Theano: A Tool for Verifying the Consistency and Completeness in Tabular Requirements

Aurora Francesca Zanenga
a.zanenga@studenti.unibg.it
University of Bergamo
Dalmine, BG, Italy

Nunzio Marco Bisceglia
n.bisceglia1@studenti.unibg.it
University of Bergamo
Dalmine, BG, Italy

Benedetta Ippoliti
b.ippoliti@studenti.unibg.it
University of Bergamo
Dalmine, BG, Italy

Andrea Bombarda
andrea.bombarda@unibg.it
University of Bergamo
Dalmine, BG, Italy

Angelo Gargantini
angelo.gargantini@unibg.it
University of Bergamo
Dalmine, BG, Italy

Akshay Rajhans
arajhans@mathworks.com
MathWorks
Natick, MA, USA

Claudio Menghi
claudio.menghi@unibg.it
University of Bergamo
Dalmine, BG, Italy
McMaster University
Hamilton, ON, Canada

Abstract

Tabular requirements prescribe software behavior through an "if-then" paradigm. Verifying the consistency and completeness of tabular requirements is crucial, as it can identify specification errors, reduce development time and costs, and help prevent potential failures. Theano is a verification tool designed to ensure the consistency and completeness of tabular requirements formulated with Simulink Requirements Tables. This demonstration paper presents the key features of Theano, illustrates its application through an automotive case study, and discusses the underlying implementation and design choices. An online video walkthrough of the case study is also available: youtu.be/p71bKupmRUQ

CCS Concepts

• **Software and its engineering** → **Requirements analysis**; *Formal software verification.*

Keywords

Simulink, Requirements Table, Tabular Expressions, Consistency, Completeness.

ACM Reference Format:
Aurora Francesca Zanenga, Nunzio Marco Bisceglia, Benedetta Ippoliti, Andrea Bombarda, Angelo Gargantini, Akshay Rajhans, and Claudio Menghi. 2025. Theano: A Tool for Verifying the Consistency and Completeness in Tabular Requirements. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728599

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FSE Companion '25, June 23–28, 2025, Trondheim, Norway
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1276-0/25/06
https://doi.org/10.1145/3696630.3728599

1 Introduction

Tabular requirements specifications (e.g., [8, 10, 16]) have been extensively studied by the academic community (e.g., [7, 9, 17, 18, 21]) and are applied in industry to support the specification and analysis of software requirements [6, 22]. These specifications are facilitated by industrial tools, such as the Requirements Table (RT) block [14] from the Simulink® Requirements Toolbox™ [13].

Software engineers require tools that support the design of tabular requirements and check for their *completeness and consistency*. Recently, we introduced Theano [1], a tool designed to verify the completeness and consistency of tabular requirements represented as Requirements Tables. Completeness and consistency checks [11] enable engineers to determine whether the requirements fully define system behavior for all possible inputs (*completeness*) and ensure that they do not compel the system to exhibit contradictory behaviors for the same input (*consistency*). Existing tools for the analysis of tabular requirements specifications assist software engineers in verifying the consistency and completeness (e.g., [5, 21]). For example, Simulink provides mechanisms to detect inconsistencies and incomplete requirements [12], contingent upon specific syntactic conditions. Theano advances these capabilities by offering enhanced support for *flexible* and *stateful* requirements.

This paper details the implementation of Theano and demonstrates its application using an automotive running example. Specifically, the implementation part discusses the main design decisions for Theano and presents a class diagram summarizing its implementation structure. The demonstration illustrates how Theano supports the design of RT and how it can help detect incompletenesses and inconsistencies in the requirements.

