Object Oriented Design of Software Tool for Finite Abstractions of Max-Plus-Linear Systems using Unified Modeling Language

Muhammadun, Dieky Adzkiya and Imam Mukhlash

***Abstract*—Max-Plus-Linear (MPL) systems are a class of discrete-event systems with a continuous state space characterizing the timing of the underlying sequential discrete events. There is a formal approach to analyze these systems based on finite abstractions. The abstraction algorithms have been in MATLAB using list data structure and in JAVA using tree data structure. The MATLAB implementation requires long computational time, whereas the JAVA one requires larger memory allocation. In this work, we discuss an object oriented design in C++ using tree data structure without recursive functions in the hope of improving the results obtained by the two previous implementations.**

***Index Terms*—Finite abstractions, max-plus-linear systems, object oriented design, unified modeling language.**

I. Introduction

MAX-PLUS-LINEAR (MPL) systems are a class of discrete-event dynamic systems [1], [2] with a continuous state space characterizing the time of occurrence of the underlying sequential discrete events. MPL systems can be used to describe the timing synchronization between interleaved processes, under the assumption that timing events are linearly dependent (within the max-plus algebra) on previous event occurrences. Such systems are employed in the analysis and scheduling of infrastructure networks, such as communication and railway systems [2], production and manufacturing lines [3], [4], or biological systems [5]. They cannot model concurrency and are related to a subclass of Timed Petri Nets, namely Timed-Event Graphs [1].

Classical dynamical analysis of MPL systems leverages their algebraic [6] or geometric features [7]. It allows investigating model properties such as its transient behavior, its periodic regimes, or its ultimate dynamical behavior [8]. Recently, there is a formal approach that has explored a new, alternative approach to analysis that is based on finite-state abstractions [9] of autonomous and nonautonomous MPL systems. The proposed abstraction procedures generates a finite-state Transition System (TS) in a finite number of steps. There is a formal relationship between the concrete model and its abstraction. More precisely, [9] argue that in general the LTS abstraction simulates the original MPL model, and furthermore they provide sufficient conditions to establish a bisimulation relation between abstract and concrete models [10].

The abstraction algorithms over MPL systems have been implemented in MATLAB using list data structure [11]. In this implementation, the process to compute transition requires a long computation time. Then the implementation has been improved by using tree data structure in JAVA language [12]. This implementation successfully accelerates the computational time but requires a larger memory allocation because its functions are recursive. In this work, we will discuss the object oriented design in C++ using tree data structure without recursive functions in the hope of improving the results obtained by the previous two implementations.

Manuscript received January 27, 2017; accepted February 28, 2017.

The authors are with the Department of Mathematics, Institut Teknologi Sepuluh Nopember, Surabaya 60111, Indonesia. Email: muhammadun14@mhs.matematika.its.ac.id, {dieky,imamm}@matematika.its.ac.id

II. Models and Preliminaries

A. Max-Plus-Linear Systems

An (autonomous) Max-Plus-Linear (MPL) system [1, Rem. 2.75] is defined as:

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

where $A \in \mathbb{R}_{\varepsilon}^{n\times n}$, $\mathbf{x}(k-1) = [x_1(k-1) \dots x_n(k-1)]^T \in \mathbb{R}^n$ for $k \in \mathbb{N}$. The detailed discussion on MPL systems can be seen in [1], [2]. The meaning of k does not represent the "time", as in the usual discrete-time systems. MPL systems are a discrete-event system. More precisely, the parameter k represents an event counter. The state $\mathbf{x}(k)$ represents the time of k-th occurrence of state events.

B. Piecewise Affine Systems

An autonomous MPL system characterized by row-finite state matrix $A \in \mathbb{R}_{\varepsilon}^{n\times n}$ can be expressed as a PWA system in the event domain [13, Sec. 3]. The regions and the corresponding affine dynamics can be constructed from coefficients $g = (g_1, \dots, g_n) \in \{1, \dots, n\}^n$ [9], [12]. For each $i \in \{1, \dots, n\}$, the coefficient g_i represents the maximal term in the i-th state equation $x_i(k) = \max\{A(i,1) + x_1, \dots, A(i,n) + x_n\}$, that is $A(i,j) + x_j \le A(i,g_i) + x_{g_i}$ for all $j \in \{1, \dots, n\}$.

