf{g}'}$ be regions generated by an n-dimensional state-space matrix. We say that they are adjacent ($R_{\mathbf{g}} > R_{\mathbf{g}'}$) if there exists a single $i \in \{1,\dots,n\}$ such that $g_i > g'_i$ and $g_j = g'_j$ for each $j \neq i$. □

Given a collection of regions generated by the state-space matrix using Algorithm 1, the procedure works as follows (cf. Algorithm 2). For each pair of adjacent regions, their intersection is assigned to the region with higher index. The maximum number of operations is $\mathscr{O}(n^{2n+1})$ [2, p. 3045], where n denotes dimension of the state space.

Algorithm 2 (Generation of a partition from regions of the PWA system)
Input: $A \in \mathbb{R}_\varepsilon^{n\times n}$, a row-finite max-plus matrix
Output: Π_{AD}, a partition of S

```
initialize $\Pi_{AD}$ with the regions of the PWA system (cf. Algorithm 1)
for all $R_{\mathbf{g}}, R_{\mathbf{g}'} \in \Pi_{AD}$ do
  if $R_{\mathbf{g}} > R_{\mathbf{g}'}$ then
    the intersection is removed from the region with lower index,
    i.e. $R_{\mathbf{g}'} := R_{\mathbf{g}'} \setminus R_{\mathbf{g}}$
  end if
end for
```
□

Example 10 Consider the MPL system as in (2). The regions of the PWA system associated with the MPL system are described in Example 2. The *AD* partition generated by Algorithm 2 has the following three blocks $R'_{(1,1)} = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 > 3\}$, $R'_{(2,1)} = \{\mathbf{x} \in \mathbb{R}^2 : 0 < x_1 - x_2 \leq 3\}$, and $R'_{(2,2)} = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \leq 0\}$. Notice that $R'_{(2,2)} = R_{(2,2)}$. The regions are shown in Fig. 5 (left). □

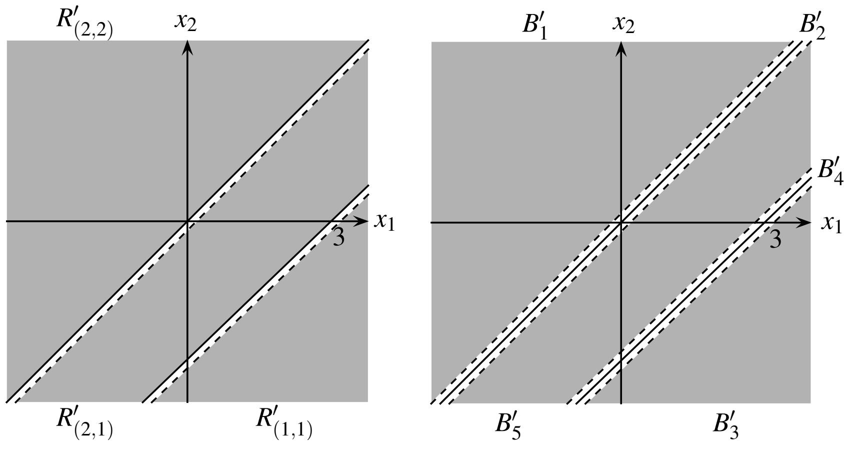

Fig. 5 The left and right plots are *AD* and Π_0 partitions of the MPL system in (2), respectively.

Implementation In VeriSiMPL 2, the approach to generate an *AD* partition is implemented in two JAVA classes *Maxpl2pwa* and *Maxpl2pwa_refine*. Recall that the JAVA class *Maxpl2pwa* generates a PWA system from an MPL system (cf. Section 2.2). Then the JAVA class *Maxpl2pwa_refine* refines the PWA regions to obtain an *AD* partition. This class has three public members *myA*, *myB* and *myD*. The members *myA* and *myB* correspond to the set of affine dynamics, whereas the member *myD* corresponds to the collection of regions. As it will be clear in Section 3.2, the set of affine dynamics is needed for the computation of transitions. The initialization of *Maxpl2pwa_refine* requires a state matrix *Ampl* and an instance of class *Maxpl2pwa*, which is the PWA system generated by the state matrix *Ampl*. $\square$

3.1.3 Π_0 Partition: Refinement of AP and AD Partitions

Let us determine the partition Π_0 of the state space S. As we discussed at the beginning of Section 3.1, the partition Π_0 is the refinement of Π_{AP} and Π_{AD}. The procedure consists of intersecting every block of Π_{AP} with every block of Π_{AD}. The maximum number of operations is $\mathscr{O}(n^2|\Pi_{AP}||\Pi_{AD}|)$, where n is the dimension of the state space, $|\Pi_{AP}|$ represents the number of blocks in the *AP* partition, and $|\Pi_{AD}|$ denotes the number of blocks in the *AD* partition.

Example 11 Let us determine the partition Π_0 of the MPL system (2) by refining Π_{AP} and Π_{AD}. Suppose that $AP = \{a\}$ and the set of states satisfying a is the following stripe $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$. The *AP* and *AD* partitions of the MPL system have been obtained in Examples 9 and 10, respectively. One can show that Π_0 partition has 5 blocks, i.e. $B'_1 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$, $B'_2 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$, $B'_3 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 > 3\}$, $B'_4 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$, $B'_5 = \{\mathbf{x} \in \mathbb{R}^2 : 0 < x_1 - x_2 < 3\}$. The Π_0 partition is shown in Fig. 5 (right).

Finally we define the set of abstract states $\hat{S}$, the abstraction function f, and the labeling function of the abstract transition system $L_f : \hat{S} \to 2^{AP}$. Since Π_0 contains 5

Table 3 Data structure used in JAVA class *Maxpl2ts_part* for computing the Π_0 partition of an MPL system.

	[0] ab	[1] $a\neg b$	[2] $\neg ab$	[3] $\neg a\neg b$
$R_{(1,1)}$ [0]	(DBM,...,DBM)	(DBM,...,DBM)	(DBM,...,DBM)	(DBM,...,DBM)
$R_{(2,1)}$ [1]	(DBM,...,DBM)	(DBM,...,DBM)	(DBM,...,DBM)	(DBM,...,DBM)
$R_{(2,2)}$ [2]	(DBM,...,DBM)	(DBM,...,DBM)	(DBM,...,DBM)	(DBM,...,DBM)

blocks, then $\hat{S} = \{\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4, \hat{s}_5\}$. The abstraction function f and the labeling function L_f are defined as follows

$$f(\mathbf{x}) = \begin{cases} \hat{s}_1, & \text{if } x_1 - x_2 < 0, \\ \hat{s}_2, & \text{if } x_1 - x_2 = 0, \\ \hat{s}_3, & \text{if } x_1 - x_2 > 3, \\ \hat{s}_4, & \text{if } x_1 - x_2 = 3, \\ \hat{s}_5, & \text{if } 0 < x_1 - x_2 < 3, \end{cases} \qquad L_f(\hat{s}_i) = \begin{cases} \{a\}, & \text{if } i \in \{2,5\}, \\ \emptyset, & \text{if } i \in \{1,3,4\}. \end{cases}$$

Let us consider $\mathscr{X}_0 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 1\}$ as the set of initial conditions. The set of initial abstract states is $I_f = \{\hat{s}_5\}$ [8, Def. 7.51]. $\square$

