8-28-2014

Formalization of Matrix Theory in HOL4

Zhiping Shi
Chinese Academy of Sciences

Yan Zhang
Capital Normal University

Zhenke Liu
Capital Normal University

Xinan Kang
Capital Normal University

Yong Guan
Capital Normal University

See next page for additional authors

Citation Details

Shi, Z., Zhang, Y., Liu, Z., Kang, X., Guan, Y., Zhang, J., & Song, X. (2014). Formalization of matrix theory in HOL4. Advances in Mechanical Engineering, 6, 195276.

Authors

Zhiping Shi, Yan Zhang, Zhenke Liu, Xinan Kang, Yong Guan, Jie Zhang, and Xiaoyu Song

Hindawi Publishing Corporation
Advances in Mechanical Engineering
Volume 2014, Article ID 195276, 16 pages
http://dx.doi.org/10.1155/2014/195276

Research Article

Formalization of Matrix Theory in HOL4

Zhiping Shi,[1,2,3] Yan Zhang,[1] Zhenke Liu,[1] Xinan Kang,[1]
Yong Guan,[1] Jie Zhang,[4] and Xiaoyu Song[5]

[1] *Beijing Key Laboratory of Electronic System Reliability Technology, Capital Normal University, Beijing 100048, China*
[2] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, 100190, China*
[3] *Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, 541004, China*
[4] *College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China*
[5] *Electrical and Computer Engineering, Portland State University, Portland, OR 97201, USA*

Correspondence should be addressed to Zhiping Shi; shizhiping@gmail.com

Received 11 January 2014; Accepted 14 March 2014; Published 28 August 2014

Academic Editor: Hongxing Wei

Matrix theory plays an important role in modeling linear systems in engineering and science. To model and analyze the intricate behavior of complex systems, it is imperative to formalize matrix theory in a metalogic setting. This paper presents the higher-order logic (HOL) formalization of the vector space and matrix theory in the HOL4 theorem proving system. Formalized theories include formal definitions of real vectors and matrices, algebraic properties, and determinants, which are verified in HOL4. Two case studies, modeling and verifying composite two-port networks and state transfer equations, are presented to demonstrate the applicability and effectiveness of our work.

1. Introduction

The matrix theory is a core subbranch of linear algebra. Matrices as operators of linear space transformations play important roles in modeling linear systems. The matrix theory has extended applications in most of science fields. In many branches of physics, including classical mechanics, optics, electromagnetism, quantum mechanics, and quantum electrodynamics, the matrix theory is used to analyze physical phenomena, such as the motion of rigid bodies. In computer graphics, matrices are used to perform 2-dimensional and 3-dimensional projection transformation. In robotics, matrices are used to address robot kinematics and dynamics. In probability theory and statistics, stochastic matrices are used to describe sets of probabilities; for instance, matrix decomposition supports modeling information compression, reconstruction, and retrieval. Matrix calculus generalizes classical analytical notions such as derivatives and exponentials to higher dimensions. Furthermore, the modern matrix theory covers subjects related to many other important mathematical branches such as graphs, combinatorics, and statistics. MATLAB's success is the typical sample of matrix applications.

Traditionally, a number of efficient numerical analysis algorithms were developed for matrix computations in order to improve the accuracy of results, yet the absolute precision in the real number field can never be reached because of round-off error, approximate algorithms to address large-scale issues, and so on. On the other hand, the property analysis of linear system based models has been done using paper-and-pencil proof methods, which is quite error prone. A tiny error or inaccuracy, however, may result in failure or even loss of lives in highly sensitive and safety-critical engineering applications. Mechanical theorem proving, on the contrary, is capable of performing precise and scalable analysis.

Mechanical theorem proving has been considered a promising and powerful method of formal proofs in pure mathematics or system analysis and verification [1–5]. Systems or any proof goals need to be modeled formally before they are verified by theorem provers, and theorem provers work based on logic theorem libraries of mathematics. The

Table 1: Some HOL4 notations and their semantics.

Meaning	HOL4 notations	Standard notations
Truth	T	T
Falsity	F	$\perp$
Negation	~t	$\neg t$
Disjunction	t1\/t2	$t1 \vee t2$
Conjunction	t1/\t2	$t1 \wedge t2$
Implication	t1==>t2	$t1 \Rightarrow t2$
Equality	t1=t2	$t1 = t2$
∀-quantification	!x.t	$\forall x.t$
∃-quantification	?x.t	$\exists x.t$
Lambda	\x.t	$\lambda x.t$

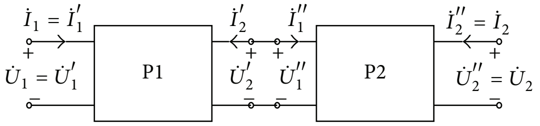

Figure 1: Two ports connected in a cascade connection.

more mathematic theorem libraries there are, the wider the scope of application of the theorem provers is [5]. It is significant to formalize matrix theory in theorem provers for extending theorem proving applications. The parts of the matrix theories have been formalized in some theorem provers. Nakamura et al. [6] presented the formalization of the matrix theory in Mizar in 2006. The COQ system has also started to provide matrices in recent years [7]. Harrison presented the formalization of Euclidean space in the HOL-light system in 2005 [8]. In Isabelle/HOL [9], some basic matrix theory has been formalized in [10, 11]. However, the HOL4 [12], which is the latest version of the HOL theorem prover, does not yet have a matrix theory in its formalized theories collections. Furthermore, no successful conversion of matrix theory from any other theorem prover can be found in the current literature. For this reason, diverse applications are not available to be verified using HOL4. For example, Liu et al. [13] presented a sophisticated formalization in HOL4 for the finite-state discrete-time Markov Chain theory which is widely used to model random processes in physical and informational systems. Without the formalized matrix theory, the state transition matrix was formalized by the list type instead of the matrix type. If our matrix theory were used to supply the formalization of Markov chain, the work in [13] would be enhanced to efficiently address scalable systems. Hence, formalizing the matrix theory in HOL4 enables formally analyzing linear system models using this theorem prover, as well as benefiting the development of enormous other theories, such as Markov chain.

We will formalize the matrix analysis theories by stages. This paper presents a systematic formalization of the matrix algebraic theory in the HOL4 system. It includes the formalization of vectors and matrices and proofs of their relevant algebraic properties. The vector and matrix are defined based on the finite Cartesian products (FCP) library of HOL4.

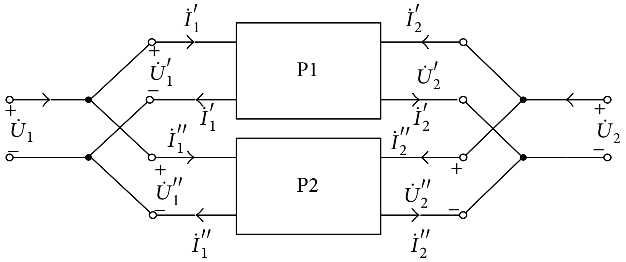

Figure 2: The two ports connected in a parallel-parallel configuration.

The properties of vectors, matrices, and determinants are characterized in accordance with linear space properties. As case studies of applications, the formal modeling and proof of the parameterized two-port networks and high-speed power of matrix solution to state transfer equations are presented. In this paper, we use HOL4 notations, and some notations are listed in Table 1.

This paper is organized as follows. Section 2 proposes the formalization of the vector space. Then, the formalization of the fundamental matrix theory is presented in Section 3. Two applications of modeling and verifying by the proposed approach are presented in Section 4. Section 5 concludes the paper.

2. Formalization of Vector Space

Matrices are transforming operators in the vector space. In this section, vectors and their algebraic operations are formalized based on the FCP library in HOL4, and the linear properties of vector space are proven.

2.1. Defining the Data Type of Vectors. N-dimensional vectors are elements of an N-dimensional vector space, denoted by A^N, where A is the element type and N is a number variable for the dimension. A^N can be constructed by the N-dimensional Cartesian products of A. The function space is as follows:

$$A \longrightarrow N \longrightarrow A^N. \qquad (1)$$

It is not trivial to define the compound type in the HOL4 system based on a simple type theory where a compound type can only depend on other types and not on terms. Harrison [8] introduced an elegant method of defining the vector type in the HOL-light theorem prover. We define n-dimensional vectors in HOL4 following Harrison's method, with the cardinality of the type being the dimension of the Cartesian product. The FCP theory was implemented and named the fcpTheory in HOL4. Assuming A to be a real type and n the index type, the real vector type is constructed based on the fcpTheory in HOL4 as follows:

$$\text{Hol\_type : real ['n] --> 'n vector}, \qquad (2)$$

where 'n stands for a type variable, which can be instantiated by a certain type. The elements of a vector are operated

Table 2: The formalization of the algebraic properties of vectors.