The set of states corresponding to coefficients g is denoted by R_g, which can be expressed explicitly as follows

$$R_g = \bigcap_{i=1}^{n} \bigcap_{j=1}^{n} \{x \in \mathbb{R}^n : A(i,j) + x_j \le A(i,g_i) + x_{g_i}\}.$$

The affine dynamics that is active in the above region is

$$x_i(k) = x_{g_i}(k-1) + A(i,g_i), \qquad i \in \{1, \dots, n\}. \tag{2}$$

C. Difference Bound Matrices

In this section, we introduce the notion of Difference Bound Matrices (DBM). DBM will be used in the finite abstraction of MPL systems to represent the (partitioning) region, the dynamics and also the set of states satisfying each atomic proposition.

Definition 1 ([14, Sec. 4.1]): 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 $\bowtie_{i,j} \in \{<, \leq\}$ represents a strict and nonstrict inequality sign, $\alpha_{i,j} \in \mathbb{R} \cup \{+\infty\}$ denotes the upper bound, for $i, j \in \{0, \ldots, n\}$ and 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 $x_1, \ldots, x_n$.

There are some operations defined over DBM such as 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. The interested reader is referred to [12, Sec. 2.3] for more detailed explanation.

D. Finite Abstractions of Transition Systems

1) Transition Systems: A transition system [10, Def. 2.1] 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 is a set of atomic propositions, and
- $L : S \to 2^{AP}$ is a labelling function.

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

2) Linear Temporal Logic: Linear Temporal Logic [10, Def. 5.1] (LTL) formulae over the set AP of atomic propositions are formed according to the following grammar:

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

The semantics of LTL formulae can be seen in [10].

3) Abstractions: Abstraction is a fundamental concept that enables the analysis of large [10, Ex. 7.53] or even infinite [10, 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 labelling 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 2 (Simulation Order [10]): Let $TS_i = (S_i, Act_i, \longrightarrow_i, I_i, AP, L_i)$, $i \in \{1, 2\}$ be transition systems over AP. A simulation for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

1) for each $s_1 \in I_1$ there exists $s_2 \in I_2$ such that $(s_1, s_2) \in \mathcal{R}$
2) for all $(s_1, s_2) \in \mathcal{R}$ 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 $(s_1', s_2') \in \mathcal{R}$

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

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 with equally labeled concrete states only.

Proposition 1 ([10]): Let $TS = (S, Act, \longrightarrow, I, AP, L)$ 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 2 ([10]): Let TS_2 simulates TS_1, assume TS_1 does not have terminal states, let φ be a linear-time property. If TS_2 satises φ, then TS_1 also satises φ.

E. Unified Modeling Language

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.

III. Main Results

A. Auxiliary Classes

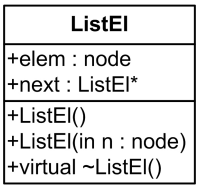

Fig. 1. Class diagram for ListEl

1) ListEl: This class (cf. Fig. 1) is used to represent an element in a singly linked list. Member variables are as follows:

1) `elem` is a variable of type `node`.
2) `next` is a pointer to `ListEl`. If this is not the last element, this variable points to the next element in the list. If this is the last element in the list, this variable does not point to anything.

Member functions are as follows:

1) `ListEl()` is a constructor without any argument. This function does not do anything.
2) `ListEl(n)` is a constructor with one argument of type `node`. In this function, value of the argument is stored in member variable `elem` and pointer `next` is initialized to null.
3) `~ListEl()` is a destructor. This function does not do anything.

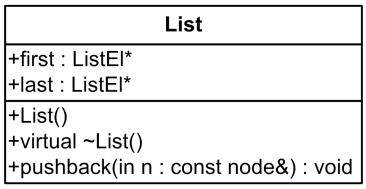

Fig. 2. Class diagram for List

2) List: This class (cf. Fig. 2) is an implementation of singly linked list, where each element is of type `ListEl`. The list stores all leaves of the abstraction tree. Member variables are as follows:

1) `first` is a pointer to `ListEl`. If the list is not empty, this variable points to the first element in the list. If the list is empty, this variable does not point to anything.
2) `last` is a pointer to `ListEl`. If the list is not empty, this variable points to the last element in the list. If the list is empty, this variable does not point to anything.

Member functions are as follows:

1) `List()` is a constructor without any argument. In this function, both member variables are initialized to null. This means that they do not point to anything.
2) `~List()` is a destructor. This function does not do anything.
3) `pushBack(n)` is a function with one argument of type `node`. In this function, the argument is inserted to end of the list.