Implementation In VeriSiMPL 2, the procedure to compute Π_0 is implemented in the JAVA class *Maxpl2ts_part*. This class has 5 public members: *partADsize*, *partAPsize*, *finalpart*, *state_labels*, *AD_labels*. The member *state_labels* is used to store the set of atomic propositions satisfied by each block. The affine dynamics that are active in each region are stored in the member variable *AD_labels*. The members *partAPsize* and *partADsize* characterize the number of blocks in *AP* and *AD* partitions, respectively. The member *finalpart* stores the Π_0 partition as a list of DBM. In order to construct *finalpart*, we combine the *AP* and *AD* partitions as a 2-dimensional array, where each element of the array is a list of DBM (cf. Table 3). With this representation, we can keep track of the atomic propositions satisfied by each block and of the corresponding active affine dynamics. The set of atomic propositions satisfied within each block is determined by the binary encoding of the column index (cf. Section 3.1.1). The row index of a block characterizes the affine dynamics that are active in the block. Finally the initialization of *Maxpl2ts_part* requires the collection of regions in the *AP* partition and the collection of regions in the *AD* partition. $\square$

3.2 Transitions: One-Step Reachability

We develop an optimized technique to determine the transition relation of the abstract transition system, which relates pairs of blocks of the partition induced by the abstraction function. Recall that the set of (concrete) states associated with an abstract state $\hat{s}$ is obtained as the inverse image of $\hat{s}$ w.r.t. the abstraction function f, i.e. $f^{-1}(\hat{s}) = \{s : f(s) = \hat{s}\}$ – $f^{-1}(\hat{s})$ is a block (expressed as a DBM) and that the dynamics within it are affine.

The transition relation of the abstract transition system is defined as follows. If there exists a transition from an outgoing state s to an incoming state s' in the concrete transition system, i.e. $s \longrightarrow s'$, then there is a transition from $f(s)$ to $f(s')$ in the abstract transition system, i.e. $f(s) \longrightarrow_f f(s')$ [8, Def. 7.51]. The existence of such a transition can be determined by a forward- or backward-reachability approach. According to the former, we calculate $f^{-1}(\hat{s}') \cap Post(f^{-1}(\hat{s}))$, whereas if we use the backward approach we compute $f^{-1}(\hat{s}) \cap Pre(f^{-1}(\hat{s}'))$: The non-emptiness of the resulting set characterizes the presence of a transition from $\hat{s}$ to $\hat{s}'$.

Remark 1 The equivalent terms "image" and "direct successors" are used when the transitions in a dynamical system are represented via a function and via a transition relation, respectively. A similar argument holds for "inverse image" and "direct predecessors". □

In this work we focus on the forward-reachability approach, since it is computationally more attractive than the backward one, as argued in [2, p. 3046]. With focus on the forward-reachability approach, given an abstract state $\hat{s}$ we employ the affine dynamics that are active in $f^{-1}(\hat{s})$ to compute the direct successors as

$$Post(f^{-1}(\hat{s})) = \{A \otimes \mathbf{x} : \mathbf{x} \in f^{-1}(\hat{s})\}.$$

Since $f^{-1}(\hat{s})$ is a DBM and the dynamics within $f^{-1}(\hat{s})$ are affine, then $Post(f^{-1}(\hat{s}))$ is a DBM (cf. Proposition 1). The overall approach to determine the transition relation of the abstract transition system is shown in Algorithm 3. The maximum number of operations is $\mathcal{O}(n^3|\hat{S}|^2)$ [2, p. 3046] where n is the dimension of the state space and $|\hat{S}|$ represents the number of abstract states.

Algorithm 3 (Computations of the transitions of the abstract transition system via forward-reachability analysis)

```
Input: $\hat{S}$, a set of abstract states
    $f : S \to \hat{S}$, an abstraction function
Output: $\longrightarrow_f \subseteq \hat{S} \times \hat{S}$, a transition relation

initialize $\longrightarrow_f$ with the empty set
for all $\hat{s} \in \hat{S}$ do
  compute the direct successors of $\hat{s}$, i.e. $Post(f^{-1}(\hat{s}))$
  for all $\hat{s}' \in \hat{S}$ do
    if $f^{-1}(\hat{s}') \cap Post(f^{-1}(\hat{s}))$ is not empty then
      define a transition from $\hat{s}$ to $\hat{s}'$, i.e. $\hat{s} \longrightarrow_f \hat{s}'$
    end if
  end for
end for
```
□

Example 12 Consider the MPL system in (2) and the outcomes from Example 11. Using Algorithm 3, one can construct the transition relation in the abstract transition system. Skipping the details associated to the computations, the obtained abstract transition system is displayed in Fig. 6. □

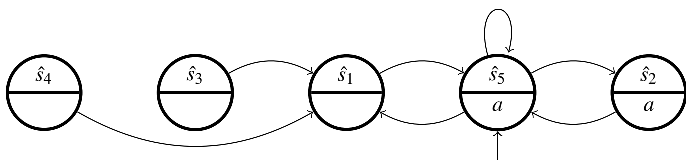

Fig. 6 The abstract transition system generated by the MPL system in (2). The initial state is $\hat{s}_5$ indicated by an incoming arrow from below. The states $\hat{s}_2$ and $\hat{s}_5$ satisfy the atomic proposition a, whereas the states $\hat{s}_1$, $\hat{s}_3$, and $\hat{s}_4$ are not associated with any specific atomic proposition.

Table 4 Data structure of the public member *trans* in JAVA class *Maxpl2ts_trans* for storing transitions.

$\hat{s}_1$ [0]	$\hat{s}_2$ [1]	$\hat{s}_3$ [2]	$\hat{s}_4$ [3]	$\hat{s}_5$ [4]
(4)	(4)	(0)	(0)	(0,1,4)

Implementation In VeriSiMPL 2, the procedure to determine the transitions of the abstract transition system is implemented in the JAVA class *Maxpl2ts_trans*. The initialization of *Maxpl2ts_trans* requires a PWA system (A, B, D) where D is the Π_0 partition of the state space of MPL system, and also requires the set of affine dynamics (A, B) that are active in each block of Π_0. The resulting transition relation is stored in two public members of *Maxpl2ts_trans*, called *adj* and *trans*. This class also has public members *refA* and *refB* to store the affine dynamics.

The public member *adj* is a 0-1 matrix. Its (i, j) element is 1 if there is a transition from j to i, otherwise the (i, j) element is equal to 0. This representation is used in the bisimulation-quotienting procedure (cf. Section 3.3). For example, the 0-1 matrix associated with the abstract transition system in Fig. 6 is given by

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

The public member *trans* stores transitions in an array of nonnegative integers. In this representation, each list contains indices of the direct successors of a state. This representation is used to generate a NuSMV file (cf. Section 3.5). For instance, the transitions of the abstract transition system in Fig. 6 are represented in Table 4. □

3.3 Bisimulation-Quotienting Procedure (Refinement Procedure)

In general the abstract transition system generated by the procedure in Sections 3.1 and 3.2 simulates the concrete transition system associated with the original MPL system [2, p. 3046]. It makes sense attempting to generate an abstract transition system that bisimulates the concrete transition system. The following Proposition 5 claims that the abstract transition system bisimulates the concrete transition system if and only if there is one outgoing transition from each abstract state. Notice that a transition system with one outgoing transition from each state is not necessarily deterministic (cf. Section 2.4): the transition system is deterministic if there is at most one initial state.