Theorem name	Formalization in HOL4	Mathematic form
VECTOR_ADD_INDEX	|- !v1 v2 i. i < dimindex (:'n) ==> ((v1 + v2) ' i = v1 ' i + v2 ' i)	$(\mathbf{v1} + \mathbf{v2})_i = \mathbf{v1}_i + \mathbf{v2}_i$
VECTOR_SUB_INDEX	|- !v1 v2 i. i < dimindex (:'n) ==> ((v1 − v2) ' i = v1 ' i − v2 ' i)	$(\mathbf{v1} - \mathbf{v2})_i = \mathbf{v1}_i - \mathbf{v2}_i$
VECTOR_ADD_COMM	|- !v1 v2. v1 + v2 = v2 + v1	$\mathbf{v1} + \mathbf{v2} = \mathbf{v2} + \mathbf{v1}$
VECTOR_ADD_MUL_LK	|- !v1 v2 k. k ∗∗ (v1 + v2) = k ∗∗ v1 + k ∗∗ v2	$k(\mathbf{v1} + \mathbf{v2}) = k \cdot \mathbf{v1} + k \cdot \mathbf{v2}$
VECTOR_LMUL_C_ASSOC	|- !k l v. k ∗ l ∗∗ v = (k ∗ l) ∗∗ v	$k \cdot l \cdot \mathbf{v} = (k \cdot l) \cdot \mathbf{v}$
VECTOR_ADD_ASSOC	|- !v1 v2 v3. v1 + v2 + v3 = v1 + (v2 + v3)	$\mathbf{v1} + \mathbf{v2} + \mathbf{v3} = \mathbf{v1} + (\mathbf{v2} + \mathbf{v3})$
VECTOR_EQ	|- !v1 v2. (v1 − v2 = vector_0) <=> (v1 = v2)	$(\mathbf{v1} - \mathbf{v2} = \mathbf{0}) <=> (\mathbf{v1} = \mathbf{v2})$
VECTOR_EQ2	|- !v1 v2. (v1 = v2) <=> !i. i < dimindex (:'n) ==> (v1 ' i = v2 ' i)	$(\mathbf{v1} = \mathbf{v2}) <=> \mathbf{v1}_i = \mathbf{v2}_i \; i = 0, \ldots, n-1$
VECTOR_ADD_LZERO	|- !v. vector_0 + v = v	$\mathbf{0} + \mathbf{v} = \mathbf{v}$
VECTOR_ADD_RZERO	|- !v. v + vector_0 = v	$\mathbf{v} + \mathbf{0} = \mathbf{v}$
VECTOR_ADD_NEG	|- !v. v + ~v = vector_0	$\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$
VECTOR_SUB_ADD	|- !v1 v2. v1 − v2 + v2 = v1	$\mathbf{v1} - \mathbf{v2} + \mathbf{v2} = \mathbf{v1}$
VECTOR_DOT_COMM	|- !v1 v2. v1 ∗∗ v2 = v2 ∗∗ v1	$\mathbf{v1} \cdot \mathbf{v2} = \mathbf{v2} \cdot \mathbf{v1}$
VECTOR_ADD_LDISTRIB	|- !v1 v2 v3. v1 ∗∗ (v2 + v3) = v1 ∗∗ v2 + v1 ∗∗ v3	$\mathbf{v1} \cdot (\mathbf{v2} + \mathbf{v3}) = \mathbf{v1} \cdot \mathbf{v2} + \mathbf{v1} \cdot \mathbf{v3}$
VECTOR_ADD_MUL_RK	|- !v1 v2 k. (v1 + v2) ∗∗ k = v1 ∗∗ k + v2 ∗∗ k	$(\mathbf{v1} + \mathbf{v2})k = \mathbf{v1} \cdot k + \mathbf{v2} \cdot k$
VECTOR_ADD_NEG2	|- !v1 v2. v1 + ~v2 = v1 − v2	$\mathbf{v1} + (-\mathbf{v2}) = \mathbf{v1} - \mathbf{v2}$
VECTOR_ADD_RDISTRIB	|- !v1 v2 v3. (v1 + v2) ∗∗ v3 = v1 ∗∗ v3 + v2 ∗∗ v3	$(\mathbf{v1} + \mathbf{v2}) \cdot \mathbf{v3} = \mathbf{v1} \cdot \mathbf{v3} + \mathbf{v2} \cdot \mathbf{v3}$
VECTOR_DOT_LMUL_K	|- !v1 v2 k. k ∗∗ v1 ∗∗ v2 = (k ∗∗ v1) ∗∗ v2	$k \cdot \mathbf{v1} \cdot \mathbf{v2} = (k \cdot \mathbf{v1}) \cdot \mathbf{v2}$
VECTOR_DOT_RMUL_K	|- !v1 v2 k. v1 ∗∗ k ∗∗ v2 = (k ∗∗ v1) ∗∗ v2	$\mathbf{v1} \cdot k \cdot \mathbf{v2} = (k \cdot \mathbf{v1}) \cdot \mathbf{v2}$
VECTOR_LNEG_UNIQ	|- !v1 v2. (v1 + v2 = vector_0) <=> (v1 = ~v2)	$(\mathbf{v1} + \mathbf{v2} = \mathbf{0}) <=> (\mathbf{v1} = -\mathbf{v2})$
VECTOR_MUL_BASIS	|- !v k. k < dimindex (:'n) ==> (v ∗∗ vector_basis k = v ' k)	$\mathbf{v} \cdot \text{vector\_basis}\, k = \mathbf{v}_k$
VECTOR_MUL_K_EQ	|- !v k. v ∗∗ k = k ∗∗ v	$k \cdot \mathbf{v} = \mathbf{v} \cdot k$
VECTOR_MUL_L1	|- !v. 1 ∗∗ v = v	$1 \cdot \mathbf{v} = \mathbf{v}$
VECTOR_MUL_LRADD	|- !v k l. (k + l) ∗∗ v = k ∗∗ v + l ∗∗ v	$(k + l) \cdot \mathbf{v} = k \cdot \mathbf{v} + l \cdot \mathbf{v}$
VECTOR_MUL_NEG	|- !v1 v2. ~v1 ∗∗ v2 = v1 ∗∗ ~v2	$-\mathbf{v1} \cdot \mathbf{v2} = \mathbf{v1} \cdot (-\mathbf{v2})$
VECTOR_MUL_RRADD	|- !v k l. v ∗∗ (k + l) = v ∗∗ k + v ∗∗ l	$\mathbf{v} \cdot (k + l) = \mathbf{v} \cdot k + \mathbf{v} \cdot l$
VECTOR_NEG	|- !v. ~v = −1 ∗∗ v	$-\mathbf{v} = -1 \cdot \mathbf{v}$
VECTOR_NEG_NEG	|- !v. ~~v = v	$- - \mathbf{v} = \mathbf{v}$
VECTOR_RNEG_UNIQ	|- !v1 v2. (v1 + v2 = vector_0) <=> (v2 = ~v1)	$(\mathbf{v1} + \mathbf{v2} = \mathbf{0}) <=> (\mathbf{v2} = -\mathbf{v1})$
VECTOR_SUB_LZERO	|- !v. vector_0 − v = ~v	$\mathbf{0} - \mathbf{v} = -\mathbf{v}$
VECTOR_EXIST_NEG	|- !v. ?v'. v + v' = vector_0	$\exists\, \mathbf{v}', \mathbf{v} + \mathbf{v}' = \mathbf{0}$
VECTOR_DOT_FCP	|- ($FCP f ∗∗ v = sum (0,dimindex (:'n)) (\i. f i ∗ v ' i)) ∧ (u ∗∗ $FCP f = sum (0,dimindex (:'n)) (\i. u ' i ∗ f i))	$\mathbf{f} \cdot \mathbf{v} = \sum_{i=1,\ldots,n}(f_i \times v_i)$ //**f** is a function $\mathbf{u} \cdot \mathbf{f} = \sum_{i=1,\ldots,n}(u_i \times f_i)$ //**v**, **u** are vectors
VECTOR_MUL_ME	|- !v. v ∗∗ matrix_E = v	$\mathbf{v} \cdot \mathbf{E} = \mathbf{v}$, **E** is the unit vector

Table 3: The formalization of the fundamental algebraic properties of the matrix operation.