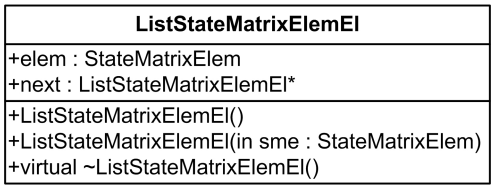

Fig. 3. Class diagram for ListStateMatrixElemEl

3) ListStateMatrixElemEl: This class ((cf. Fig. 3)) is used to represent an element in a singly linked list. Member variables are as follows:

1) `elem` is a variable of type `StateMatrixElem`.
2) `next` is a pointer to `ListStateMatrixElemEl`. If this is not the last element, this variable points to the next element in the list. If this is the last element in the list, this variable does not point to anything.

Member functions are as follows:

1) `ListStateMatrixElemEl()` is a constructor without any argument. This function does not do anything.
2) `ListStateMatrixElemEl(sme)` is a constructor with one argument of type `StateMatrixElem`. In this function, value of the argument is stored in member variable `elem` and pointer `next` is initialized to null.
3) `~ListStateMatrixElemEl()` is a destructor. This function does not do anything.

4) ListStateMatrixElem: This class (cf. Fig. 4) is an implementation of singly linked list, where each element is of type `ListStateMatrixElemEl`. This linked list is used to store entries of the state matrix. We use a linked list, rather than an array, because we do not know size of the state matrix a-priori. Member variables are as follows:

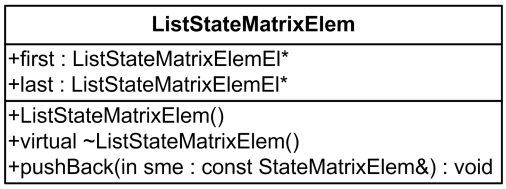

Fig. 4. Class diagram for ListStateMatrixElem

1) `first` is a pointer to `ListStateMatrixElemEl`. If the list is not empty, this variable points to the first element in the list. If the list is empty, this variable does not point to anything.
2) `last` is a pointer to `ListStateMatrixElemEl`. If the list is not empty, this variable points to the last element in the list. If the list is empty, this variable does not point to anything.

Member functions are as follows:

1) `ListStateMatrixElem()` is a constructor without any argument. In this function, both member variables are initialized to null. This means that they do not point to anything.
2) `~ListStateMatrixElem()` is a destructor. This function does not do anything.
3) `pushBack(sme)` is a function with one argument of type `StateMatrixElem`. In this function, the argument is inserted to end of the list.

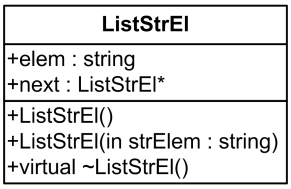

Fig. 5. Class diagram for ListStrEl

5) ListStrEl: This class (cf. Fig. 5) is used to represent an element in a singly linked list. Member variables are as follows:

1) `elem` is a variable of type `string`.
2) `next` is a pointer to `ListStrEl`. If this is not the last element, this variable points to the next element in the list. If this is the last element in the list, this variable does not point to anything.

Member functions are as follows:

1) `ListStrEl()` is a constructor without any argument. This function does not do anything.
2) `ListStrEl(strElem)` is a constructor with one argument of type `string`. In this function, value of the argument is stored in member variable `elem` and pointer `next` is initialized to null.
3) `~ListStrEl()` is a destructor. This function does not do anything.

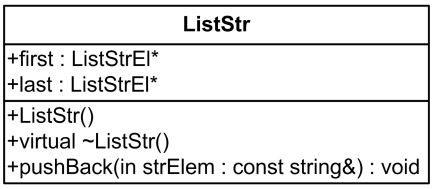

Fig. 6. Class diagram for ListStr

6) ListStr: This class (cf. Fig. 6) is an implementation of singly linked list, where each element is of type `ListStrEl`. This list stores the specifications that are going to be checked against the model. We use a linked list, rather than an array, because we do not know a-priori the number of specifications. Member variables are as follows:

1) `first` is a pointer to `ListStrEl`. If the list is not empty, this variable points to the first element in the list. If the list is empty, this variable does not point to anything.
2) `last` is a pointer to `ListStrEl`. If the list is not empty, this variable points to the last element in the list. If the list is empty, this variable does not point to anything.