Proposition 5 ([2, Th. 4]) *Let TS be the concrete transition system generated by an MPL system and TS_f be the abstract transition system induced by an abstraction function $f : S \to \hat{S}$. The binary relation $\mathscr{R} = \{(s, f(s)) : s \in S\}$ is a bisimulation for (TS, TS_f) if and only if $|Post(\hat{s})| = 1$ for all $\hat{s} \in \hat{S}$.*

The procedure to generate an abstract transition system that bisimulates the concrete transition system works as follows. For each abstract state $\hat{s}$ with more than one outgoing transition, the corresponding block is refined according to the states $f^{-1}(\hat{s}')$, where $\hat{s}'$ is any of the direct successors of $\hat{s}$. The incoming and outgoing transitions are updated accordingly. These steps are repeated until all abstract states have one outgoing transition.

Let us focus on the refinement step of the procedure. Suppose that an abstract state $\hat{s}$ has more than one outgoing transition, i.e. $|Post(\hat{s})| > 1$. For each $\hat{s}' \in Post(\hat{s})$, we define a block consisting of the set of states such that the direct successor is in $f^{-1}(\hat{s}')$, i.e. $\{s \in f^{-1}(\hat{s}) : f(Post(s)) = \hat{s}'\} = f^{-1}(\hat{s}) \cap Pre(f^{-1}(\hat{s}'))$. Computationally we determine the inverse image of $f^{-1}(\hat{s}')$ w.r.t. the affine dynamics that are active in $f^{-1}(\hat{s})$, then we intersect the obtained inverse image with $f^{-1}(\hat{s})$. Notice that each block in the partition of $f^{-1}(\hat{s})$ is a DBM since $f^{-1}(\hat{s}')$ is a DBM, the inverse image of a DBM w.r.t. affine dynamics is a DBM, and the intersection of two DBM is a DBM (cf. Section 2.3).

Let us elaborate on the completeness of this procedure: unfortunately, in general such a procedure does not necessarily terminate in a finite number of steps. This in particular may happen in the presence of cycles in the abstract transition system containing abstract states with multiple outgoing transitions. An upper bound on the number of generated abstract states can be used as a stopping criterion. Sufficient conditions for the existence of a finite-state abstract transition system that bisimulates the concrete transition system have been discussed in [2, pp. 3046-3047]: these entail the completeness of the discussed refinement procedure.

Example 13 Let us apply the bisimulation-quotienting procedure to the abstract transition system in Fig. 6. Observe that from state $\hat{s}_5$ there are three outgoing transitions. The destinations of the outgoing transitions are $\hat{s}_1$, $\hat{s}_5$, and $\hat{s}_2$. Thus the set of states $\hat{s}_5$ is partitioned into three blocks: the blocks associated with outgoing transitions to $\hat{s}_1$, $\hat{s}_5$, and $\hat{s}_2$ are respectively $\{\mathbf{x} \in \mathbb{R}^2 : 2 < x_1 - x_2 < 3\}$, $\{\mathbf{x} \in \mathbb{R}^2 : 0 < x_1 - x_2 < 2\}$, and $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 2\}$. After the refinement step, the partition Π_1 is a set of 7 blocks.

Next we characterize the abstract transition system associated with the refined partition Π_1. Since Π_1 contains 7 blocks, the set of abstract states becomes $\hat{S}' = \{\hat{s}'_1, \ldots, \hat{s}'_7\}$. The abstraction function f' and the labeling function $L_{f'}$ are defined as

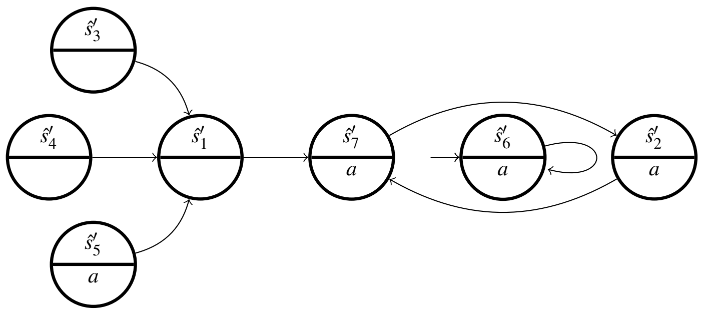

Fig. 7 The abstract transition system obtained after applying the bisimulation-quotienting procedure. The states $\hat{s}_2$, $\hat{s}_5$, $\hat{s}_6$ and $\hat{s}_7$ satisfy the atomic proposition a, whereas the states $\hat{s}_1$, $\hat{s}_3$ and $\hat{s}_4$ do not satisfy any atomic proposition.

follows:

$$f'(\mathbf{x}) = \begin{cases} \hat{s}'_1, & \text{if } x_1 - x_2 < 0, \\ \hat{s}'_2, & \text{if } x_1 - x_2 = 0, \\ \hat{s}'_3, & \text{if } x_1 - x_2 > 3, \\ \hat{s}'_4, & \text{if } x_1 - x_2 = 3, \\ \hat{s}'_5, & \text{if } 2 < x_1 - x_2 < 3, \\ \hat{s}'_6, & \text{if } 0 < x_1 - x_2 < 2, \\ \hat{s}'_7, & \text{if } x_1 - x_2 = 2, \end{cases} \qquad L_{f'}(\hat{s}'_i) = \begin{cases} \{a\}, & \text{if } i \in \{2,5,6,7\}, \\ \emptyset, & \text{if } i \in \{1,3,4\}. \end{cases}$$

Recall that the set of initial states is $\mathscr{X}_0 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 1\}$. Thus the set of initial abstract states is $I_{f'} = \{\hat{s}'_6\}$. The abstract transition system is depicted in Fig. 7 and it bisimulates the concrete transition system since all abstract states have one outgoing transition (cf. Proposition 5). □

Implementation In VeriSiMPL 2, the determinization procedure is implemented in the JAVA class *Maxpl2ts_trans_refine*. This class has three public members called *trans*, *D*, and *state_labels*. The public member *trans* is used to store the transition relation of the abstract transition system obtained after determinization. The public member *D* represents the partition of the state space obtained after determinization. The public member *state_labels* stores the atomic propositions satisfied by each block. The initialization of this class requires a transition relation that is represented as a 0-1 matrix, the PWA system (A,B,D) where D is a partition of the state space, the state labels for tracking the atomic propositions that are satisfied within each block, and an upper bound on the number of generated abstract states. □

3.4 Initial Abstract States

An abstract state is initial whenever the labeling function maps an initial concrete state to the abstract state [8, Def. 7.51]. We describe a procedure to determine the set of initial states in the abstract transition system. For each abstract state $\hat{s}$, check whether the concrete states represented by $\hat{s}$ are intersected with the initial concrete states, i.e. check whether $f^{-1}(\hat{s}) \cap \mathscr{X}_0$ is empty for all $\hat{s} \in \hat{S}$. If $f^{-1}(\hat{s}) \cap \mathscr{X}_0$ is empty, $\hat{s}$ is not an initial abstract state, otherwise $\hat{s}$ is an initial abstract state. In order to derive an expression for the maximum number of operations, we assume that $\mathscr{X}_0$ is a union of q_0 DBM. The maximum number of operations is $\mathscr{O}(n^3|\hat{S}|q_0)$ where n is the dimension of the state space and $|\hat{S}|$ denotes the number of abstract states.