Theorem name	Formalization in HOL4	Mathematic form
MATRIX_ADD_COMM	|- !A B. A + B = B + A	$\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$
MATRIX_ADD_ASSOC	|- !A B C. A + B + C = A + (B + C)	$\mathbf{A}+\mathbf{B}+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$
MATRIX_ADD_LDISTRIB	!A B C. (A + B) ** C = A ** C + B ** C	$(\mathbf{A}+\mathbf{B})\cdot\mathbf{C}=\mathbf{A}\cdot\mathbf{C}+\mathbf{B}\cdot\mathbf{C}$
MATRIX_MUL_ASSOC	|- !A B C. (A ** B) ** C = A ** B ** C	$(\mathbf{A}\cdot\mathbf{B})\cdot\mathbf{C}=\mathbf{A}\cdot\mathbf{B}\cdot\mathbf{C}$
MATRIX_MUL_VEC_RADD	|- !A v1 v2. A ** (v1 + v2) = A ** v1 + A ** v2	$\mathbf{A}\cdot(\mathbf{v1}+\mathbf{v2})=\mathbf{A}\cdot\mathbf{v1}+\mathbf{A}\cdot\mathbf{v2}$
MATRIX_MUL_VEC_LADD	|- !A v1 v2. (v1 + v2) ** A = v1 ** A + v2 ** A	$(\mathbf{v1}+\mathbf{v2})\cdot\mathbf{A}=\mathbf{v1}\cdot\mathbf{A}+\mathbf{v2}\cdot\mathbf{A}$
TRANSP_MUL	|- !A B. transp (A ** B) = transp B ** transp A	$(\mathbf{A}\cdot\mathbf{B})^{\mathrm{T}}=\mathbf{B}^{\mathrm{T}}\cdot\mathbf{A}^{\mathrm{T}}$
MATRIX_ADD_L0	|- !A. matrix_0 + A = A	$\mathbf{0}+\mathbf{A}=\mathbf{A}$
MATRIX_ADD_R0	|- !A. A + matrix_0 = A	$\mathbf{A}+\mathbf{0}=\mathbf{A}$
MATRIX_ADD_MUL_LR	|- !A B k. k ** (A + B) = k ** A + k ** B	$k(\mathbf{A}+\mathbf{B})=k\mathbf{A}+k\mathbf{B}$
MATRIX_ADD_MUL_RR	|- !A B k. (A + B) ** k = A ** k + B ** k	$(\mathbf{A}+\mathbf{B})k=\mathbf{A}k+\mathbf{B}k$
MATRIX_ADD_NEG	|- !A. A + ~A = matrix_0	$\mathbf{A}+(-\mathbf{A})=\mathbf{0}$
MATRIX_EQ	|- !A B. (A = B) <=> !i j. i < dimindex (:'m) ∧ j < dimindex (:'n) ==> (A ' i ' j = B ' i ' j)	That two matrices are equal means that all corresponding elements of the two matrices are equal
MATRIX_EC	|- !k. k < dimindex (:'n) ==> (column matrix_E k = vector_basis k)	Get the kth column of the identity matrix
MATRIX_ER	|- !k. k < dimindex (:'n) ==> (row matrix_E k = vector_basis k)	Get the kth row of the identity matrix
MATRIX_MUL_K_EQ	|- !A k. A ** k = k ** A	$\mathbf{A}\cdot k=k\cdot\mathbf{A}$
MATRIX_MUL_LME	|- !A. matrix_E ** A = A	$\mathbf{I}\cdot\mathbf{A}=\mathbf{A}$ ($\mathbf{I}$ stands for identity matrix)
MATRIX_MUL_RME	|- !A. A ** matrix_E = A	$\mathbf{A}\cdot\mathbf{I}=\mathbf{A}$
MATRIX_MUL_LRADD	|- !A k l. (k + l) ** A = k ** A + l ** A	$(k+l)\cdot\mathbf{A}=k\cdot\mathbf{A}+l\cdot\mathbf{A}$
MATRIX_MUL_RRADD	|- !A k l. A ** (k + l) = A ** k + A ** l	$\mathbf{A}\cdot(k+l)=\mathbf{A}\cdot k+\mathbf{A}\cdot l$
MATRIX_NEG	|- !A. ~A = –1 ** A	$-\mathbf{A}=(-1)\cdot\mathbf{A}$
MATRIX_NEG_NEG	|- !A. ~~A = A	$--\mathbf{A}=\mathbf{A}$
MATRIX_MUL_NEG	|- !A B. ~A ** B = A ** ~B	$(-\mathbf{A})\cdot\mathbf{B}=\mathbf{A}\cdot(-\mathbf{B})$
MATRIX_NEG_PROD	|- !A B. ~A ** B = ~(A ** B)	$(-\mathbf{A})\cdot\mathbf{B}=-(\mathbf{A}\cdot\mathbf{B})$
MATRIX_SUB_LZERO	|- !A. matrix_0 – A = ~A	$\mathbf{0}-\mathbf{A}=-\mathbf{A}$
MATRIX_LNEG_UNIQ	|- !A B. (A + B = matrix_0) <=> (A = ~B)	$(\mathbf{A}+\mathbf{B}=\mathbf{0})$ <=> $(\mathbf{A}=-\mathbf{B})$
MATRIX_RNEG_UNIQ	|- !A B. (A + B = matrix_0) <=> (B = ~A)	$(\mathbf{A}+\mathbf{B}=\mathbf{0})$ <=> $(\mathbf{B}=-\mathbf{A})$
MATRIX_PROD_LMUL_K	|- !A B k. (k ** A) ** B = k ** A ** B	$(k\cdot\mathbf{A})\cdot\mathbf{B}=k\cdot\mathbf{A}\cdot\mathbf{B}$
MATRIX_PROD_RMUL_K	|- !A B k. A ** k ** B = k ** A ** B	$\mathbf{A}\cdot k\cdot\mathbf{B}=k\cdot\mathbf{A}\cdot\mathbf{B}$
MATRIX_RMUL_C_ASSOC	|- !A k l. k ** l ** A = (k * l) ** A	$k\cdot l\cdot\mathbf{A}=(k\cdot l)\cdot\mathbf{A}$
TRANSP_TRANSP	|- !A. transp (transp A) = A	$\left(\mathbf{A}^{\mathrm{T}}\right)^{\mathrm{T}}=\mathbf{A}$
MATRIX_ADD_INDEX	|- !A B i j. i < dimindex (:'m) ∧ j < dimindex (:'n) ==> ((A + B) ' i ' j = A ' i ' j + B ' i ' j)	$(\mathbf{A}+\mathbf{B})_{ij}=\mathbf{A}_{ij}+\mathbf{B}_{ij}$
MATRIX_SUB_INDEX	|- !A B i j. i < dimindex (:'m) ∧ j < dimindex (:'n) ==> ((A – B) ' i ' j = A ' i ' j – B ' i ' j)	$(\mathbf{A}-\mathbf{B})_{ij}=\mathbf{A}_{ij}-\mathbf{B}_{ij}$
MATRIX_ROW_ADD	|- !A B i. i < dimindex (:'m) ==> (row (A + B) i = vector_add (row A i) (row B i))	The ith row of $\mathbf{A}+\mathbf{B}$ equals the addition of the ith row of $\mathbf{A}$ and the ith row of $\mathbf{B}$
MATRIX_COLUMN_ADD	|- !A B i. i < dimindex (:'n) ==> (column (A + B) i = vector_add (column A i) (column B i))	The ith column of $\mathbf{A}+\mathbf{B}$ equals the addition of the ith column of $\mathbf{A}$ and the ith column of $\mathbf{B}$
MATRIX_ADD_MUL_VECR	|- !A B v. (A + B) ** v = vector_add (A ** v) (B ** v)	$(\mathbf{A}+\mathbf{B})\cdot\mathbf{v}=(\mathbf{A}\cdot\mathbf{v})+(\mathbf{B}\cdot\mathbf{v})$
MATRIX_ADD_MUL_VECL	|- !A B v. v ** (A + B) = vector_add (v ** A) (v ** B)	$\mathbf{v}\cdot(\mathbf{A}+\mathbf{B})=(\mathbf{v}\cdot\mathbf{A})+(\mathbf{v}\cdot\mathbf{B})$

Table 4: The lemmas for proving MATRIX_MUL_ASSOC.

Lemmas	Formalization in HOL4
Lemma 4: column_FCP	|- k < dimindex (:'n) ==> (column (FCP i j. f i j) k = FCP i. f i k)
Lemma 5: row_FCP	|- k < dimindex (:'m) ==> (row (FCP i j. f i j) k = FCP j. f k j)
Lemma 6: SUM_SUM	|- sum (m,n) (\i. sum (k,l) (\j. f i j)) = sum (k,l) (\j. sum (m,n) (\i. f i j))
Lemma 7: SUM_MULT_L	|- sum (m,n) (\i. f i) ∗ c = sum (m,n) (\i. f i ∗ c)
Lemma 8: SUM_MULT_R	|- c ∗ sum (m,n) (\i. f i) = sum (m,n) (\i. c ∗ f i)

```
- g‘!(v: 'n vector) k:num. k<dimindex(:'n) ==>
(v ∗∗ ((vector_basis k): 'n vector) = v ' k)’;                (∗ The goal to prove ∗)
> val it =
      Proof manager status: 1 proof.
      1. Incomplete goalstack:
             Initial goal:
             !v k. k < dimindex (:'n) ==> (v ∗∗ vector_basis k = v ' k)
       : proofs
- e(REWRITE_TAC [vector_dot_def]);                             (∗ Execute rewrite tactic with vector_dot_def ∗)
OK..
1 subgoal:                                                      (∗ Produce a subgoal ∗)
> val it =                                                      (∗ Echo the subgoal ∗)
      !v k.
        k < dimindex (:'n) ==>
        (sum (0,dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) = v ' k)
        : proof
- e(MP_TAC vmb_lemma3);                                         (∗ Execute Modus Ponens with vmb_lemma3∗)
OK..
1 subgoal:                                                      (∗ Produce a further subgoal ∗)
> val it =                                                      (∗ Echo the subgoal ∗)
      (!v k.                                                    (∗ The subgoal looks like A=A ∗)
          k < dimindex (:'n) ==>
         (sum (0,dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) = v ' k)) ==>
      !v k.
         k < dimindex (:'n) ==>
         (sum (0,dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) = v ' k)
         : proof
- e(SRW_TAC [fcpLib.FCP_ss][]);
OK..
<<HOL message: Initialising SRW simpset . . . done>>
Goal proved.                                                    (∗ Goal is proved ∗)
|- (!v k.                                                       (∗ Echo the proved subgoal in goal stack ∗)
         k < dimindex (:'n) ==>
         (sum (0,dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) =v ' k))
    ==>
    !v k.
         k < dimindex (:'n) ==>
         (sum (0,dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) = v ' k)
Goal proved.
|- !v k.                                                        (∗ Echo the proved subgoal in goal stack ∗)
        k < dimindex (:'n) ==>
        (sum (0,dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) = v ' k)
> val it =
      Initial goal proved.                                      (∗ Echo the proved initial goal in goal stack ∗)
       |- !v k. k < dimindex (:'n) ==> (v ∗∗ vector_basis k = v ' k): proof
```