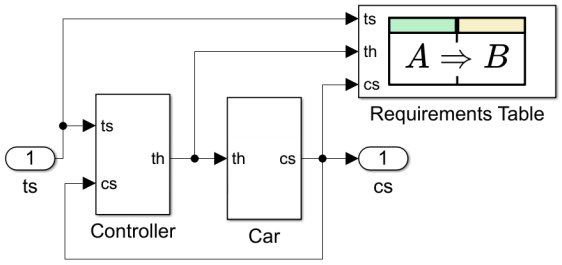

Figure 1: Simulink Model for a Speed Cruise Control System

ID	Pre.	Post.
1	cs < ts	th > **prev** (th)

(a) Version v1

ID	Pre.	Post.
1	cs < ts	th > **prev** (th)
2	cs > ts	th < **prev** (th)
3	ts − 2 ≤ cs ≤ ts + 2	th == **prev** (th)

(b) Version v2

ID	Pre.	Post.
1	cs ≤ ts	th ≥ **prev** (th)
2	cs ≥ ts	th ≤ **prev** (th)
3	ts − 2 ≤ cs ≤ ts + 2	th == **prev** (th)

(c) Version v3

Figure 2: Requirements Tables for Our Running Example

This paper is organized as follows: Section 2 presents our illustrative example and introduces the concept of flexible and stateful requirements, Section 3 describes the implementation of Theano, Section 4 demonstrates its application in the running example, and Section 5 concludes with a discussion of our findings and directions for future work.

2 Flexible and Stateful Requirements

Tabular requirements can be defined in any Simulink model, such as the one for a vehicle's Cruise Control system reported in Figure 1. The Cruise Control system is designed to maintain a constant vehicle speed by automatically adjusting acceleration or deceleration as needed. The model in Figure 1 includes an RT. Figure 2 presents three potential RT examples applicable to the Cruise Control system. Each row in the table represents a system requirement, defined by an identifier (**ID**), a precondition (**Pre.**), and a postcondition (**Post.**). The inputs for the RTs are the current speed of the vehicle (`cs`), the desired target speed (`ts`), and the throttle (`th`) applied to the vehicle.

Flexible requirements do not specify exact values in the postcondition of a requirement. Unlike *rigid* postconditions, which define precise values for system variables, flexible requirements specify a range of values for the variables to support multiple equally valid implementations. For example, the requirement from the Requirements Table in Figure 2a is flexible: When the precondition is satisfied (`cs<ts`), the postcondition (`th` > **`prev`** (`th`)) does not prescribe an exact value for the throttle variable (`th`).

Stateful requirements use the previous value of a variable (a.k.a. "previous state") in the precondition of a requirement. This enables engineers to specify the temporal behaviors of a system. For example, the third requirement from the Requirements Table in Figure 2b is stateful: When the precondition is satisfied ($ts - 2 \leq cs \leq ts + 2$), the postcondition (`th` == **`prev`** (`th`)) specifies that the value of the `th` equals its previous value. The requirement is stateful because the value the `th` assumes depends on the value it assumed in the previous step (a.k.a. state).

Table 1: Encodings for the Requirements Tables

Encoding	Bound	Trace	Semantics
`UeUfFs`	Unbounded	Function	Fixed Step
`UeUfVs`	Unbounded	Function	Variable Step
`UeArFs`	Unbounded	Array	Fixed Step
`UeArVs`	Unbounded	Array	Variable Step
`BeUfFs`	Bounded	Function	Fixed Step
`BeUfVs`	Bounded	Function	Variable Step
`BeArFs`	Bounded	Array	Fixed Step
`BeArVs`	Bounded	Array	Variable Step

In practice, requirements often contain at least some flexible and stateful requirements. Simulink and many other tools have limited support for the analysis of consistency and completeness in stateful requirements, as reasoning about temporal behaviors introduces additional complexity. The availability of tools to analyze such requirements would enable earlier validation of requirements, possibly leading to better requirements and safer implementations. Theano mitigates this need by supporting flexible and stateful requirements. To reach this objective, Theano is implemented as follows.

3 Implementation

Figure 3 presents the main components of Theano: the RT2Text (1) and the text2Z3 (2) components. The RT2Text (1) component is a MATLAB (R2023a) script that extracts the requirements from the Requirements Tables. The text2Z3 (2) component is a translator developed in antlr [4] (v4.5) that translates the Requirements Tables into a logical formula embedded into a Z3Py [19] Python script. The script embeds a call to an SMT solver (Z3 [20]) that returns *sat* if the logical formula is satisfiable, or *unsat* if it is not. This section describes the text2Z3 (2) component since

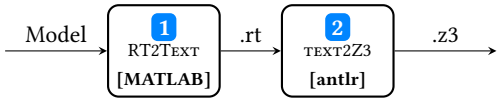

Figure 3: Implementation of THEANO

it produces the logical formulae encoding the requirements and implements the eight different encodings supported by THEANO.