Example 14 For some examples of obtaining the initial abstract states, we refer the reader to Examples 11 and 13. □

Implementation In VeriSiMPL 2, the procedure to determine the set of initial abstract states is implemented in the JAVA class *CallInitStates*. This class has one public member called *initStates* that stores the index of initial abstract states. The initialization of this class requires two parameters: the initial concrete states and the partition associated with the abstraction function. Both parameters are represented as a union of finitely many DBM. □

3.5 Interface with the NuSMV model checker

Using the abstraction procedure described in the preceding subsections, in general we can construct an abstract transition system that simulates the original MPL system. In order to verify an LTL specification or a universal/existential fragment of CTL properties over the abstract transition system in an automated way, we feed the abstract transition system and the property of interest to an existing model checking tool. Here we employ the symbolic model checker NuSMV, which embeds BDD structures and symbolic computations geared towards time and space efficiency.

NuSMV [13] is a model checker originated from the extension of the CMU SMV, a BDD-based model checker. It allows to check finite-state systems against specifications described in both LTL and CTL. Furthermore, it can also perform bounded model checking [11]. In order to run NuSMV, it needs to be configured so that NuSMV model checking files can be run through an interactive shell.

Example 15 Let us check whether the abstract transition system in Fig. 6 satisfies the LTL formula $\Box a$. One can visually check that the abstract transition system does not satisfy the given LTL formula. A run of the query on NuSMV outputs the path $\hat{s}_5\hat{s}_1\hat{s}_5\hat{s}_5\hat{s}_5\ldots$ as a counterexample, which is indeed associated to an output trace that invalidates the given LTL specification. Since the abstract transition system simulates the original MPL system (2), this does not imply that the MPL system does not satisfy the LTL formula $\Box a$.

In order to reach conclusive statements for the LTL formula $\Box a$ over the MPL system, we examine whether the abstract transition system in Fig. 7, a bisimulation of the concrete MPL system, satisfies the formula. One can see that the abstract transition system satisfies the LTL formula, as confirmed by a NuSMV call. We conclude that the original MPL system (2) satisfies the LTL formula $\Box a$. □

Implementation In VeriSiMPL 2, a JAVA class *Ts2nusmv* is implemented to translate and write the abstract transition system as a NuSMV file. This class has no public member. The initialization of the class requires a list of the transitions between states, the state labels, the initial abstract states, and the properties to be verified, expressed as LTL or CTL formulae. The initial abstract states are computed by the JAVA class *CalInitStates* (cf. Section 3.4). After writing the data into a NuSMV file, *Ts2nusmv* feeds this file into NuSMV and retrieves the verification results. Next, it displays the results directly in the JAVA runtime terminal. □

4 Tree-Based Finite Abstractions of Max-Plus-Linear Systems

As it will be clear in Section 5, the runtime bottleneck of the abstraction procedure discussed in Section 3 lies in the calculation of transitions. We mitigate this limitation by updating the main data structure (and related algorithms) from an array list to a tree. Sections 4.1 and 4.2 discuss the usage of the tree data structure for partitioning the state space and for computing the transitions, respectively. Since the determinization procedure does not benefit in general from the tree structure, it is left to be the same as in Section 3.3. Furthermore the procedure for generating a NuSMV file (cf. Section 3.5) can still be used when the main data structure is a tree.

4.1 States: Partitioning Procedure

We discuss a partitioning procedure that leverages a tree structure. The procedure constructs a tree representing a partition of the state space: the tree is called space-partitioning tree.

The partitioning procedure consists of two steps. In the first step, we generate a space-partitioning tree that represents the *AP* partition (cf. Section 4.1.1). In the second step, we refine the obtained *AP* partition w.r.t. the *AD* partition (cf. Section 4.1.2) by increasing the height of the space-partitioning tree. The resulting space-partitioning tree represents the Π_0 partition.

4.1.1 AP Partition

Let us describe a procedure to generate a space-partitioning tree that represents an *AP* partition. The procedure is a modification of the procedure described in Section 3.1.1: where in this section the *AP* partition is newly represented as a tree. The space-partitioning tree associated with the *AP* partition has two levels: in the first level, a root node represents the entire state space, i.e. $\mathbb{R}^n$. Each block of the *AP* partition is represented by a node in the second level, as a child of the root node.

Example 16 Let us revisit Example 9. Suppose that we have one atomic proposition a, i.e. $AP = \{a\}$, where the set of states satisfying a is $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$. The *AP* partition has three blocks $B_1 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$, $B_2 = \{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$, and $B_3 = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \geq 3\}$. The space-partitioning tree representing the obtained *AP* partition is depicted in Fig. 8, where the nodes R^a, $R^{\neg a1}$, and $R^{\neg a2}$ represent B_2, B_1, and B_3, respectively. □

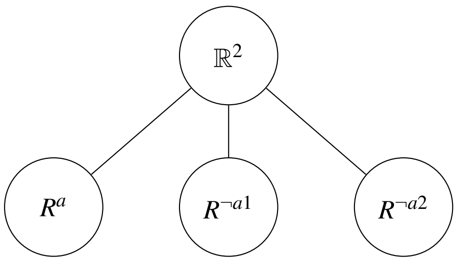

Fig. 8 The space-partitioning tree representing the *AP* partition obtained in Example 16.

Implementation In VeriSiMPL 2, the procedure to generate a space-partitioning tree representing an *AP* partition is implemented in the JAVA class *PartAgainstAP_tree*. This class has one public member *tree* corresponding to the space-partitioning tree representing the *AP* partition. The initialization of the class requires a list of DBM, where each DBM corresponds to the set of states satisfying an atomic proposition. The implementation is similar with *PartAgainstAP* (cf. Section 3.1.1). In *PartAgainstAP* the *AP* partition is represented as an array list, whereas in *PartAgainstAP_tree* the *AP* partition is represented as a two-level tree. □

4.1.2 Π_0 Partition: Refinement of AP and AD Partitions

In this section we describe a procedure to refine the *AP* partition obtained in Section 4.1.1 w.r.t. an *AD* partition by increasing the height of the space-partitioning tree. The procedure consists of two steps. In the first step, under each of the nodes in the second level of the space-partitioning tree, we build a tree that corresponds to the regions of the PWA system generated by an MPL system. Recall that the collection of PWA regions is a cover of the state space and in general not a partition. Thus in a second step, by leveraging the tree structure, we refine the PWA regions in order to obtain a partition of the state space.

Let us focus on building a tree that corresponds to the PWA regions under each node in the second level of the space-partitioning tree. We modify the backtracking procedure to determine the PWA regions (cf. Section 2.2). In Section 2.2, every node in the potential search tree (cf. Fig. 1) is associated with the PWA region (5). In this section, every node in the potential search tree (cf. Fig. 1) is associated with the intersection of PWA region (5) and the set of states represented by the considered node in the second level of the space-partitioning tree. Every time we find a node other than the root node in the potential search tree that represents a nonempty set of states, we create a node in the space-partitioning tree representing the same states.