Algorithm 1: The interactive proof of the property VECTOR_MUL_BASIS.

```
- g‘!(v:'n vector) k:num. k<dimindex(:'n) ==>
       (v ∗∗ ((vector_basis k):'n vector) = v %% k)’;
> val it =
      Proof manager status: 1 proof.
      1. Incomplete goalstack:
            Initial goal:
            !v k. k < dimindex (:'n) ==> (v ∗∗ vector_basis k = v ' k)
        : proofs
- e(REWRITE_TAC [vector_dot_def] THEN                                (∗ Batch of tactics ∗)
    MP_TAC vmb_lemma3 THEN
    SRW_TAC [fcpLib.FCP_ss][]);
OK..
<<HOL message: Initialising SRW simpset . . . done>>
> val it =
      Initial goal proved.
|- !v k. k < dimindex (:'n) ==> (v ∗∗ vector_basis k = v ' k): proof
```

Algorithm 2: The batch-command proof of the property VECTOR_MUL_BASIS.

```
val VECTOR_ARITH_TAC =
    REPEAT GEN_TAC THEN
    REWRITE_TAC[dot_def, GSYM SUM_ADD_COUNT, GSYM SUM_SUB_COUNT,
                   GSYM SUM_LMUL, GSYM SUM_RMUL, GSYM SUM_NEG] THEN
    (MATCH_MP_TAC SUM_EQ_COUNT ORELSE MATCH_MP_TAC SUM_EQ_0_COUNT ORELSE
      GEN_REWRITE_TAC ONCE_DEPTH_CONV empty_rewrites [CART_EQ]) THEN
    SIMP_TAC bool_ss[GSYM FORALL_AND_THM] THEN TRY EQ_TAC THEN
    TRY(HO_MATCH_MP_TAC MONO_ALL) THEN TRY(GEN_TAC) THEN
    REWRITE_TAC[TAUT ‘(a ==> b) ∧ (a ==> c) <=> a ==> b ∧ c’,
                   TAUT ‘(a ==> b) ∨ (a ==> c) <=> a ==> b ∨ c’] THEN
    TRY(MATCH_MP_TAC(TAUT ‘(a ==> b ==> c) ==> (a ==> b) ==> (a ==> c)’)) THEN
    SRW_TAC [FCP_ss] [vec_add_def, vec_sub_def, vec_neg_def, vec_mul_def, VECTOR_0,
                   vec_map_def, vec_map2_def] THEN POP_ASSUM MP_TAC THEN
    REAL_ARITH_TAC;
```

Algorithm 3: The decision procedure for vector arithmetic.

using the indexing operator “'” (or, alternatively, %%) in the fcpTheory. For example, the ith element of a vector, which is written as x_i in mathematics, is denoted by “x ' i” (or x %% i). According to the fcpTheory, two vectors are equal if and only if their corresponding elements are equal.

2.2. Formalizing the Operations of Vectors. This subsection gives the formalization of the operations of N-dimensional vectors. The arithmetic operations of vectors are pointwise on elements of the vectors. In order to conveniently deal with the issue of dimensionality and eliminate the problem of interaction with the FCP binder, two mapping functions are given to simplify the operating of all of the elements of vectors and matrices:

Definition 1 (vec_map_def). Consider the following:

|- !f v. vector_map f v = FCP i. f (v ' i).

Definition 2 (vec_map2_def). Consider the following:

|- !f v1 v2. vector_map2 f v1 v2 = FCP i. f (v1 ' i) (v2 ' i)

where the symbol “|-” is the preceding turnstile for definitions and theorems and “FCP i” means for all 0<= i < size of vectors. Obviously, here “FCP i” has better readability and more expressive power than the lambda calculus “\i,” which does not bound “i” implicitly. The above two definitions are for one and two vectors, respectively. The definitions of addition, subtraction, and negative operators on vectors are given based on the two mapping functions. For readability, “+” and “−” are overloaded to denote addition and subtraction; “∼” denotes negative; and the dollar symbol in front of an operator indicates that the operator has a special syntactic status.

Definition 3 (vector_add_def). Consider the following:

|- $+ = vector_map2 $+.

Definition 4 (vector_sub_def). Consider the following:

|- $− = vector_map2 $−.

```
g‘!(A:('m,'n) matrix) (B:('n,'p) matrix) (C:('p,'q) matrix).
         (A ∗∗ B) ∗∗ C = A ∗∗ (B ∗∗ C)’;
> val it =
      Proof manager status: 1 proof.
      1. Incomplete goalstack:
             Initial goal:
             !A B C. (A ∗∗ B) ∗∗ C = A ∗∗ B ∗∗ C
       : proofs
-e(SRW_TAC[fcpLib.FCP_ss][matrix_prod_def,row_FCP,column_FCP,
      VECTO R_DOT_FCP]THEN
      SRW_TAC [][vector_dot_def, SUM_MULT_L, SUM_MULT_R] THEN
      SRW_TAC [][SUM_SUM] THEN
      MATCH_MP_TAC SUM_EQ THEN
      SRW_TAC [][] THEN
      MATCH_MP_TAC SUM_EQ THEN
      SRW_TAC [][] THEN
      SRW_TAC [fcpLib.FCP_ss][row_def, column_def] THEN
      REAL_ARITH_TAC);
OK..
<<HOL message: Initialising SRW simpset... done>>
> val it =
      Initial goal proved.
      |- !A B C. (A ∗∗ B) ∗∗ C = A ∗∗ B ∗∗ C: proof
```

Algorithm 4: The compact process of proving the property MATRIX_MUL_ASSOC.

```
|- val DET_ROW_SPAN = prove
 (‘!A: ('n,'n) matrix i x.
          i < dimindex(:'n) ∧
          x IN span {ROW j A |j < dimindex(:'n) ∧ ~(j = i)}
          ==> (DET(FCP k. if k = i then ROW i A + x else ROW k A) = DET A)’,      (∗ The goal ∗)
  GEN_TAC THEN GEN_TAC THEN
  SIMP_TAC bool_ss[GSYM AND_IMP_INTRO, RIGHT_FORALL_IMP_THM] THEN
  DISCH_TAC THEN
  HO_MATCH_MP_TAC SPAN_INDUCT_ALT THEN CONJ_TAC THENL
   [AP_TERM_TAC THEN SRW_TAC [FCP_ss][VECTOR_ADD_RID] THEN
    COND_CASES_TAC THEN SRW_TAC [FCP_ss][ROW_DEF],
    ALL_TAC] THEN
  REPEAT GEN_TAC THEN CONV_TAC (DEPTH_CONV SET_SPEC_CONV) THEN
  DISCH_THEN(CONJUNCTS_THEN2 (X_CHOOSE_TAC ‘j:num’) (SUBST_ALL_TAC o SYM)) THEN
  REWRITE_TAC[VECTOR_ADD_ASSOC] THEN
  ONCE_REWRITE_TACVECTOR_ARITH
       “a + c ∗ x + y:real['n] = (a + y) + c ∗ x”] THEN
  ABBREV_TAC ‘z = ROW i (A:real['n]['n]) + y’ THEN
  ASM_SIMP_TAC bool_ss[DET_ROW_MUL, DET_ROW_ADD] THEN   (∗ Using Theorems 3 and 4 ∗)
  MATCH_MP_TAC(prove(‘(d = &0) ==> (a + c ∗ d = a)’,
        STRIP_TAC THEN ASM_REWRITE_TAC[]THEN REAL_ARITH_TAC)) THEN
  MATCH_MP_TAC DET_IDENTICAL_ROWS THEN                   (∗ Using Theorem 5 ∗)
  MAP_EVERY EXISTS_TAC [‘i:num’, ‘j:num’] THEN
  SRW_TAC[FCP_ss][ROW_DEF]).
```

Algorithm 5: The proof of Theorem DET_ROW_SPAN.

Definition 5 (vector_neg_def). Consider the following:

|- $~ = vector_map numeric_negate.

Two kinds of products of vectors are implemented. One is the inner product of vectors:

$$x \times y = \sum_{i=1}^{n} x_i y_i. \qquad (3)$$

```
val TWO_PORT_CASCADE=store_thm(“TWO_PORT_CASCADE”,          (∗ The theorem name ∗)
     “!v1:2 vector v1':2 vector v1":2 vector                (∗ Begin of the theorem ∗)
         v2:2 vector v2':2 vector v2":2 vector
         T:(2,2) matrix    T':(2,2) matrix T":(2,2) matrix.
     (!i. i<dimindex(:2) ==> (v1 %% i = v1' %% i)) ∧
     (!i. i<dimindex(:2) ==> (v2' %% i = v1" %% i)) ∧
     (!i. i<dimindex(:2) ==> (v2" %% i = v2 %% i))∧
     (v1' = T' ∗∗ v2') ∧ (v1" = T" ∗∗ v2") ∧ (v1 = T ∗∗ v2) ==>
     (T' ∗∗ T" ∗∗ v2 = T ∗∗ v2)”,                            (∗ End of the theorem ∗)
     REPEAT GEN_TAC THEN                                    (∗ Begin of the proof ∗)
     REWRITE_TAC [GSYM VECTOR_EQ2] THEN
     PROVE_TAC []);                                         (∗ End of the proof ∗)
```