Member functions are as follows:

1) `ListStr()` is a constructor without any argument. In this function, both member variables are initialized to null. This means that they do not point to anything.
2) `~ListStr()` is a destructor. This function does not do anything.
3) `pushBack(strElem)` is a function with one argument of type `string`. In this function, the argument is inserted to end of the list.

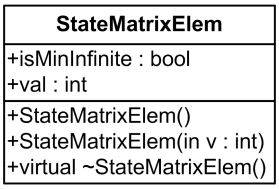

Fig. 7. Class diagram for StateMatrixElem

7) StateMatrixElem: This class (cf. Fig. 7) represents an element of the state matrix of an MPL system, whose value can be either a finite integer or minus infinity. Member variables are as follows:

1) `isMinInfinite` is a Boolean variable that represents whether the element is finite or not. If the value of `isMinInfinite` is true, then the element is minus infinity and the value stored in `val` is ignored. However if the value of `isMinInfinite` is false, then the element equals the value of `val`.
2) `val` is an integer variable which stores the element, if the element is finite, i.e. the value of `isMinInfinite` is false.

Member functions are as follows:

1) `StateMatrixElem()` is a constructor without any argument. In this function, the element is initialized to minus infinity.
2) `StateMatrixElem(v)` is a constructor with one argument `v` of type integer. The element is initialized to the argument of the function.
3) `~StateMatrixElem()` is a destructor. This function does not do anything.

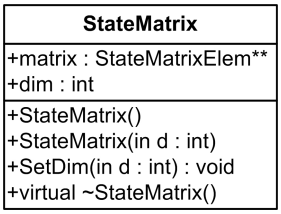

Fig. 8. Class diagram for StateMatrix

8) StateMatrix: This class (cf. Fig. 8) represents the state matrix of an MPL system. The dimension of state matrix must be $n \times n$, which will be allocated dynamically. Member variables are as follows:

1) `matrix` is a variable of type pointer to pointer to `StateMatrixElem`, which is equivalent to a two-dimensional array of `StateMatrixElem`. This variable stores entries of the state matrix.
2) `dim` is an integer variable that stores dimension of the state matrix. In other words, dimension of the state matrix is given by `dim` $\times$ `dim`.

Member functions are as follows:

1) `StateMatrix()` is a constructor without any argument. This function does not do anything.
2) `StateMatrix(d)` is a constructor with one argument of type integer. This function generates a matrix of size `d` $\times$ `d`. The entries are minus infinity.
3) `SetDim(d)` is a function with one argument of type integer. This function generates a matrix of size `d` $\times$ `d`. The entries are minus infinity.
4) `~StateMatrix()` is a destructor. This function does not do anything.

B. Difference-Bound Matrices

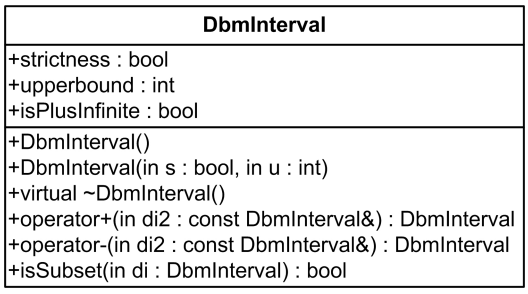

Fig. 9. Class diagram for DbmInterval

1) DbmInterval: This class (cf. Fig. 9) represents an element of a DBM, namely $x_i - x_j \bowtie_{i,j} \alpha_{i,j}$ (cf. Section II-C). Member variables are used to characterize such interval:

1) `strictness` is a Boolean variable. If the value is true, then the inequality sign is not strict, i.e. $\leq$. If the value is false, then the inequality sign is strict, i.e. $<$.
2) `isPlusInfinite` is a Boolean variable. If the value is true, then the upper bound is $+\infty$ and the inequality sign is not strict, i.e. the value of `strictness` is false. In this case, the value of `strictness` is ignored. On the other hand, if the value of `isPlusInfinite` is false, then the upper bound is defined as the value of `upperbound` and the inequality sign depends on the value of `strictness`.
3) `upperbound` is an integer variable that represents the upper bound if the upper bound is finite. If the upper bound is not finite, the value stored in this member variable is ignored.