Example 17 We consider the MPL system (2) and a single atomic proposition a, i.e. $AP = \{a\}$. The set of states satisfying a is $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$. The space-partitioning tree associated with the *AP* partition is shown in Fig. 8. Let us construct a tree under every node in the second level of the space-partitioning tree. First let us consider the node R^a. The set of states represented by this node is $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$ (cf. Example 16). Next we traverse the potential search tree (cf. Fig. 1) in a depth-first manner:

1. Root node in the potential search tree represents $\mathbb{R}^n \cap \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\} = \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$. Although this node represents a nonempty set of states, we do not create a node in the space-partitioning tree because this is a root node in the potential search tree. We continue to the leftmost child of the root node in the potential search tree.
2. Node $R_{(1)}$ in the potential search tree represents $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \ge 3\} \cap \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\} = \emptyset$. Since $R_{(1)}$ represents the empty set, according to the backtracking procedure (cf. Section 2.2), there is no need to consider $R_{(1,1)}$ and $R_{(1,2)}$. We continue to the next child of the root node in the potential search tree.
3. Node $R_{(2)}$ in the potential search tree represents $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \le 3\} \cap \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\} = \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$. Since $R_{(2)}$ represents a nonempty set of states, we create a node $R^a_{(2)}$ in the space-partitioning tree (cf. Fig. 9). Node $R^a_{(2)}$ is a child of R^a. Node $R^a_{(2)}$ represents $\{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$. Then we continue to the leftmost child of the node $R_{(2)}$ in the potential search tree.
4. Node $R_{(2,1)}$ in the potential search tree represents $\{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 \le 3\} \cap \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\} = \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$. Since $R_{(2,1)}$ represents a nonempty set of states, we create a node $R^a_{(2,1)}$ in the space-partitioning tree (cf. Fig. 9). Node $R^a_{(2,1)}$ is a child of $R^a_{(2)}$. Node $R^a_{(2,1)}$ represents $\{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$. Then we continue to the next child of the node $R_{(2)}$ in the potential search tree.
5. Node $R_{(2,2)}$ in the potential search tree represents $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \le 0\} \cap \{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\} = \{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$. Since $R_{(2,2)}$ represents a nonempty set of states, we create a node $R^a_{(2,2)}$ in the space-partitioning tree (cf. Fig. 9). Node $R^a_{(2,2)}$ is a child of $R^a_{(2)}$. Node $R^a_{(2,2)}$ represents $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$.

A similar procedure can be applied to other nodes in the second level of the space-partitioning tree. Skipping the computational details, after applying the procedure to all nodes in the second level of the space-partitioning tree, we obtain the tree shown in Fig. 9. The set of states represented by $R^{\neg a1}_{(2)}$, $R^{\neg a1}_{(2,2)}$, $R^{\neg a2}_{(1)}$, $R^{\neg a2}_{(1,1)}$, $R^{\neg a2}_{(2)}$, and $R^{\neg a2}_{(2,1)}$ is $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$, $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$, $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \ge 3\}$, $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \ge 3\}$, $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$, and $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$, respectively. Observe that the tree in Fig. 9 is not a space-partitioning tree because the sets represented by $R^{\neg a2}_{(1)}$ and $R^{\neg a2}_{(2)}$ overlap. $\square$

The next step is to refine the PWA regions by using the generated tree structure. The refinement procedure traverses the tree obtained in the first step in a breadth-first manner. For each internal node, we check whether the states represented by the children of the internal node form a partition of the states represented by the internal node. If the states represented by the children are overlapping, we remove the overlapped region from the states represented by the nodes with lower index and their descendant (cf. Algorithm 2).

Example 18 Let us refine the PWA regions associated with the tree constructed in Example 17 (cf. Fig. 9). The refinement procedure traverses the internal nodes of the tree in a breadth-first way:

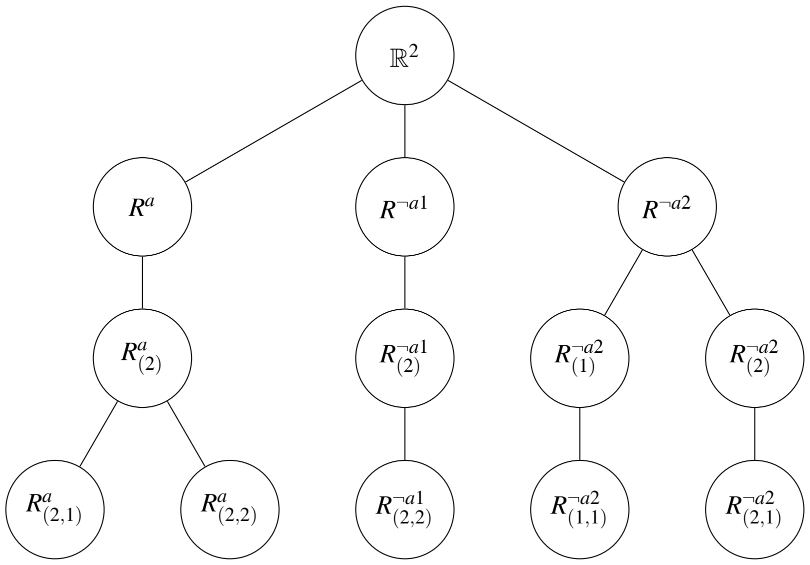

Fig. 9 The tree obtained in Examples 17 and 18. The tree obtained in Example 17 is not a space-partitioning tree, whereas the tree obtained in Example 18 is a space-partitioning tree.

1. Root node: since the states represented by R^a, $R^{\neg a1}$, and $R^{\neg a2}$ are a partition of $\mathbb{R}^2$, we do not do anything.
2. Nodes R^a and $R^{\neg a1}$: since these nodes have one child, the states represented by their child cannot overlap. We continue to the next node.
3. Node $R^{\neg a2}$: the states represented by $R^{\neg a2}$, $R^{\neg a2}_{(1)}$, and $R^{\neg a2}_{(2)}$ are $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \geq 3\}$, $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \geq 3\}$, and $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$, respectively. Notice that the states represented by $R^{\neg a2}_{(1)}$ overlap with the states represented by $R^{\neg a2}_{(2)}$. We remove the overlapping region $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$ from states represented by $R^{\neg a2}_{(1)}$ and $R^{\neg a2}_{(1,1)}$. The states represented by both $R^{\neg a2}_{(1)}$ and $R^{\neg a2}_{(1,1)}$ become $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 > 3\}$.
4. Node $R^a_{(2)}$: the states represented by $R^a_{(2)}$, $R^a_{(2,1)}$, and $R^a_{(2,2)}$ are $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$, $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$, and $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$, respectively. Notice that the states represented by $R^a_{(2,1)}$ overlap with the states represented by $R^a_{(2,2)}$. We remove the overlapping region $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$ from states represented by $R^a_{(2,1)}$. The states represented by $R^a_{(2,1)}$ become $\{\mathbf{x} \in \mathbb{R}^2 : 0 < x_1 - x_2 < 3\}$.
5. Nodes $R^{\neg a1}_{(2)}$, $R^{\neg a2}_{(1)}$, and $R^{\neg a2}_{(2)}$: these nodes have one child, thus their children cannot overlap.