Algorithm 6: The proved HOL4 theorem of the property of a cascade connection of two ports.

```
Val TWO_PARALLEL_PORT = store_thm(“TWO_PARALLEL_PORT”,
       “!X:2 vector X':2 vector X":2 vector                 (∗ Begin of the theorem ∗)
           U:2 vector U':2 vector U":2 vector
           Y:(2,2) matrix Y':(2,2) matrix Y":(2,2) matrix.
        (!i. i<dimindex(:2) ==> ((U' %% i) = (U %% i))) ∧
        (!i. i<dimindex(:2) ==> ((U" %% i) = (U %% i))) ∧
        (!i. i<dimindex(:2) ==> ((X %% i) = (X' %% i) + (X" %% i))) ∧
        (X' = Y' ∗∗ U') ∧
        (X" = Y" ∗∗ U") ∧
        (X = Y ∗∗ U)==>
        (Y ∗∗ U = (Y' + Y") ∗∗ U)”,                          (∗ End of the theorem ∗)
     REPEAT GEN_TAC THEN
     REWRITE_TAC [GSYM VECTOR_EQ2] THEN
     SUBGOAL_THEN                                           (∗ Begin of SUBGOAL ∗)
         (- -‘(!i. i<dimindex(:2) ==>
         (((X:2 vector) %% i) = (X' %% i) + (X" %% i))) = (X = X' + X")’- -)
     (fn th => ONCE_REWRITE_TAC[th]) THENL
     [‘(!i. i<dimindex(:2) ==>
           (((X:2 vector) %% i) = (X' %% i) + (X" %% i))) =
       (!i. i<dimindex(:2) ==>((X %% i) = (X' + X") %% i))’
       by SRW_TAC [] [VECTOR_ADD_INDEX] THEN
       ‘(!i. i<dimindex(:2) ==>
            (((X:2 vector) %% i) = (X' + X") %% i)) = (X = X' + X")’
       by SRW_TAC [] [VECTOR_EQ2] THEN
       PROVE_TAC [],ALL_TAC] THEN                           (∗ End of SUBGOAL ∗)
     REWRITE_TAC [MATRIX_ADD_MUL_VEC] THEN
     PROVE_TAC []);
```

Algorithm 7: The proved HOL4 theorem of the property of a parallel-parallel configuration of two ports.

The symbol “∑” can be presented by the following function in realTheory:

|- !f n m. (sum (n, 0) f = 0) ∧ (sum (n, SUC m) f = sum (n, m) f + f (n + m)).

We overload “∗∗” for the multiplication of vectors. The inner product (dot product) of two vectors is defined as follows.

Definition 6 (vector_dot_def). Consider the following:

|- !x y. x ∗∗ y = sum (0, dimindex (:'n)) (\i. x ' i ∗ y ' i).

Note that n is a type variable; thus, dimindex is used to obtain the cardinality of the type. Another is a scalar product operation, which multiplies a vector by a scalar. This includes two cases: the scalar may be at the right or left side.

Definition 7 (vector_rmul_def). Consider the following:

|- !k v. k ∗∗ v = FCP i. k ∗ v ' i.

Definition 8 (vector_lmul_def). Consider the following:

|- !v k. v ∗∗ k = FCP i. v ' i ∗ k.

We present two special vectors: the zero vector and the base vectors.

```
// Induction of the solution by power of matrix for state transfer equation
val POW_M_INDUCT = prove
      (‘!(f:num->real['n]) (A:real['n]['n]) n. (!k:num. 1 <= k∧ (f(k) = A ** f(k − 1))) ==> (f(n) = (A pow_matrix n) **
f(0))’,
      REPEAT STRIP_TAC THEN Induct_on ‘n’ THENL [METIS_TAC[pow_matrix, MATRIX_VECTOR_MUL_LID],
MATCH_MP_TAC EQ_TRANS THEN EXISTS_TAC ‘(A pow_matrix 1) ** ((A pow_matrix n) ** f 0)’ THEN CONJ_TAC
THENL [METIS_TAC [MATRIX_VECTOR_MUL_ASSOC, POW_M_1, SUC_SUB1], METIS_TAC[ADD1, POW_M_ADD,
ADD_SYM, MATRIX_VECTOR_MUL_ASSOC]]]
        );

// Definition of power of matrix
val pow_matrix = Define
   ‘($pow_matrix (x:real['n]['n]) 0 = (MAT 1:real['n]['n])) ∧ ($pow_matrix x (SUC n) = ($pow_matrix x n) ** x)’;

// Property: a matrix to power of 1 is identical.
val POW_M_1 = prove
   (‘!x. x pow_matrix 1 = x’,
    GEN_TAC THEN REWRITE_TAC[num_CONV “1:num”] THEN
    REWRITE_TAC[pow_matrix, MATRIX_MUL_LID]);

// Property: power of addition.
val POW_M_ADD = prove
    (‘!x m n. x pow_matrix (m + n) = (x pow_matrix m) ** (x pow_matrix n)’,
    GEN_TAC THEN GEN_TAC THEN INDUCT_TAC THEN
    ASM_REWRITE_TAC[pow_matrix, ADD_CLAUSES, MATRIX_MUL_RID] THEN
    METIS_TAC[MATRIX_MUL_ASSOC]);

//Note: “MAT 1: real['n]['n]” stands for the n-by-n identity matrix.
```

Algorithm 8: Induction proof of the solution by power of matrix for state transfer equation.

```
// Prove that the solution can be computed by high-speed power of matrix
val POW_M_FAST_INDUCT = prove
    (‘!(f:num->real['n]) (A:real['n]['n]) n. (!k:num. 1 <= k∧ (f(k) = A ** f(k−1))) ==> (if EVEN n then (?m. f(n) = ((A
pow_matrix m) ** (A pow_matrix m) ** f(0))) else (?m. f(n) = (((A pow_matrix m) ** (A pow_matrix m) ** A) **
f(0))))’,
    REPEAT STRIP_TAC THEN COND_CASES_TAC THENL[
    REWRITE_TAC[MATRIX_VECTOR_MUL_ASSOC, GSYM POW_M_ADD, GSYM TIMES2] THEN FIRST_X_ASSUM
MP_TAC THEN METIS_TAC[EVEN_EXISTS, TIMES2, POW_M_INDUCT, POW_M_ODD_EVEN, POW_M_EVEN],
    SUBGOAL_THEN ‘!m. ((A pow_matrix m) ** (A pow_matrix m) ** A) = A pow_matrix SUC(2*m)’ ASSUME_TAC
THENL [REWRITE_TAC[MATRIX_MUL_ASSOC, ADD1, ADD_ASSOC, POW_M_ADD, TIMES2, POW_M_1], ALL_TAC]
THEN UNDISCH_TAC ‘~EVEN n’ THEN REWRITE_TAC[EVEN_ODD, ODD_EXISTS] THEN METIS_TAC[TIMES2,
POW_M_INDUCT, POW_M_ODD_EVEN, POW_M_ODD]]
    );

// To prove above theorem the following properties are presented.
// Proof of highspeed power of matrix.
val POW_M_ODD_EVEN = prove
    (‘!x n. if EVEN n then (?m. x pow_matrix n = (x pow_matrix m) ** (x pow_matrix m)) else (?m. x pow_matrix n =
(x pow_matrix m) ** (x pow_matrix m) ** x)’,
    SRW_TAC[][] THEN POP_ASSUM MP_TAC THENL [SRW_TAC[][EVEN_EXISTS], SRW_TAC[][EVEN_ODD,
ODD_EXISTS]] THEN EXISTS_TAC ‘m’ THENL [METIS_TAC[TIMES2, POW_M_ADD], SRW_TAC[][TIMES2,
POW_M_ADD, POW_M_1, MATRIX_MUL_ASSOC, ADD_SUC, pow_matrix]]);

// High-speed power of matrix for odd and even respectively.
val POW_M_ODD = prove
    (‘!x n. ODD n ==> (?m. x pow_matrix n = (x pow_matrix m) ** (x pow_matrix m) ** x)’,
    METIS_TAC[POW_M_ODD_EVEN, ODD_EVEN]);
val POW_M_EVEN = prove
    (‘!x n. EVEN n ==> (?m. x pow_matrix n = (x pow_matrix m) ** (x pow_matrix m))’,
    METIS_TAC[POW_M_ODD_EVEN, ODD_EVEN]);
```

Algorithm 9: Proof of the solution by high-speed power of matrix for state transfer equation.

```
// Example of frog jumping: state function and state transfer matrix
// State function for frog jumping
val FROG_STATE_DEF = Define’
      FROG_STATE f g = (\k. (vector_dim2 (f k) (g k)))’;
// State transfer matrix for frog jumping
val FROG_TRANS_DEF = Define’
      FROG_TRANS p = (matrix_dim22 0 1 (p – 1) (p – 2))’;
// Prove high-speed power of matrix solution by POW_M_FAST_INDUCT
val POW_M_FAST_INDUCT_EXAMPLE = prove
   (‘!n p f g. (!k:num. 1 <= k∧ ((FROG_STATE f g) k = FROG_TRANS p ∗∗ (FROG_STATE f g)(k – 1))) ==>
      (if EVEN n
            then (?m. (FROG_STATE f g) n = ((FROG_TRANS p pow_matrix m) ∗∗ (FROG_TRANS p pow_matrix m)
∗∗ (FROG_STATE f g) 0))
            else (?m. (FROG_STATE f g) n = (((FROG_TRANS p pow_matrix m) ∗∗ (FROG_TRANS p pow_matrix m)
∗∗ FROG_TRANS p) ∗∗ (FROG_STATE f g) 0)))’,
      REPEAT GEN_TAC THEN MATCH_ACCEPT_TAC POW_M_FAST_INDUCT
   );
// Function used above: directly assign a 2-element vector with 2 real numbers
val vector_dim2_def = Define’
     vector_dim2 v1 v2 = (FCP j. if j = 0then v1 else v2):real[2]’;
// Function used above: directly assign a 2-by-2 matrix with 4 real numbers
val matrix_dim22_def = Define’
     matrix_dim22 m1 m2 m3 m4 =
      (FCP j k. if (j = 0) ∧ (k = 0) then m1 else (if (j = 0) ∧(k = 1) then m2 else (if (j = 1) ∧ (k = 0) then m3 else
m4))):real[2][2]’;
```

Algorithm 10: Instantiating proof of the solution for the frog jumping problem.