Member functions are as follows:

1) `DbmInterval()` is a constructor without any argument. In this function, upper bound of the interval is initialized to $+\infty$.
2) `DbmInterval(s,u)` is a constructor with two arguments. The first agument `s` is of type Boolean and the second argument `u` is of type integer. In this function, upper bound of the interval is defined to be finite. Furthermore, the upper bound is defined to be `u` and the strictness is defined to be `s`.
3) `~DbmInterval()` is a destructor. This function does not do anything.
4) `operator+(di2)` is a function that overloads the plus operator. This function adds *this* DbmInterval and the DbmInterval in the argument `di2`. This function returns a DbmInterval, which represents result of the addition.
5) `operator-(di2)` is a function that overloads the minus operator. This function intersects *this* DbmInterval and the DbmInterval in the argument `di2`. This function returns a DbmInterval, which represents result of the intersection.
6) `isSubset(di)` is a function that checks whether *this* DbmInterval is a subset of the DbmInterval in the argument `di`. If *this* DbmInterval is a subset of the DbmInterval in the argument `di`, then this function returns true. Otherwise if *this* DbmInterval is not a subset of the DbmInterval in the argument `di`, then this function returns false.

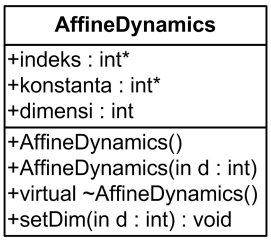

Fig. 10. Class diagram for AffineDynamics

2) AffineDynamics: This class (cf. Fig. 10) represents an affine dynamics generated by an MPL system (2). Member variables are as follows:

1) `indeks` (in English: index) is a variable of type pointer to integer. This variable stores the integer coefficients $g_1, \ldots, g_n$.
2) `konstanta` (in English: constants) is a variable of type pointer to integer. This variable stores integer constants $A(1, g_1), \ldots, A(n, g_n)$.
3) `dimensi` (in English: dimension) is an integer variable. This variable stores size of the state matrix. Remember that the state matrix is a square matrix. Thus, we only need a single integer variable to store its dimension.

Member functions are as follows:

1) `AffineDynamics()` is a constructor without any argument. This function does not do anything.
2) `AffineDynamics(d)` is a constructor with one argument `d` of type integer. In this function, member variables `indeks` and `konstanta` are defined as one-dimensional integer arrays of size `d`. Furthermore, all entries of both member variables are initialized to zero.
3) `~AffineDynamics()` is a destructor. This function does not do anything.
4) `setDim(d)` is a function with one argument `d` of type integer. In this function, member variables `indeks` and `konstanta` are defined as one-dimensional integer arrays of size `d`. Furthermore, all entries of both member variables are initialized to zero.

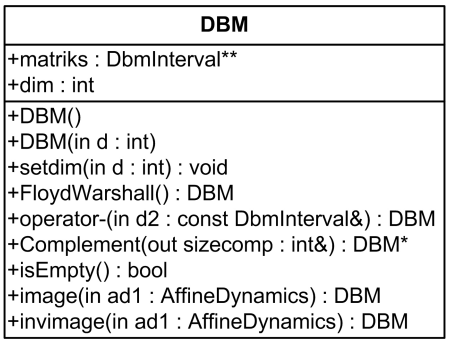

Fig. 11. Class diagram for DBM

3) DBM: This class (cf. Fig. 11) represents a DBM. Member variables are as follows:

1) `matriks` is a variable of type pointer to pointer of `DbmInterval`. In other words, this variable represents a two-dimensional array of type `DbmInterval`.
2) `dim` is an integer variable that represents the size or dimension of the DBM. The special variable x_0 is not considered when determining the dimension of a DBM.

Member functions are as follows:

1) `DBM()` is a constructor without any argument. This function does not do anything.
2) `DBM(d)` is a constructor with one argument of type integer. This function creates DBM $\mathbb{R}^{\mathrm{d}}$. Notice that size of the preceding DBM is `d`.
3) `setdim(d)` is a function with one argument of type integer. This function creates DBM $\mathbb{R}^{\mathrm{d}}$. Notice that size of the preceding DBM is `d`.