The states represented by each node in the space-partitioning tree (cf. Fig. 9) after the refinement procedure are summarized in Table 5. □

Implementation In VeriSiMPL 2, the procedure to determine the Π_0 partition by using a tree structure is implemented in the JAVA class *Maxpl2ts_part_tree*. This class

Table 5 The set of states represented by each node in the space-partitioning tree (cf. Fig. 9) after the refinement (first and second columns). The third column represents the abstract state associated with the concrete states represented by leaf nodes, according to Example 11.

Root node	$\mathbb{R}^2$	
Node R^a	$\{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$	
Node $R^{\neg a1}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$	
Node $R^{\neg a2}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \ge 3\}$	
Node $R^a_{(2)}$	$\{\mathbf{x} \in \mathbb{R}^2 : 0 \le x_1 - x_2 < 3\}$	
Node $R^{\neg a1}_{(2)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$	
Node $R^{\neg a2}_{(1)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 > 3\}$	
Node $R^{\neg a2}_{(2)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$	
Node $R^a_{(2,1)}$	$\{\mathbf{x} \in \mathbb{R}^2 : 0 < x_1 - x_2 < 3\}$	$\hat{s}_5$
Node $R^a_{(2,2)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$	$\hat{s}_2$
Node $R^{\neg a1}_{(2,2)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$	$\hat{s}_1$
Node $R^{\neg a2}_{(1,1)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 > 3\}$	$\hat{s}_3$
Node $R^{\neg a2}_{(2,1)}$	$\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 3\}$	$\hat{s}_4$

has 7 public members A, B, *ds*, *flag*, *Nr*, *total*, *tree*. The space-partitioning tree is stored in the public member *tree*. The other public members are needed to construct the space-partitioning tree. This class requires a state matrix and a space-partitioning tree associated with the *AP* partition (cf. Section 4.1.1) for initialization. The procedure to construct a tree that corresponds to PWA regions under each node in the second level of the space-partitioning tree is implemented as two recursive functions *loop* and *recursive* using the depth-first approach. The tree-based refinement procedure is implemented as a recursive function *refineTree* using the breadth-first approach. □

4.2 Transitions: One-Step Reachability

The advantage of using a tree structure, compared to a list, is the significant improvement on the calculation time to generate the transitions in the abstract model. Using a list as the data structure, to compute the transitions the procedure considers all the pairs of blocks (cf. Algorithm 3), whereas applying a backtracking technique to the space-partitioning tree, the procedure to compute the transitions skips many pairs of blocks.

Let us discuss the procedure. For every leaf node of the space-partitioning tree, the following steps are taken. First, we compute the image of the states represented by the leaf node. Then the backtracking algorithm traverses the space-partitioning tree recursively, starting from the root, in a depth-first order. At each node, the algorithm checks whether the states represented by the node intersect with the image. If the intersection is empty, the whole sub-tree rooted at the node is skipped (pruned).

Example 19 Consider the partition Π_0 obtained in Example 18 and the abstraction function in Table 5. Let us determine the outgoing transitions from the abstract state $\hat{s}_1$. The abstract state $\hat{s}_1$ represents concrete states $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$. The affine dynamics are $x_1(k) = x_2(k-1)+5$, $x_2(k) = x_2(k-1)+3$. Using the procedure discussed in Section 2.3.5, the image of the concrete states w.r.t. its affine dynamics is $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 2\}$. Next we visit the space-partitioning tree in Fig. 9 by using the depth-first order:

1. Root node: since the image has non-empty intersection with $\mathbb{R}^2$, we continue to the leftmost child of the root node.
2. Node R^a: the image has non-empty intersection with the states represented by R^a. The states represented by R^a are $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$. Then we continue to the child of node R^a.
3. Node $R^a_{(2)}$: the image has non-empty intersection with the states represented by $R^a_{(2)}$. The states represented by $R^a_{(2)}$ are $\{\mathbf{x} \in \mathbb{R}^2 : 0 \leq x_1 - x_2 < 3\}$. Then we continue to the leftmost child of node $R^a_{(2)}$.
4. Node $R^a_{(2,1)}$: the image has non-empty intersection with the states represented by $R^a_{(2,1)}$. The states represented by $R^a_{(2,1)}$ are $\{\mathbf{x} \in \mathbb{R}^2 : 0 < x_1 - x_2 < 3\}$. Thus there is a transition from $\hat{s}_1$ to $\hat{s}_5$ (cf. Table 5). Then we continue to the next child of $R^a_{(2)}$.
5. Node $R^a_{(2,2)}$: the image has empty intersection with the states represented by $R^a_{(2,2)}$. The states represented by $R^a_{(2,2)}$ are $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 = 0\}$. Since $R^a_{(2,2)}$ is a leaf node, nothing is pruned. According to the depth-first order, we continue with the middle child of the root node.
6. Node $R^{\neg a1}$: the image has empty intersection with the states represented by $R^{\neg a1}$. The states represented by $R^{\neg a1}$ are $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 < 0\}$. Thus we prune the whole sub-tree rooted at $R^{\neg a1}$, i.e. we skip $R^{\neg a1}_{(2)}$ and $R^{\neg a1}_{(2,2)}$. Then we continue to the rightmost child of the root node.
7. Node $R^{\neg a2}$: the image has empty intersection with the states represented by $R^{\neg a2}$. The states represented by $R^{\neg a2}$ are $\{\mathbf{x} \in \mathbb{R}^2 : x_1 - x_2 \geq 3\}$. Thus we prune the whole sub-tree rooted at $R^{\neg a2}$, i.e. we skip $R^{\neg a2}_{(1)}$, $R^{\neg a2}_{(1,1)}$, $R^{\neg a2}_{(2)}$, and $R^{\neg a2}_{(2,1)}$.

In summary, there is one outgoing transition from $\hat{s}_1$, which ends up in $\hat{s}_5$. □

Implementation In VeriSiMPL 2, the JAVA class *Maxpl2ts_trans_tree* is used to calculate the transition relation between abstract states. Before computing transitions, *Maxpl2ts_trans_tree* uses a depth-first function *addNumber* to add a unique number *no* to each leaf node for identification. The function *addNumber* also generates *dslist* that stores a list of the leaf nodes. Notice that *dslist* also represents the Π_0 partition. Next, the procedure to compute transitions is implemented as a recursive function *DepthFirstTree* which is based on depth-first search. The function *DepthFirstTree* takes one leaf node and the space-partitioning tree as parameters, and generates variable *list* that represents leaf nodes that can be reached from the given leaf node in one step. The variable *list* is a list containing the identity number of the leaf nodes. Thus if the space-partitioning tree has *m* leaf nodes, *DepthFirstTree* will be run for *m* times.

In order to determinize the abstract transition system, the generated partition *dslist* and transition relation *list* can be fed to the JAVA class *Maxpl2ts_trans_refine* (cf. Section 3.3). □

Table 6 Numerical benchmark for the list-based implementation in VeriSiMPL 2, versus the earlier VeriSiMPL software, over the abstraction procedure on MPL systems. Each entry represents mean and maximal values over 10 independently generated experiments.