To encode Requirements Tables and verify completeness and consistency, THEANO supports the eight encodings from Table 1. The encodings enable engineers to select whether they want to consider an unbounded (`Ue`) or bounded (`Be`) encoding, they plan to use a fixed (`Fs`) or a variable (`Vs`) step solver for simulating their models, and they want the SMT solver to use arrays (`Ar`) or uninterpreted functions (`Uf`). An extensive description of the advantages and disadvantages of each of these encodings and their implication on software development is out of scope, and the interested reader can refer to our original publication [1]. The TEXT2Z3 (2) component is designed to support all these encodings.

Figure 4 presents the class diagram, for a portion of the TEXT2Z3 component, describing the relevant classes and their relations. The abstract class *RT2Z3Visitor* translates an RT into a Z3 formula. This class implements the methods provided by the interface *RTVisitor*, which describes how to translate the different constructs of an RT. Specifically, the interface *RTVisitor* contains one method for every RT construct that specifies the translator's behavior for that specified rule. The class *RT2Z3Visitor* parameterizes the interface *RTVisitor* with the type returned by its methods (*Z3Formula*). The class *RT2Z3Visitor* has some abstract methods whose behavior depends on the selected encoding and is implemented by its sub-classes.

The *RT2Z3Visitor* class is extended by two abstract classes, *BoundedVisitor* and *UnboundedVisitor*, detailing the translation for some constructs of the Requirements Tables. The *BoundedVisitor* class has two attributes: the bound and the current index. The method *visit*, when parameterized with an object of type *Variable*, uses the index to generate a formula referring to the value of the signal variable at that index. The method *visit* applied to an object of type *PrevExpression* calculates the previous index from the value of the index attribute. The methods *visit* applied to an object of type *IsStartup* or *IsNotStartup* check if the index refers to the system startup time. The attribute *bound* contains the bound to be considered by the bounded encoding and is used by the subclasses of the *BoundedVisitor* via the method *getBound*. The *UnboundedVisitor* class provides an alternative unbounded implementation for the abstract methods of the class *RT2Z3Visitor*. Since this encoding is not bounded, these methods use a variable "*i*" to generate a formula encoding the value of the signal in a generic index.

The *BoundedVisitor* and *UnboundedVisitor* classes are extended by four different classes each representing the four bounded (i.e., `BeUfFs`, `BeUfVs`, `BeArFs`, `BeArVs`) and unbounded (i.e., `UeUfFs`, `UeUfVs`, `UeArFs`, `UeArVs`) encodings. Figure 4 presents the classes *BeArFs* and *UeArFs* as an example. These classes override the *visit* method parameterized with an object of class *DurFormula* that requires an invariant to hold for a specific duration. The implementation of this method depends on the encoding: The fixed and variable steps differ in how the portion of the trace to be considered is computed and whether functions or arrays to encode the trace are used. To realize this behavior, we use the strategy pattern as detailed in Figure 4.

Figure 4 shows that the *RT2Z3Visitor* is related with an encoder represented by class *Encoder* defining (a) the monotonicity constraint mandating the time to increase (method *getMonConst*), (b) a string that defines the variable used to encode the trace (method *getVarDef*), (c) an expression that represents the value assumed by a signal (*sn*) at a specific position (*pos*)[1] (method *getTrPos*), (d) a formula that is true if a position is a startup position (method *isStart*), (e) an expression that for a given position returns the value of that signal in its previous position (method *prev*). The *BoundedEncoder* and *UnboundedEncoder* classes define the behavior of the method *getMonConst* for a finite and infinite number of positions for the trace. The implementations of the encoder classes use the methods provided by the *StepEncoder* class.