Definition 9 (vector_0_def). Consider the following:

|- vector_0 = FCP i. 0.

Definition 10 (vector_basis_def). Consider the following:

|- !k. vector_basis k = FCP i. if i = k then 1 else 0.

2.3. Proofs of the Algebraic Properties of Vectors. In this subsection, the algebraic properties of vectors are formalized and verified. Most of the properties are linear properties, because vector space is linear space. Table 2 shows the formalization of these properties.

In Table 2, **v**, **v1**, **v2**, and **v3** represent vectors and k, l represent scalar constants. These properties have been proven based on the definitions in Section 2.2, and the proofs are pointwise on elements of the vectors. To illustrate the process of proving the properties, we present the proof of the property VECTOR_MUL_BASIS in Algorithm 1. The proving is a cumbersome process. This property says that the inner product of one vector and a base vector produces the corresponding elements of the vector. We prove three lemmas as follows to support the proof of VECTOR_MUL_BASIS.

Lemma 1 (vmb_lemma1). *Consider the following*:

|- !v k. k < dimindex (:'n) ==>

(sum (0, dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) =

sum (0, dimindex (:'n)) (\i. if i = k then v ' k else 0)).

Lemma 2 (vmb_lemma2). *Consider the following*:

|- !v k. k < dimindex (:'n) ==>

(sum (0, dimindex (:'n)) (\i. if i = k then v ' k else 0) = v ' k).

Based on Lemmas 1 and 2, it is easy to prove Lemma 3.

Lemma 3 (vmb_lemma3). *Consider the following*:

|- !v k.

k < dimindex (:'n) ==>

(sum (0, dimindex (:'n)) (\i. v ' i ∗ vector_basis k ' i) = v ' k).

Algorithm 1 shows the detailed interaction of proving the property VECTOR_MUL_BASIS, where "-" is the command prompt in HOL4; "g" guides the proving goal; "e" guides the tactics of proving; ">" is the echo prompt; and "|-" guides the goal or subgoal that is proved. If all of the tactics involved in the proof are known, they can be sequenced together with "THEN" to construct a batch-command style process. The batch-command style proving process takes just one step, as shown in Algorithm 2.

Choosing strategies and theorems for each step of proofs is tedious work. Fortunately, to some extent decision procedures can help to automatically produce a proof. In practice, a simple decision procedure named VECTOR_ARITH_TAC is developed by putting together many potentially useful theorems, definitions, and strategies. The decision procedure is shown in Algorithm 3. This procedure can automatically prove most of the arithmetic properties of vectors.

3. Formalization of Fundamental Matrix Theory

3.1. Defining the Data Type of Matrices. A matrix is a two-dimensional array of numbers with many rows and columns. A matrix type is defined in the same way as a vector type. A row or a column of a matrix is a vector; thus, we use a Cartesian product twice to present the $M \times N$ matrix type:

$$A \longrightarrow \ N \longrightarrow \ M \longrightarrow \left(A^{N}\right)^{M}. \tag{4}$$

The HOL4 type is written as follows:

$$\text{Hol\_type : real ['n] ['m] --> ('m, 'n) matrix.} \tag{5}$$

As with vectors, one can generally use “x ' i ' j” where informally one would write x_{ij} for indexing. The fcpTheory can ensure that two matrices are equal if and only if their corresponding elements are equal.

3.2. Formalizing the Operations of Matrices. This subsection presents the formalization of the arithmetic operations of matrix theory. The arithmetic operations of matrices are pointwise on elements of matrices. The mapping function for vectors can be easily generalized to matrix type, given as follows.

Definition 11 (matrix_map_def). Consider the following:

|- !f m. matrix_map f m = FCP i j. f (m ' i ' j).

Definition 12 (matrix_map2_def). Consider the following:

|- !f m1 m2. matrix_map2 f m1 m2 = FCP i j. f (m1 ' i ' j) (m2 ' i ' j).

The usual operations of addition, subtraction, and negation of matrices are defined as follows.

Definition 13 (matrix_add_def). Consider the following:

|- matrix_add = matrix_map2 \$+.

Definition 14 (matrix_sub_def). Consider the following:

|- matrix_sub = matrix_map2 \$−.

Definition 15 (matrix_neg_def). Consider the following:

|- matrix_neg = matrix_map numeric_negate.

These operations are defined based on the mapping functions. Obviously, the matrices involved in addition and subtraction must have the same number of rows and columns, and the elements of both matrices are dealt with in the same order.

The multiplication of matrices is based on the inner products of vectors. Therefore, this operation is defined after the definitions of the row extracting and the column extracting operations. Letting A, B be matrices of R, the definitions are as follows.

Definition 16 (row_def). Consider the following:

|- !A k. row A k = FCP j. A ' k ' j.

Definition 17 (column_def). Consider the following:

|- !A k. column A k = FCP i. A ' i ' k.

Definition 18 (matrix_prod_def). Consider the following:

!A B. A ∗∗ B = FCP i j. row A i ∗∗ column B j.

Note that the dimension of the rows of matrix A must be equal to the number of columns of matrix B. This requirement must be satisfied to obtain the inner product of vectors.

In addition, other operations are defined, such as transposition, multiplication with a vector or a real number, and exponentiation.

Definition 19 (matrix_transp_def). Consider the following:

|- !A. transp A = FCP i j. A ' j ' i.

Definition 20 (matrix_lmul_vector_def). Consider the following:

|- !v A. v ∗∗ A = FCP i. v ∗∗ column A i.

Definition 21 (matrix_rmul_vector_def). Consider the following:

|- !A v. A ∗∗ v = FCP i. row A i ∗∗ v.

Definition 22 (matrix_lmul_scalar_def). Consider the following:

|- !k A. k ∗∗ A = FCP i j. k ∗ A ' i ' j.

Definition 23 (matrix_rmul_scalar_def). Consider the following:

|- !A k. A ∗∗ k = FCP i j. A ' i ' j ∗ k.

Definition 24 (matrix_pow_def). Consider the following:

|- (!A. matrix_pow A 0 = matrix_E) ∧ !A k. matrix_pow A (SUC k) = A ∗∗ matrix_pow A k.

Inverse matrices are useful for many applications, such as analyzing groups of linear equations. We present the definition, which says that a square matrix may have an inverse matrix, after the definition of the identity matrix.

Definition 25 (matrix_E_def). Consider the following:

```
|- matrix_E = FCP i j. if i = j then 1 else 0.
```

Definition 26 (matrix_inv_def). Consider the following:

```
|- !A. matrix_inv A <=> ?A'. (A ** A' = matrix_E) ∧
(A' ** A = matrix_E).
```

The definition of the zero matrix, whose elements are all 0, is given as follow.

Definition 27 (matrix_0_def). Consider the following:

```
|- matrix_0 = FCP i j. 0.
```

3.3. Verification of the Algebraic Properties of Matrices. The fundamental algebraic properties of matrices are formalized and verified in this subsection. The properties are formally modeled in terms of the above definitions and shown in Table 3.

The properties are proven in a pointwise way based on the definitions of the matrix operations and the vector properties. As an example, we present proofs of a frequently used property named MATRIX_MUL_ASSOC. To reduce proofs of the properties, the lemmas in Table 4 are proved in advance.

Algorithm 4 shows the proof of MATRIX_MUL_ASSOC. First, the definitions of matrix_prod_def, row_FCP, and column_FCP are used to expand matrix products into vector products, and then the definition of vector_dot_def is used to expand the vector products into summations of real products. Finally, the conclusion that the corresponding elements of both sides are equal is drawn. Only the batch-command style proving process is shown in Algorithm 4.

The formalization of these theorems forms a base of reasoning for the transformation of the linear system.

Some special matrices play important parts in applications. For example, the square matrix applies on the determinant. In the next subsection, we will present the formalization of determinant.

3.4. Formalization of the Determinant. In the matrix theory, the determinant is a value defined only for square matrices and indicates discriminative information. When the matrix is that of the coefficients of a group of linear equations, that the determinant is nonzero or zero determines that the system has a unique solution exactly or there are either no solutions or many solutions, respectively. When the matrix corresponds to a linear transformation of a vector space, the nonzero determinant means that the transformation has an inverse operation. In this subsection, we present the formalization of the determinant.

The determinant of a square matrix A is evaluated by the entries of A. The determinant of a matrix of arbitrary size can be defined by the Leibniz formula:

$$\mathrm{DET}\left(\mathbf{A}: R^{n\times n}\right) = \sum_{p\in S_n} \mathrm{SIGN}\left(p\right) \prod_{i=0}^{n-1} a_{ip(i)}, \qquad (6)$$

where $a_{ip(i)}$ is the entry of ith row and $p(i)$ column of $\mathbf{A}$; $p(i)$ is the ith element of a permutation $\mathbf{p}$ of the subscript set $\{0, 1, 2, \ldots, n-1\}$; and SIGN( ) is the signature function of permutations. A permutation is a function that reorders the set of natural numbers. The value in the ith position after the reordering p is denoted by $p(i)$. The sum, denoted by S_n, is computed over all permutations, denoted S_n, of the subscript set $\{0, 1, 2, \ldots, n-1\}$. So, the determinant of an n-by-n matrix $\mathbf{A}$ is defined as follows.