4) `FloydWarshall()` is used to compute the canonical-form representation of *this* DBM. Since the canonical-form representation of a DBM is again a DBM, this function returns a DBM.
5) `operator-(d2)` is a function that overloads the minus operator. This function has one argument of type `DBM`. The purpose is to determine the intersection of two DBM. Since the intersection of two DBM is a DBM, this function returns a DBM.
6) `Complement(sizecomp)` is a function to compute the complement of *this* DBM. The complement of a DBM is in general a union of finitely many DBM. This function has an argument of type address of an integer. This argument is not used for the input, but this is used as the output to store the number of DBM that becomes the complement of *this* DBM.
7) `isEmpty()` is a function to check whether *this* DBM is empty or not. This function returns a Boolean value. If this function returns true, then *this* DBM is empty. If this function returns false, then *this* DBM is not empty.
8) `image(ad1)` is a function to determine the image of *this* DBM w.r.t. the affine dynamics defined in the argument. Since the image of a DBM w.r.t. an affine dynamics is a DBM, this function returns a DBM.
9) `invimage(ad1)` is a function to determine the inverse image of *this* DBM w.r.t. the affine dynamics defined in the argument. Since the inverse image of a DBM w.r.t. an affine dynamics is a DBM, this function returns a DBM.

C. Abstraction Classes

1) Node: This class (cf. Fig. 12) represents a node in the partition tree. Member variables are as follows:

1) `d` is a variable of type `DBM`. This variable stores the DBM represented by a node in the partition tree.
2) `dipenuhi` (in English: satisfied) is a variable of type pointer to Boolean. In other words, the data type is an array of Boolean. The size of the array is equal to the number of atomic propositions. If the i-th atomic proposition is satisfied, then the i-th element of the array is true. If the i-th atomic proposition is not satisfied, then the i-th element of the array is false.
3) `ad` is a variable of type `AffineDynamics`. This variable stores the affine dynamics that are active in the DBM represented by this node.
4) `state` is an integer variable that represents the unique identifier for each leaf node. As such, this variable is only used for nodes that become leaf.
5) `numChild` is an integer variable that stores the number of children of this node.
6) `child` is a variable of type pointer to `node`. This variable will be initialized to a dynamic one-dimensional array of type `node`. The number of elements in the array equals the number of children of this node.

Member functions are as follows:

1) `node()` is a constructor without any argument. This function does not do anything.
2) `node(d1,dipenuhi1,numAP,lev,ApDbm)` is a constructor to build the AP partition tree. This function is recursive in the following sense: in this constructor, we create some objects of type `node` that will call this constructor.
3) `node(A,d1,ad1,lev)` is a constructor to build the AD partition tree. This function is recursive in the following sense: in this constructor, we create some objects of type `node` that will call this constructor.
4) `node(d1,dipenuhi1,numAP,lev,ApDbm,A, ad1)` is a constructor to build the Π_0 partition tree. This function is a combination of the preceding two constructors. This function is recursive in the following sense: in this constructor, we create some objects of type `node` that will call this constructor.

2) Tree: This class (cf. Fig. 13) is used to represent a tree, for example *AP* partition tree, *AD* partition tree and Π_0 partition tree. This class has a single member variable `root` of type node. This variable is used to store root of the tree. This class has a constructor and a destructor. Both functions do not do anything.

3) AbstractionTree: This class (cf. Fig. 14) is used to store the abstract transition system. Initially, the partition of the state space is represented as a tree. Member variables are as follows:

1) `numAP` is an integer variable that is used to store the number of atomic propositions.
2) `ApDbm` is a pointer to `DBM`. We assume that the set of states satisfying each atomic proposition is a DBM. This variable is used to represent the set of states that satisfies each atomic proposition.
3) `ApPartTree` is a variable of type `tree` that is used to store the *AP* partition tree.
4) `AdPartTree` is a variable of type `tree` that is used to store the *AD* partition tree.
5) `pi0PartTree` is a variable of type `tree` that is used to store the Π_0 partition tree.
6) `A` is a variable of type `StateMatrix`, which is used to store the state matrix.
7) `pi0PartTreeLeaf` is a variable that represents a list of `nodes`. This variable is used to store the leaf nodes in the Π_0 partition tree.
8) `numpi0PartTreeLeaf` is an integer variable which stores the number of leaf nodes in the Π_0 partition tree.
9) `adj` is a two-dimensional matrix, where each entry is a Boolean variable. This variable is used to represent the set of transitions in the abstract transition system. If the entry in i-th row and j-th column is true, then there is a transition from j-th node to i-th node. If the entry in i-th row and j-th column is false, then there is no transition from j-th node to i-th node. This variable is still used after the refinement phase.
10) `numInitStates` is an integer variable that represents the number of DBM that defines the initial states. The initial states are represented by a union of finitely many DBM.
11) `initStates` is a pointer to `DBM` (or equivalently, an array of `DBM`). This variable stores the initial states of