The *StepEncoder* class has two subclasses and provides the methods *defTimestamp* and *getSig* that rely on the *TraceEncoder*. The abstract method *getMConst* (used by method *getMonConst* of the *Encoder*) returns the monotonicity constraint for a specific signal *s* and position *p*. The *FixedStepEncoder* and the *VariableStepEncoder* provide two implementations for this method.

The *TraceEncoder* provides two methods: *defTimestamp* defines the variable that will encode the timestamp of the system, and *getSig* returns an *Z3Exp* that accesses the value of the signal *s* at index *i*. The behaviors of these methods depend on whether an encoding based on arrays or an uninterpreted function is selected and implemented by the classes *UfEncoder* and *ArEncoder*.

To summarize, depending on the selected encodings an object of appropriate classes *BeUfFs*, *BeUfVs*, *BeArFs*, *BeArVs*, *UeUfFs*, *UeUfVs*, *UeAfFs*, or *UeArVs* are created. These classes use an appropriate *Encoder*, *StepEncoder*, and *TraceEncoder* to convert the RTs into Z3 formulae.

4 Demonstration

We used the `BeUfFs` encoding for our demonstration, based on the results reported in the research paper associated with this demonstration [15]. We demonstrate how THEANO helps find incompleteness and inconsistencies on the RT from our automotive running example.

Specifically, THEANO detects that the RT shown in Figure 2a is *incomplete*; version *v1* of the RT does not specify the system behavior when the current speed (`cs`) is greater or equal to the target speed (`ts`).

To solve the problem, engineers proposed the modified version *v2* of the RT, as shown in Figure 2b. This version incorporates cases where the current speed (`cs`) exceeds the target speed (`ts`) and where it is within ± 2 km/h of the target speed. THEANO confirms that the RT is *complete* since it specifies the value of the outputs for all the possible inputs. However, THEANO detects that the RT is *inconsistent*. If the current speed (`cs`) is lower than the target speed (`ts`), the preconditions of requirements 1 and 3 are satisfied. However, the postcondition of requirement 1 forces the throttle

[1]Note that the position is modeled by a String since it can represent a generic position with index "i."