	VeriSiMPL		list implementation in VeriSiMPL 2	
size of MPL system	time for generation of states	time for generation of transitions	time for generation of states	time for generation of transitions
3	{0.12;0.13}[sec]	{0.05;0.06}[sec]	{0.01;0.05}[sec]	{0.00;0.01}[sec]
4	{0.24;0.26}[sec]	{0.10;0.12}[sec]	{0.01;0.06}[sec]	{0.01;0.02}[sec]
5	{0.51;0.59}[sec]	{0.23;0.29}[sec]	{0.01;0.09}[sec]	{0.01;0.04}[sec]
6	{1.12;1.42}[sec]	{0.58;0.85}[sec]	{0.02;0.10}[sec]	{0.06;0.08}[sec]
7	{2.30;3.07}[sec]	{1.55;2.61}[sec]	{0.04;0.14}[sec]	{0.30;0.48}[sec]
8	{6.07;6.86}[sec]	{8.69;11.93}[sec]	{0.15;0.22}[sec]	{1.76;2.12}[sec]
9	{12.35;16.07}[sec]	{26.39;44.75}[sec]	{0.57;0.86}[sec]	{8.86;13.37}[sec]
10	{42.49;61.30}[sec]	{3.39;5.81}[min]	{1.48;1.98}[sec]	{27.75;40.87}[sec]
11	{1.64;3.48}[min]	{9.89; 28.74}[min]	{7.83;12.95}[sec]	{3.11;5.29}[min]
12	—	—	{28.24;48.07}[sec]	{13.55;24.55}[min]
13	—	—	{2.26;3.32}[min]	{1.26;2.82}[hr]

5 Computational Benchmark

In order to showcase how the new implementation in VeriSiMPL 2 is more efficient than the earlier one in VeriSiMPL, we compare the performance of both software in a fair manner. We start comparing the list implementation in both VeriSiMPL 2 with VeriSiMPL (cf. Section 5.1). Then we compare the list and tree implementations in VeriSiMPL 2 (cf. Section 5.2).

In this benchmark, we compute the runtime required to obtain the abstraction of an MPL system as a finite-state transition system, for increasing dimensions n of the given MPL system (which is the number of the continuous variables). For each given dimension n, we independently generate 10 row-finite matrices A, with 2 finite elements placed at random in each row. The finite elements are integers generated at random, and taking values between 1 and 100. In each dimension, we always use the same 10 random matrices for all the implementations, namely VeriSiMPL, the list implementation in VeriSiMPL 2, and the tree implementation in VeriSiMPL 2. The experiments are run on an Intel® Core$^{\text{TM}}$ i5-3427U 1.80 GHz laptop with 4.00 GB (2.17 GB available) of memory.

5.1 Comparison of VeriSiMPL with the List Implementation in VeriSiMPL 2

For dimension $n \in \{3,\ldots,13\}$, Table 6 reports the (mean and maximal values for the) time needed to construct the abstract transition system within both VeriSiMPL and VeriSiMPL 2. The abstraction procedure is divided into two successive steps: generation of the abstract states and of the transitions.

As we can see from Table 6, for the generation time of states, VeriSiMPL 2 is about 2 to 57 times faster than VeriSiMPL, depending on the dimension. As for the generation time of transitions (representing the bottleneck of the overall procedure),

Table 7 Numerical benchmark for list implementation against tree implementation in VeriSiMPL 2 for abstraction procedure of MPL systems. Each entry represents mean and maximal values over 10 independent experiments.

	list implementation		tree implementation	
size of MPL system	time for generation of states	time for generation of transitions	time for generation of states	time for generation of transitions
3	{0.01;0.05}[sec]	{0.00;0.01}[sec]	{0.01;0.06}[sec]	{0.01;0.02}[sec]
4	{0.01;0.06}[sec]	{0.01;0.02}[sec]	{0.01;0.08}[sec]	{0.01;0.02}[sec]
5	{0.01;0.09}[sec]	{0.01;0.04}[sec]	{0.04;0.15}[sec]	{0.04;0.13}[sec]
6	{0.02;0.10}[sec]	{0.06;0.08}[sec]	{0.06;0.11}[sec]	{0.06;0.12}[sec]
7	{0.04;0.14}[sec]	{0.30;0.48}[sec]	{0.16;0.40}[sec]	{0.09;0.15}[sec]
8	{0.15;0.22}[sec]	{1.76;2.12}[sec]	{0.51;0.67}[sec]	{0.22;0.35}[sec]
9	{0.57;0.86}[sec]	{8.86;13.37}[sec]	{1.29;1.82}[sec]	{0.98;1.06}[sec]

VeriSiMPL 2 is about 3 to 23 times faster than VeriSiMPL, depending on the dimension.

From dimension 12, we stop experimenting on VeriSiMPL because for dimension 11, the abstraction procedure already takes several minutes to calculate states and about half an hour to calculate transitions. Also, the time difference will become even more conspicuous as dimensionality increases. On the other hand, with an 11-dimensional state matrix, VeriSiMPL 2 only takes an average of 7.83 seconds to generate the states and an average of 3.11 minutes to generate the transitions. Even with a 13-dimensional state matrix, VeriSiMPL 2 only takes an average of 2.26 minutes to generate the states and an average of 1.26 hours to generate the transitions.

Through the comparison with VeriSiMPL, VeriSiMPL 2 shows its remarkable improvement in speeding up the abstraction procedure of MPL systems. We can easily notice that the generation time of transitions is still the bottleneck of the whole procedure. For a 13-dimensional state matrix, while the generation of states only takes an average of 2.26 minutes, the generation of transitions takes an average of 1.26 hours, with a maximal value of 2.82 hours. In order to further speed up the calculation of transitions, we have implemented the tree structure within the abstraction procedure. The comparison between the list and tree implementations in VeriSiMPL 2 will be discussed in the next section.

5.2 Comparison of List and Tree Implementations in VeriSiMPL 2

For dimension $n \in \{3,\dots,9\}$, Table 7 reports the (mean and maximal values for the) time needed to construct the abstract transition system using list and tree implementations in VeriSiMPL 2. The abstraction procedure comprises of two successive steps: the generation of states and of transitions.

From Table 7, for the generation of states, the tree implementation is about 1 to 4 times slower than the list implementation. The reason for this outcome is that we need to build a space-partitioning tree instead of a one-dimensional array list for storing the partition sets of the state space. However, since the generation of states is not the bottleneck of the abstraction procedure, this outcome is not a decisive factor for measuring the overall efficiency.

For the generation of transitions, when the dimension is less than or equal to 6, the tree implementation is slightly slower than the list implementation. On the other hand, when the dimension is greater than or equal to 7, the tree implementation is about 3 to 13 times faster than the list implementation. For example in 9-dimensional state matrices, the list implementation takes an average of 8.86 seconds to generate transitions, whereas the tree implementation only takes an average of 0.98 seconds. Furthermore the increase of runtime in the tree implementation is not noticeable when the dimension increases. For instance the list implementation jumps from 0.08 seconds (6-dimensional) to 0.48 seconds (7-dimensional), whereas the tree implementation only increases from 0.12 seconds to 0.15 seconds.