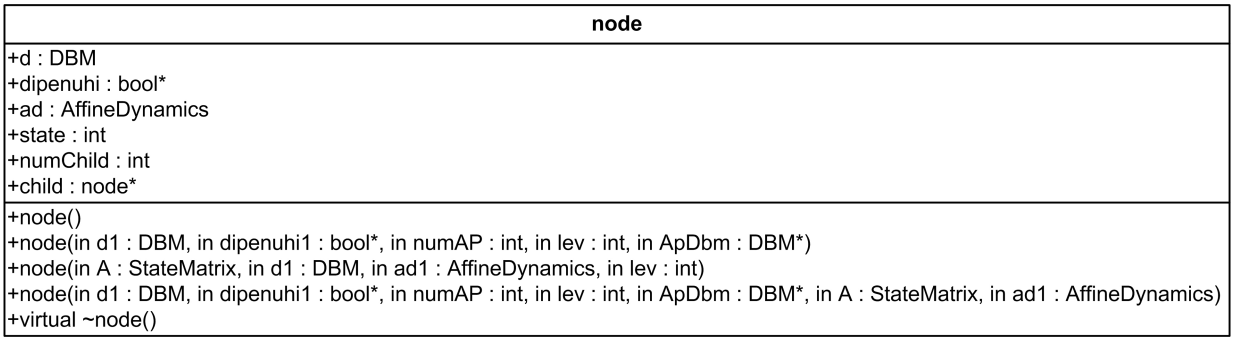

Fig. 12. Class diagram for Node

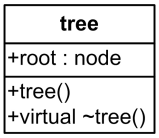

Fig. 13. Class diagram for Tree

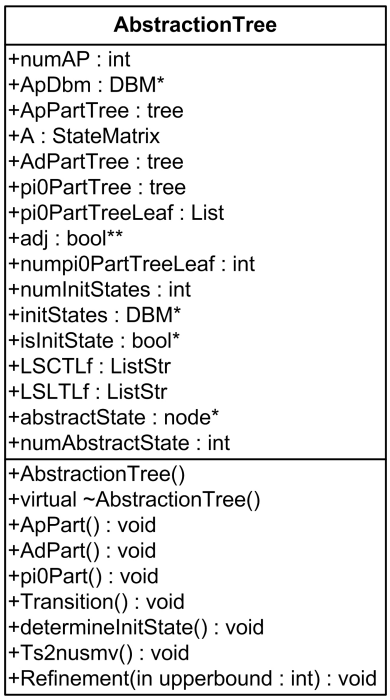

Fig. 14. Class diagram for AbstractionTree

the concrete transition system, i.e. the MPL system.

12) `isInitState` is an array of boolean variables, where the size is the same with the number of abstract states. This variable represents the set of abstract initial states. If the i-th element is `true`, then the i-th abstract state is an initial state. If the i-th element is `false`, then the i-th abstract state is not an initial state.
13) `LSCTLf` is a variable of type `ListStr` which stores the set of CTL specifications.
14) `LSLTLf` is a variable of type `ListStr` which stores the set of LTL specifications.
15) `abstractState` is an array of `node` which represents the set of abstract states. This variable is still used after the refinement phase.
16) `numAbstractState` is an integer variable that represents the number of abstract states. This variable is still used after the refinement phase.

Member functions are as follows:

1) `AbstractionTree()` is a constructor without any argument. This function does not do anything.
2) `~AbstractionTree()` is a destructor. This function does not do anything.
3) `ApPart()` is a function to construct the *AP* partition tree. The result is stored in member variable `ApPartTree`.
4) `AdPart()` is a function to construct the *AD* partition tree. The result is stored in member variable `AdPartTree`.
5) `pi0Part()` is a function to construct the Π_0 partition tree. Roughly speaking, this function is a combination of `ApPart()` and `AdPart()`. The result is stored in member variable `pi0PartTree`.
6) `Transition()` is a function to compute the set of transitions. In order to minimize the memory usage, this function is not implemented in a recursive manner. This function uses the Π_0 partition that is stored as a tree, i.e. member variable `pi0PartTree`. The result is stored in member variable `adj`. Furthermore, this function also initializes member variable `pi0PartTreeLeaf`.
7) `determineInitState()` is a function to determine the set of initial states over the abstract transition system. This function uses member variable `abstractState`. Since member variable `abstractState` is initialized in member function `Refinement(upperbound)`, this function has to be called after execution of the refinement function. The result is stored in member variables `numInitStates` and `initStates`.
8) `Refinement(upperbound)` is used to refine the abstract transition system. In general, this procedure does not necessarily terminate in a finite time. Thus, we define one argument `upperbound` which represents the maximum number of abstract states (for the stopping criterion of the procedure). This function uses member variable `pi0PartTreeLeaf`. Since the member variable is initialized in member function `Transition()`,

then this function can be executed after the execution of the function to compute transitions. Furthermore, in this function, we initialize member variables `abstractState` and `numAbstractState`.