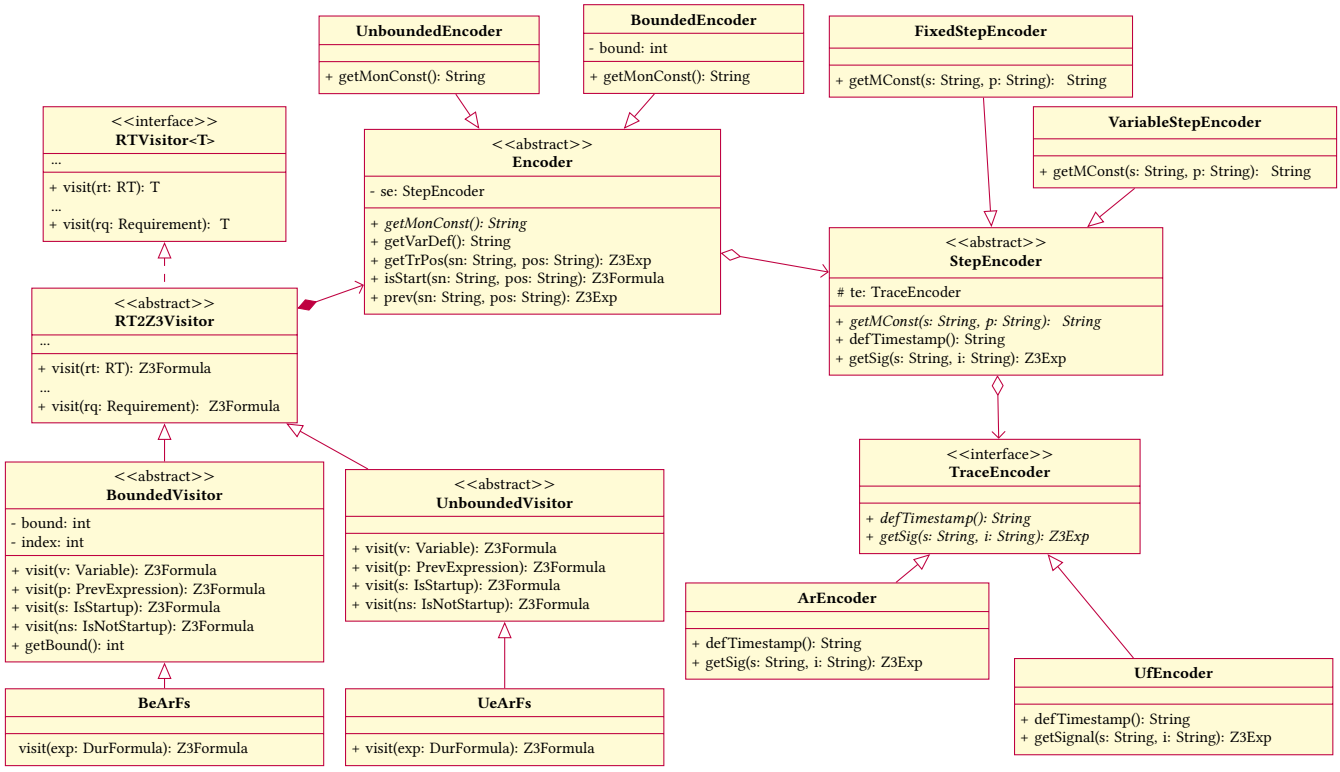

Figure 4: Class Diagram for the Translator of Theano

```
java -jar Theano.jar -i TableCar_v3.rt -o TableCar_v3_completeness.py
-e BeUfFs -t completeness -b 6

java -jar Theano.jar -i TableCar_v3.rt -o TableCar_v3_consistency.py
-e BeUfFs -t consistency -b 6

python TableCar_v3_completeness.py; python TableCar_v3_consistency.py

Requirements Table Complete (unsat)
Requirements Table Consistent (unsat)
```

Figure 5: Results of Our Running Example from Theano

(th) to be greater than the previous value (**prev** (th)), while the postcondition of requirement 3 forces its value to match its previous value (**prev** (th)). This conflict leads to an inconsistency in the RT.

To resolve this, engineers developed version *v3* of the RT, shown in Figure 2c, by modifying the preconditions and postconditions for requirements 1 and 2. Theano can not detect any *incompleteness* and *inconsistency* of this RT since (a) it defines the software behaviors for all the possible inputs and (b) it does not force the system to behave differently when two preconditions are satisfied.

Figure 5 shows the results of our demonstration using Theano, which runs both the completeness and consistency checks for the version *v3* of the RT from Figure 2c. The tool requires as input the Requirements Table (`-i`), the encoding to be used (`-e`), the type of check to be done (`-t`), and the bound (`-b`). Each tool call produces a Python file (specified with the `-o` argument) which must be executed to actually perform the checks.

5 Conclusion

This demonstration paper shows how the scientific contribution from [15] has been transferred into the tool Theano. Unlike the original publication [15], it demonstrates the tool with an automotive case study and discusses our implementation details and design decisions. Theano is integrated with Simulink. Future work will consider integrating it with other tools.

Tool and Data Availability

Theano is publicly available as an open source tool [1]. Two videos showing how to install Theano [3] and how to run the example [2] are available online.

Acknowledgments

This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU, by the European Union - Next Generation EU. "Sustainable Mobility Center (Centro Nazionale per la Mobilità Sostenibile - CNMS)," M4C2 - Investment 1.4, Project Code CN_00000023, by PNC - ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 – CUP: B53C22006700001, and by EC under project GLACIATION (101070141).

References

[1] FOSELAB, University of Bergamo 2023. *Theano*. FOSELAB, University of Bergamo. https://github.com/foselab/Theano/

[2] FOSELAB, University of Bergamo 2025. *Theano Demo Walkthrough*. FOSELAB, University of Bergamo. https://www.youtube.com/watch?v=p71bKupmRUQ

[3] FOSELAB, University of Bergamo 2025. *Theano Installation Guide*. FOSELAB, University of Bergamo. https://www.youtube.com/watch?v=mkdkDA_5MFI

[4] antlr. 2023. *antlr*. antlr. https://www.antlr.org/

[5] Marsha Chechik and John Gannon. 2001. Automatic analysis of consistency between requirements and designs. *Transactions on Software Engineering* 27, 7 (2001), 651–672. https://doi.org/10.1109/32.935856 IEEE.