Definition 28 (determinant). Consider the following:

```
|- DET(A:('n, 'n) matrix) =

    SUM {p | p PERMUTES count(dimindex (:
    'n))}

        (\p. SIGN(p) * (PRODUCT
        (count(dimindex (:'n))) (\i. A ' i ' (p
        i)))),
```

where permutation is defined by

```
|- p PERMUTES s = (!x. x NOT IN s ==> p x = x) ∧
(!y. ?!x. p x = y).
```

The definition claims that p is a permutation of a set of natural numbers s. Informally, a permutation of a set of natural numbers is an arrangement of those natural numbers into a particular order. In the above formal definition, for any natural number y, there must exist a position x of permutation p where y dwells. It is a sophisticated definition rather than a trivial translation.

The signature of a permutation p is denoted by SIGN p and defined as +1 if p is even and –1 if p is odd.

Consider the following:

```
|-(SIGN p):real = if EVENPERM p then &1 else – &1.
```

The function EVENPERM estimates the parity of a permutation. The parity of a permutation p can be estimated by the parity of the times of swap operating for transforming the identity permutation, denoted by I, into p. EVENPERM is defined as follows:

```
|- EVENPERM p = EVEN(@n. SWAPSEQ n p),
```

where "SWAPSEQ n p" means that p could be converted from the identify permutation I via performing the swap operator n times. SWAPSEQ is defined with mathematical induction as follows:

```
|- val (SWAPSEQ_RULES, SWAPSEQ_INDUCT,
SWAPSEQ_CASES) =

    Hol_reln

        '(SWAPSEQ 0 I) ∧ (*Basis*)
        (!a b p n. SWAPSEQ n p ∧ ~(a = b) ==>
        SWAPSEQ (SUC n) (SWAP(a,b) o p))';
        (*Induction step*).
```

The induction basis states that I could become I with 0 times swap; and the induction step argues that if p could become I with n times swap, then the permutation which is converted by swapping a and b elements of p could become I with n+1 (SUC n) times swap. “SWAP(a,b) o p” produces a new permutation by swapping ath and bth elements of p, where “o” is a combining operator.

The determinant has many interesting properties. Some basic properties of the determinants are presented as follows.

Theorem 1 (DET_TRANSPOSE). *The determinant of transpose matrix A^T equals that of A. Consider the following:*

|- !A:('n, 'n) matrix. DET (TRANSP A) = DET A.

Theorem 2 (DET_ROW_ZERO). *If all elements of an arbitrary column of matrix A are 0, then its determinant is 0. Consider the following:*

|- !A: ('n, 'n) matrix i. i < dimindex(:'n) ∧ (ROW i A = VECTOR_0) ==> (DET A = &0).

Theorem 3 (DET_ROW_ADD). *If one column (row) of a matrix* **A** *is written as a sum of two column (row) vectors, a and b, and all other columns (rows) are left unchanged, then the determinant of* **A** *is the sum of the determinants of the matrices obtained from* **A** *by replacing the column (row), respectively, by a and b. Consider the following:*

|-!a b c k. k < dimindex(:'n)

==> (DET ((FCP i. if i = k then a + b else c i): ('n, 'n) matrix) =

DET ((FCP i. if i = k then a else c i): ('n, 'n) matrix) +
DET ((FCP i. if i = k then b else c i): ('n, 'n) matrix).

The proofs of the above theorems are intuitive according to the definitions.

Theorem 4 (DET_ROW_MUL). *When multiplying a scalar to a column (row) of the matrix, its determinant will be multiplied by the same scalar. Consider the following:*

|-!a b c k. k < dimindex(:'n)

==> (DET((FCP i. if i = k then c ∗ a else b i): ('n, 'n) matrix) =

c ∗ DET((FCP i. if i = k then a else b i): ('n, 'n) matrix)).

Theorem 4 could be proven by Theorem 3 intuitively.

Theoremx 5 (DET_IDENTICAL_ROWS). *If two columns (rows) of a matrix are identical, then its determinant is 0. Consider the following:*

|-!A: ('n, 'n) matrix i j.

i < dimindex(:'n) ∧ j < dimindex(:'n) ∧ ~(i = j) ∧ (ROW i A = ROW j A)

==> (DET A = &0).

In the determinant of A, for any permutation p_ij {. . . , i, . . . , j, . . .}, there must exist p_ji = SWAP (i,j) o p_ij, which holds SIGN p_ij = ∼ SIGN p_ji, and (PRODUCT (count(dimindex (:'n))) (\i. A ' i ' (p_ij i))) = (PRODUCT (count(dimindex (:'n))) (\i. A ' i ' (p_ji i))). So, the theorem could be proven.

Theorem 6 (DET_DEPENDENT_ROWS). *If the columns (rows) of the* a *matrix form a linearly dependent set, then the determinant of the matrix is 0. Consider the following:*

|- ‘!A: ('n, 'n) matrix. dependent(ROWS A) ==> (DET A = &0).

Because the rows of A are linearly dependent, one row could be rewritten as a linear combination of other rows. Then the determinant of A is rewritten as the sum of several separated determinants by Theorem 3. The matrices corresponding to the separated determinants have duplicated rows, so the separated determinants are all equal to 0 according to Theorem 5.

Theorem 7 (DET_ROW_SPAN). *Adding a linear combination of the other columns (rows) to one row leaves the determinant unchanged. The formalization and proof are show in Algorithm* 5.

To prove Theorem 7, the determinant of the new matrix, which is formed by adding linear combinations of rows of A into any one of the rows, is rewritten as a sum of the subdeterminants in accordance with Theorems 3 and 4. One of the matrices corresponding to the subdeterminants is matrix A, and the rest of corresponding matrices have duplicated rows. Then, Theorem 4 is employed to finish the proof.

4. Applications

In this section, two formal modeling and proving applications are presented, parameterized two-port networks and state transfer equations.

4.1. Parameterized Two-Port Networks. The behavior of many electronic components can be described by their characteristic matrices. Many complex passive and linear circuits can be modeled using a two-port network model [14], which is used to model an isolate portion of a larger circuit in mathematical circuit analysis techniques. Two-port networks can describe any linear circuit with four terminals provided that it does not contain an independent source and satisfies the port conditions. The examples include filters, matching networks, transmission lines, transformers, and small-signal models for transistors. It is meaningful to formally model a two-port network for formally modeling and verifying complex linear circuits. Here, we present the formal models of two-port networks. A two-port network is abstracted as a black box with four terminals: voltage U_1 and current I_1 at the input port and voltage U_2 and current I_2 at the output port. When any two of the four variables are given, the other two can always be derived by a certain 2 × 2 parameter matrix. Furthermore, when two or more two-port networks are connected, the parameters of the combined network can be calculated by performing matrix algebra on the parameter matrices of the component two ports.

Two-port networks can be connected in different ways. We verify two connecting styles: the cascade configuration as shown in Figure 1 and the parallel-parallel configuration as shown in Figure 2.

When the two-port P1 and the two-port P2 are connected in a cascade configuration, they form a composite two-port network, as shown in Figure 1 with symbol definitions. Let the transmission parameters of P1 and P2 be matrices T' and T''. The individual two-port networks are described as follows:

$$\begin{bmatrix} \dot{U}_1' \\ \dot{I}_1' \end{bmatrix} = T' \begin{bmatrix} \dot{U}_2' \\ -\dot{I}_2' \end{bmatrix}, \qquad \begin{bmatrix} \dot{U}_1'' \\ \dot{I}_1'' \end{bmatrix} = T'' \begin{bmatrix} \dot{U}_2'' \\ -\dot{I}_2'' \end{bmatrix}. \tag{7}$$

Let

$$\begin{gathered} v_1 = \begin{bmatrix} \dot{U}_1 \\ \dot{I}_1 \end{bmatrix}, \qquad v_1' = \begin{bmatrix} \dot{U}_1' \\ \dot{I}_1' \end{bmatrix}, \qquad v_1'' = \begin{bmatrix} \dot{U}_1'' \\ \dot{I}_1'' \end{bmatrix}, \\ v_2 = \begin{bmatrix} \dot{U}_2 \\ -\dot{I}_2 \end{bmatrix}, \qquad v_2' = \begin{bmatrix} \dot{U}_2' \\ -\dot{I}_2' \end{bmatrix}, \qquad v_2' = \begin{bmatrix} \dot{U}_2'' \\ -\dot{I}_2'' \end{bmatrix}. \end{gathered} \tag{8}$$

For a cascade configuration,

$$\begin{gathered} \dot{U}_1 = \dot{U}_1', \qquad \dot{U}_2' = \dot{U}_1'', \qquad \dot{U}_2'' = \dot{U}_2, \\ \dot{I}_1 = \dot{I}_1', \qquad \dot{I}_2' = -\dot{I}_1'', \qquad \dot{I}_2'' = \dot{I}_2. \end{gathered} \tag{9}$$

We have

$$v_1 = v_1', \qquad v_2' = v_1'', \qquad v_2'' = v_2. \tag{10}$$

Let T be the parameter matrix of the composite two-port network; we can then make the following deduction:

$$v_1 = v_1' = T'v_2' = T'v_1'' = T'T''v_2'' = T'T''v_2 = Tv_2. \tag{11}$$

Therefore, we have the relationship of the composite two-port parameter and the parameters of individual two ports connected in a cascade connection:

$$T'T'' = T. \tag{12}$$

The mathematical process is formally verified in HOL4, as shown in Algorithm 6. The property is proved based on the definition of the multiplication of matrices and the theorem of the equality of vectors. Note that the “2” in the “2 vector” is a type, whose cardinality is 2.