6 Conclusions and Future Work

This work has described a technique to generate abstractions of Max-Plus-Linear (MPL) systems, characterized as finite-state transition systems. The procedure is based on partitioning the state space and on the one-step dynamics to relate partitioning regions. The resulting finite-state transition system has been shown to either simulate or bisimulate the original MPL system.

The list-based and tree-based abstraction procedure have been implemented in VeriSiMPL 2 and its performance has been tested on a numerical benchmark. By using a tree structure, the generation time of transitions reduces significantly compared with the implementation using a list. However the tree-based procedure requires a higher amount of memory because many operations (over the tree) are implemented as recursive functions. Along this line, the authors are interested in rewriting the relevant recursive functions as non-recursive ones. In order to further improve performance of the abstraction procedure, the authors are also interested in 1) generating an *AD* partition without refinement; 2) constructing a space-partitioning tree associated with an *AP* partition such that each level of the tree corresponds to an atomic proposition; 3) directly incorporating binary decision diagrams in the abstraction procedure; and 4) leveraging other techniques such as predicate abstractions and satisfiability modulo theories.

References

1. Adzkiya, D., Abate, A.: VeriSiMPL: Verification via biSimulations of MPL models. In: K. Joshi, M. Siegle, M. Stoelinga, P. D'Argenio (eds.) Proc. 10th Int. Conf. Quantitative Evaluation of Systems (QEST'13), *Lecture Notes in Computer Science*, vol. 8054, pp. 253–256. Springer, Heidelberg (2013). URL http://sourceforge.net/projects/verisimpl/
2. Adzkiya, D., De Schutter, B., Abate, A.: Finite abstractions of max-plus-linear systems. IEEE Trans. Autom. Control **58**(12), 3039–3053 (2013)
3. Adzkiya, D., De Schutter, B., Abate, A.: Backward reachability of autonomous max-plus-linear systems. In: Proc. 12th Int. Workshop Discrete Event Systems, pp. 117–122. Cachan (2014)
4. Adzkiya, D., De Schutter, B., Abate, A.: Forward reachability computation for autonomous max-plus-linear systems. In: E. Ábrahám, K. Havelund (eds.) Tools and Algorithms for the Construction

and Analysis of Systems (TACAS'14), *Lecture Notes in Computer Science*, vol. 8413, pp. 248–262. Springer, Heidelberg (2014)

5. Adzkiya, D., De Schutter, B., Abate, A.: Computational techniques for reachability analysis of max-plus-linear systems. Automatica **53**(0), 293–302 (2015)
6. Alur, R., Henzinger, T., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. Proc. IEEE **88**(7), 971–984 (2000)
7. Baccelli, F., Cohen, G., Olsder, G., Quadrat, J.P.: Synchronization and Linearity, An Algebra for Discrete Event Systems. John Wiley and Sons (1992)
8. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
9. Bellman, R.: On a routing problem. Quart. Appl. Math. **16**, 87–90 (1958)
10. Bemporad, A., Ferrari-Trecate, G., Morari, M.: Observability and controllability of piecewise affine and hybrid systems. IEEE Trans. Autom. Control **45**(10), 1864–1876 (2000)
11. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: W. Cleaveland (ed.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), *Lecture Notes in Computer Science*, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
12. Brackley, C.A., Broomhead, D.S., Romano, M.C., Thiel, M.: A max-plus model of ribosome dynamics during mRNA translation. Journal of Theoretical Biology **303**(0), 128–140 (2012)
13. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: E. Brinksma, K. Larsen (eds.) Computer Aided Verification (CAV'02), *Lecture Notes in Computer Science*, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
14. Cohen, G., Gaubert, S., Quadrat, J.P.: Max-plus algebra and system theory: Where we are and where to go now. Annual Reviews in Control **23**(0), 207–219 (1999)
15. Cuninghame-Green, R.: Minimax Algebra, *Lecture Notes in Economics and Mathematical Systems*, vol. 166. Springer-Verlag, Berlin, Germany (1979)
16. De Schutter, B.: On the ultimate behavior of the sequence of consecutive powers of a matrix in the max-plus algebra. Linear Algebra and its Applications **307**(1-3), 103–117 (2000)
17. De Schutter, B., van den Boom, T.: Model predictive control for max-plus-linear discrete event systems. Automatica **37**(7), 1049–1056 (2001)
18. Di Loreto, M., Gaubert, S., Katz, R., Loiseau, J.: Duality between invariant spaces for max-plus linear discrete event systems. SIAM Journal on Control and Optimization **48**(8), 5606–5628 (2010)
19. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: J. Sifakis (ed.) Automatic Verification Methods for Finite State Systems, *Lecture Notes in Computer Science*, vol. 407, chap. 17, pp. 197–212. Springer, Heidelberg (1990)
20. van Eekelen, J., Lefeber, E., Rooda, J.: Coupling event domain and time domain models of manufacturing systems. In: Proc. 45th IEEE Conf. Decision and Control (CDC'06), pp. 6068–6073 (2006)
21. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM **5**(6), 345 (1962)
22. Heemels, W., De Schutter, B., Bemporad, A.: Equivalence of hybrid dynamical models. Automatica **37**(7), 1085–1091 (2001)
23. Heidergott, B., Olsder, G., van der Woude, J.: Max Plus at Work–Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications. Princeton University Press (2006)
24. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
25. Katz, R.D.: Max-plus (A,B)-invariant spaces and control of timed discrete-event systems. IEEE Trans. Autom. Control **52**(2), 229–241 (2007)
26. Lafferriere, G., Pappas, G., Sastry, S.: O-minimal hybrid systems. Mathematics of Control, Signals and Systems **13**(1), 1–21 (2000)
27. Leenaerts, D., van Bokhoven, W.: Piecewise Linear Modeling and Analysis. Kluwer Academic Publishers, Boston (1998)
28. Maia, C., Andrade, C., Hardouin, L.: On the control of max-plus linear system subject to state restriction. Automatica **47**(5), 988–992 (2011)
29. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs, NJ (1989)
30. Pappas, G.J.: Bisimilar linear systems. Automatica **39**(12), 2035–2047 (2003)
31. Roset, B., Nijmeijer, H., van Eekelen, J., Lefeber, E., Rooda, J.: Event driven manufacturing systems as time domain control systems. In: Proc. 44th IEEE Conf. Decision and Control and European Control Conf. (CDC-ECC'05), pp. 446–451 (2005)
32. Shaffer, C.: Data Structures and Algorithm Analysis in Java, Third Edition. Dover Publications (2011)
33. Sontag, E.D.: Nonlinear regulation: The piecewise-linear approach. IEEE Trans. Autom. Control **26**(2), 346–358 (1981)

34. Wegener, I.: Branching Programs and Binary Decision Diagrams - Theory and Applications. SIAM Monographs on Discrete Mathematics and Applications (2000)
35. Yordanov, B., Belta, C.: Formal analysis of discrete-time piecewise affine systems. IEEE Trans. Autom. Control **55**(12), 2834–2840 (2010)
36. Yordanov, B., Tůmová, J., Černá, I., Barnat, J., Belta, C.: Formal analysis of piecewise affine systems through formula-guided refinement. Automatica **49**(1), 261–266 (2013)