9) `Ts2nusmv()` is a function to generate a NuSMV language stored in a file from the abstract transition system. This function uses member variables `numAbstractState`, `abstractState`, `numInitStates`, `initStates` and `adj`. Thus, this function can be executed after the functions `Transition()`, `Refinement(upperbound)` and `determineInitState()` have been executed.

REFERENCES

[1] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat, *Synchronization and Linearity, An Algebra for Discrete Event Systems*. John Wiley and Sons, 1992.

[2] B. Heidergott, G. Olsder, and J. van der Woude, *Max Plus at Work–Modeling and Analysis of Synchronized Systems: A Course on Max-Plus Algebra and Its Applications*. Princeton University Press, 2006.

[3] B. Roset, H. Nijmeijer, J. van Eekelen, E. Lefeber, and J. Rooda, "Event driven manufacturing systems as time domain control systems," in *Proc. 44th IEEE Conf. Decision and Control and European Control Conf. (CDC-ECC'05)*, Dec. 2005, pp. 446–451.

[4] J. van Eekelen, E. Lefeber, and J. Rooda, "Coupling event domain and time domain models of manufacturing systems," in *Proc. 45th IEEE Conf. Decision and Control (CDC'06)*, Dec. 2006, pp. 6068–6073.

[5] C. A. Brackley, D. S. Broomhead, M. C. Romano, and M. Thiel, "A max-plus model of ribosome dynamics during mRNA translation," *Journal of Theoretical Biology*, vol. 303, no. 0, pp. 128–140, Jun. 2012.

[6] S. Gaubert and R. Katz, "Reachability and invariance problems in max-plus algebra," in *Positive Systems*, ser. Lecture Notes in Control and Information Science, L. Benvenuti, A. De Santis, and L. Farina, Eds. Springer, Heidelberg, Apr. 2003, vol. 294, ch. 4, pp. 15–22.

[7] R. D. Katz, "Max-plus (A, B)-invariant spaces and control of timed discrete-event systems," *IEEE Trans. Autom. Control*, vol. 52, no. 2, pp. 229–241, Feb. 2007.

[8] B. De Schutter, "On the ultimate behavior of the sequence of consecutive powers of a matrix in the max-plus algebra," *Linear Algebra and its Applications*, vol. 307, no. 1-3, pp. 103–117, Mar. 2000.

[9] D. Adzkiya, B. De Schutter, and A. Abate, "Finite abstractions of max-plus-linear systems," *IEEE Trans. Autom. Control*, vol. 58, no. 12, pp. 3039–3053, Dec. 2013.

[10] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[11] D. Adzkiya and A. Abate, "VeriSiMPL: Verification via biSimulations of MPL models," in *Proc. 10th Int. Conf. Quantitative Evaluation of Systems (QEST'13)*, ser. Lecture Notes in Computer Science, K. Joshi, M. Siegle, M. Stoelinga, and P. D'Argenio, Eds., vol. 8054. Springer, Heidelberg, Sep. 2013, pp. 253–256. [Online]. Available: http://sourceforge.net/projects/verisimpl/

[12] D. Adzkiya, Y. Zhang, and A. Abate, "VeriSiMPL 2: An open-source software for the verification of max-plus-linear systems," *Discrete Event Dynamic Systems*, vol. 26, no. 1, pp. 109–145, 2016.

[13] W. Heemels, B. De Schutter, and A. Bemporad, "On the equivalence of classes of hybrid dynamical models," in *Proc. 40th IEEE Conf. Decision and Control*, vol. 1, 2001, pp. 364–369.

[14] D. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Automatic Verification Methods for Finite State Systems*, ser. Lecture Notes in Computer Science, J. Sifakis, Ed. Springer, Heidelberg, 1990, vol. 407, ch. 17, pp. 197–212.