Note that “v %% i” is used to replace “v ' i” simply to avoid confusion with superscripts “′” and “″.” It can be seen from Algorithm 6 that the formalized vector and matrix are used to model the two-port network and its property, and the proof is very brief thanks to the formalized theory.

When two-port P1 and two-port P2 are connected in a parallel-parallel configuration, they form a composite two-port network, as shown in Figure 2 with symbol definitions. The input voltage and output voltage of the composite two-port networks equal that of the individual two ports, respectively; that is,

$$\dot{U}_1 = \dot{U}_1' = \dot{U}_1'', \qquad \dot{U}_2 = \dot{U}_2' = \dot{U}_2''. \tag{13}$$

If each of the port conditions is not changed by the parallel connection, the current of the composite two-port networks is equal to the sum of the current of the component two ports:

$$\dot{I}_1 = \dot{I}_1' + \dot{I}_1'', \qquad \dot{I}_2 = \dot{I}_2' + \dot{I}_2''. \tag{14}$$

Let

$$\begin{gathered} X = \begin{bmatrix} \dot{I}_1 \\ \dot{I}_2 \end{bmatrix}, \qquad X' = \begin{bmatrix} \dot{I}_1' \\ \dot{I}_2' \end{bmatrix}, \qquad X'' = \begin{bmatrix} \dot{I}_1'' \\ \dot{I}_2'' \end{bmatrix}, \\ U = \begin{bmatrix} \dot{U}_1 \\ \dot{U}_2 \end{bmatrix}, \qquad U' = \begin{bmatrix} \dot{U}_1' \\ \dot{U}_2' \end{bmatrix}, \qquad U'' = \begin{bmatrix} \dot{U}_1'' \\ \dot{U}_2'' \end{bmatrix}. \end{gathered} \tag{15}$$

That is to say,

$$X = X' + X'', \qquad U = U' = U''. \tag{16}$$

Let Y' and Y'' be the y-parameter matrices of P1 and P2, respectively; then,

$$X' = Y'U', \qquad X'' = Y''U''. \tag{17}$$

Let Y be the y-parameter matrix of the composite two-port network. We have

$$X = X' + X'' = Y'U' + Y''U'' = \left(Y' + Y''\right)U = YU. \tag{18}$$

The equation shows that the y-parameters of the composite network are found by the matrix addition of the two individual y-parameter matrices of two ports connected in a parallel-parallel configuration; that is,

$$Y = Y' + Y''. \tag{19}$$

The above mathematic process is verified formally in HOL4, as shown in Algorithm 7.

In the proof, a lemma “(!i. i<dimindex(:2) ==> ((X %% i) = (X′ %% i) + (X″ %% i))) = (X = X′ + X″),” which has not previously been proven, is needed; it is introduced as a subgoal. In addition, the theorem MATRIX_ADD_MUL_VEC is used to prove the goal.

4.2. State Transfer Equations. In this subsection, we present an example that the high-speed power of matrix is employed to solve state transfer equation problems, and the formalization and verification are illustrated. The problem is described as follows. There are p ($2 \leq p \leq 1000000$) lotus flowers in a lake and on one of the flowers there is a frog. The frog is capable of jumping from any flower to any other one. The frog can and must move from a flower it stays on to another flower by a jumping. If the frog starts to jump from a flower and comes back the same flower by n $(2 \le n \le 2^{31} - 1)$ jumping, how many jumping paths are there in all? The solution can be derived by iteration. Let $f[n]$ denote the number of jumping paths by n jumping, and $g[n]$ denotes the number of jumping paths from one flower to another. So, the iterations can be written as

$$\begin{aligned} f[n] &= (p-1) \times g[n-1], \\ g[n] &= f[n-1] + (p-2) \times g[n-1]. \end{aligned} \tag{20}$$

Consider $(f[n] \quad g[n])$ as state vector on time n; then $(f[n-1] \quad g[n-1])$ is state vector on time $n-1$; the state transfer equation is

$$(f[n] \quad g[n]) = (f[n-1] \quad g[n-1]) \begin{pmatrix} 0 & 1 \\ p-1 & p-2 \end{pmatrix}. \tag{21}$$

The solution is easy to be deduced iteratively as

$$(f[n] \quad g[n]) = (f[0] \quad g[0]) \begin{pmatrix} 0 & 1 \\ p-1 & p-2 \end{pmatrix}^{n}. \tag{22}$$

When n could be very large, the high-speed power of matrix is indispensable to compute the solution.

In general, many state transfer problems could be modeled by state transfer equations and the high-speed power of matrix can speed up computing the results. In the rest of the section, we formally prove that the state transfer equation could be solved by power of matrix and further prove that there is high-speed power of matrix speeding up computing the solution.

First, prove that if "$\mathbf{f}(k) = \mathbf{A} \cdot \mathbf{f}(k-1)$" holds, then "$\mathbf{f}(n) = (\mathbf{A}^n) \cdot \mathbf{f}(0)$" holds. The proof is conducted by mathematical induction on n and shown in Algorithm 8. The first step is to prove "$\mathbf{f}(1) = \mathbf{A}^1 \cdot \mathbf{f}(0)$"; and the second step is to prove that if "$\mathbf{f}(n) = \mathbf{A}^n \cdot \mathbf{f}(0)$" holds, then "$\mathbf{f}(n+1) = \mathbf{A}^{n+1} \cdot \mathbf{f}(0)$" holds.

Second, prove that the solution can be computed by high-speed power of matrix. The solution by power of matrix could be derived by POW_M_INDUCT, and then the solution by high-speed power of matrix is verified. To prove this, the high-speed power of matrix is proved in advance. The proof is shown in Algorithm 9.

Third, the problem of the frog jumping could be formalized and verified by instantiating the above proof according to (22). The proof is shown in Algorithm 10.

5. Conclusions

Vectors and matrices are extensively used to model linear transformation of engineer and scientific problems. In this paper, the vector and matrix algebra, which are the fundamentals of linear system models, were formalized in the HOL4 theorem prover. Vectors and matrices were constructed based on the FCP library; then the properties of the operations of vectors and matrices were formally verified. The formalized vector and matrix theories help to extend the applications of HOL4. In order to illustrate the usefulness of the formalized matrix theory, we formally analyzed the behaviors of two kinds of composite two-port networks and high-speed power of matrix solution for state transfer equations. The proposed approach is able to offer exact results and is not subject to slip up. Our future work will focus on the formalization of properties of linear transformation and the function matrix in HOL4, and the formalized matrix analysis theories will be employed to model and verify linear systems in engineering and scientific domains.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the International Cooperation Program on Science and Technology (2010DFB10930 and 2011DFG13000), the National Natural Science Foundation of China (60873006, 61070049, 61170304, 61104035, 61373034, and 61303014), the Natural Science Foundation of the City of Beijing (4122017), the S&R Key Program of the Beijing Municipal Education Commission (KZ201210028036), and the Open Project Program of State Key Laboratory of Computer Architecture and the Open Project Program of Guangxi Key Laboratory Trusted Software.

References

[1] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–193, 1999.

[2] W. Wu and X. Gao, "Mathematics mechanization and applications after thirty years," *Frontiers of Computer Science in China*, vol. 1, no. 1, pp. 1–8, 2007.

[3] J. Liu and H. Lin, "Proof system for applied Pi calculus," in *Theoretical Computer Science*, vol. 323 of *IFIP Advances in Information and Communication Technology*, pp. 229–243, Springer, Berlin, Germany, 2010.

[4] Y. Li, W. N. N. Hung, and X. Song, "A novel formalization of symbolic trajectory evaluation semantics in Isabelle/HOL," *Theoretical Computer Science*, vol. 412, no. 25, pp. 2746–2765, 2011.

[5] L. Chang, Z. Shi, T. Gu, and L. Zhao, "A family of dynamic description logics for representing and reasoning about actions," *Journal of Automated Reasoning*, vol. 49, no. 1, pp. 1–52, 2012.

[6] Y. Nakamura, N. Tamura, and W. Chang, "A theory of matrices of real elements," *Formalized Mathematics*, vol. 14, no. 1, pp. 21–28, 2006.

[7] I. Pasca, "Formally verified conditions for regularity of interval matrices," in *Intelligent Computer Mathematics*, vol. 6167 of *Lecture Notes in Computer Science*, pp. 219–233, Springer, Berlin, Germany, 2010.

[8] J. Harrison, "A HOL theory of Euclidean space," in *Theorem Proving in Higher Order Logics*, vol. 3603 of *Lecture Notes in Computer Science*, pp. 114–129, Springer, Berlin, Germany, 2005.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2002.

[10] S. Obua, *Flyspeck II: the basic linear programs [Ph.D. thesis]*, Technische Universitat Munchen, Munich, Germany, 2008.

[11] S. Obua, "Proving bounds for real linear programs in Isabelle/HOL," in *Theorem Proving in Higher Order Logics*, vol. 3603 of *Lecture Notes in Computer Science*, pp. 227–244, Springer, Berlin, Germany, 2005.

[12] K. Slind and M. Norrish, "A brief overview of HOL4," in *Theorem Proving in Higher Order Logics*, vol. 5170 of *Lecture Notes in Computer Science*, pp. 28–32, Springer, Berlin, Germany, 2008.

[13] L. Liu, O. Hasan, and S. Tahar, "Formal reasoning about finite-state discrete-time Markov chains in HOL," *Journal of Computer Science and Technology*, vol. 28, no. 2, pp. 217–231, 2013.

[14] R. C. Jaeger and T. N. Blalock, *Microelectronic Circuit Design*, McGraw-Hill, Boston, Mass, USA, 3rd edition, 2006.