[6] Judith Crow and Ben Di Vito. 1998. Formalizing Space Shuttle software requirements: Four case studies. *Transactions on Software Engineering and Methodology* 7, 3 (1998), 296–332. https://doi.org/10.1145/287000.287023 ACM.

[7] Colin Eles and Mark Lawford. 2011. A Tabular Expression Toolbox for Matlab/Simulink. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 494–499.

[8] Stuart Roland Faulk and David Lorge Parnas. 1989. *State determination in hard-embedded systems*. Ph. D. Dissertation. The University of North Carolina at Chapel Hill. AAI9007280.

[9] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. 1996. Automated consistency checking of requirements specifications. *Transactions on Software Engineering and Methodology* 5, 3 (1996), 231–261. ACM.

[10] Kathryn L. Heninger. 1980. Specifying software requirements for complex systems: New techniques and their application. *Transactions on Software Engineering* SE-6, 1 (1980), 2–13. https://doi.org/10.1109/TSE.1980.230208 IEEE.

[11] MathWorks. 2023. *Identify Inconsistent and Incomplete Formal Requirement Sets*. MathWorks. https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html

[12] Mathworks. 2023. *Identify Inconsistent and Incomplete Formal Requirement Sets*. Mathworks. https://www.mathworks.com/help/slrequirements/ug/check-requirements-table-block.html

[13] Mathworks. 2023. *Requirements Toolbox*. Mathworks. https://www.mathworks.com/products/requirements-toolbox.html

[14] MathWorks. 2023. *Use a Requirements Table Block to Create Formal Requirements*. MathWorks. https://www.mathworks.com/help/slrequirements/ug/use-requirements-table-block.html

[15] Claudio Menghi, Eugene Balai, Darren Valovcin, Christoph Sticksel, and Akshay Rajhans. 2024. Completeness and Consistency of Tabular Requirements: An SMT-Based Verification Approach. *Transactions on Software Engineering* (2024). https://doi.org/10.1109/TSE.2025.3530820

[16] S Meyers and Stephanie White. 1983. Software requirements methodology and tool study for A6-E technology transfer.

[17] Linna Pang, Chen-Wei Wang, Mark Lawford, and Alan Wassyng. 2014. Formalizing and Verifying Function Blocks Using Tabular Expressions and PVS. In *Formal Techniques for Safety-Critical Systems*, Cyrille Artho and Peter Csaba Ölveczky (Eds.). Springer International Publishing, Cham, 125–141.

[18] Dennis K. Peters, Mark Lawford, and Baltasar Trancón y Widemann. 2007. An IDE for software development using tabular expressions. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research* (Richmond Hill, Ontario, Canada) *(CASCON '07)*. IBM Corp., USA, 248–251. https://doi.org/10.1145/1321211.1321238

[19] Microsoft Research. 2020. *Z3*. Microsoft Research. https://ericpony.github.io/z3py-tutorial/guide-examples.htm

[20] Microsoft Research. 2020. *Z3*. Microsoft Research. https://github.com/Z3Prover/z3

[21] Neeraj Kumar Singh, Mark Lawford, Thomas S. E. Maibaum, and Alan Wassyng. 2017. *Use of Tabular Expressions for Refinement Automation*. Springer International Publishing, Cham, 167–182.

[22] Alan Wassyng, Mark S. Lawford, and Thomas S.E. Maibaum. 2011. Software certification experience in the canadian nuclear industry: lessons for the future. In *Proceedings of the Ninth ACM International Conference on Embedded Software* (Taipei, Taiwan) *(EMSOFT '11)*. Association for Computing Machinery, New York, NY, USA, 219–226. https://doi.org/10.1145/2038642.2038676