Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 159 (2019) 1431–1438

Procedia Computer Science

www.elsevier.com/locate/procedia

23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

Dunuen: A User-Friendly Formal Verification Tool

Giovanni Capobianco[a], Umberto Di Giacomo[a], Francesco Mercaldo[b,a,*], Antonella Santone[a,*]

[a]*Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy*
[b]*Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy*

Abstract

Formal verification allows checking the design and the behaviour of a system. One of the main limitations to the adoption of formal verification techniques is the process of model creation using specification languages. For this reason a tool supporting this activity is necessary. Actually, there are several tools allowing analysts to verify models expressed into specification languages. These tools provide support for automatically checking whether a system satisfies a property. However, to use such tools it is important to deeply know a precise notation for defining a system, i.e., the Calculus of Communicating Systems. Since systems are often expressed as time-series, to overcome this problem, we provide an user-friendly tool able to automatically generate a system model starting from the CSV - Comma-Separated Values format (the most widespread format considered to release dataset). In this way we hide the details about the model construction form the analyst, which can only focus immediately on the properties to verify.

We introduce *Dunuen*, a tool allowing the user to firstly perform a kind of pre-processing operation starting from a CSV file, as discretization or removing attributes; subsequently it automatically creates a formal model from the pre-processed CSV file and, by invoking the model checker we embedded in *Dunuen*, it finally verifies whether the generated model satisfies a property expressed in temporal logic through a graphic interface, proposing formal methods as an alternative to machine learning for classification tasks.

Keywords: Formal verification, Automatic Tool, Model Checking

1. Introduction and Related Work

Formal methods are mathematical techniques, often supported by tools, for developing software and hardware systems. Mathematical formalism allows the users to analyze and verify these models during the program life-cycle: requirements engineering [19,8], specification, architecture [21,20], design [21,20], implementation, testing, maintenance, and evolution [6].

Formal methods can be considered as "the applied mathematics for modeling and analyzing ICT systems [2]. Their aim is to establish system correctness with mathematical formalism. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex software and hardware systems. Also, formal methods

* Francesco Mercaldo, Antonella Santone
E-mail address: francesco.mercaldo@iit.cnr.it, antonella.santone@unimol.it

1877-0509
10.1016/j.procs.2019.09.313

are one of the highly recommended verification techniques for software development of safety-critical systems[4,7] according to several practices standard, such as International Electrotechnical Commission (IEC) and European Space Agency (ESA). During the last two decades, research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects[15,5].

These techniques are accompanied by powerful software tools that can be used to automate various verification steps[9,10]. Formal methods rely on different formal verification techniques such as model based verification algorithm, e.g. model checking.

Model-based verification techniques are based on models that describe all the possible system states in a mathematically precise and rigorous manner. The system models are accompanied by algorithms that systematically explore all the possible states. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios[13,18,3]. Due to unremitting improvements of underlying algorithms and data structures, together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples are nowadays applicable to realistic designs.

Model checking is one of the most widespread verification techniques. It explores all possible system states in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property.

The general model checking workflow comprises the following steps:

- **Model construction:** Creating an abstract representation of the system;
- **Property specification:** Encoding the formal specification describing the desired/expected system behaviour;
- **Verification:** Automatically verifying the correctness of the model relative to the formal specification.

Several tools in last years were developed for formal verification for instance, *Construction and Analysis of Distributed Processes* (CADP)[11] is a comprehensive software toolbox that implements the results of concurrency theory. Another model checker is NUSMV[1] , designed to be a well structured, open, flexible and documented platform for model checking. PRISM[14] considers probabilistic models i.e., it encodes the probability of making a transition between states. One factor that limits the use of model checking (and of the formal verification tools we cited) to specialists is related to the deep knowledge of formal specification languages, i.e to provide a support tool that automatically accomplish the **Model construction**. To solve this problem, in this paper we propose *Dunuen*, a tool aimed to automatically generate a formal model starting from a comma separate value (CSV) file. We focus on the CSV file as source for the model building, considering that a plethora of problem are described as time-series.

Dunuen basically performs following tasks i.e., pre-processing, discretization and model building.

Furthermore, once generated the formal model of the system under analysis, *Dunuen* allows directly to programmatically invoke a formal environment verification tool (i.e., CWB-NC). A real-world case study showing the user friendly tool interfaces is presented.

The paper proceeds as follows: in the next Section the *Dunuen* tool is presented, in Section 3 a real-world case study is discussed and, finally, last section is related to conclusion and future work.

2. Technique and Implementation

This section briefly describes the *Dunuen* tool in order to explain its main functionalities available through a series of tabs i.e., *Pre-processing*, *Discretization* and *Model Building*.

2.1. Pre-processing

Dunuen provides a set of pre-processing functionalities such as loading CSV file, deleting of specific rows and saving the modified CSV file, as shown in Figure 1.

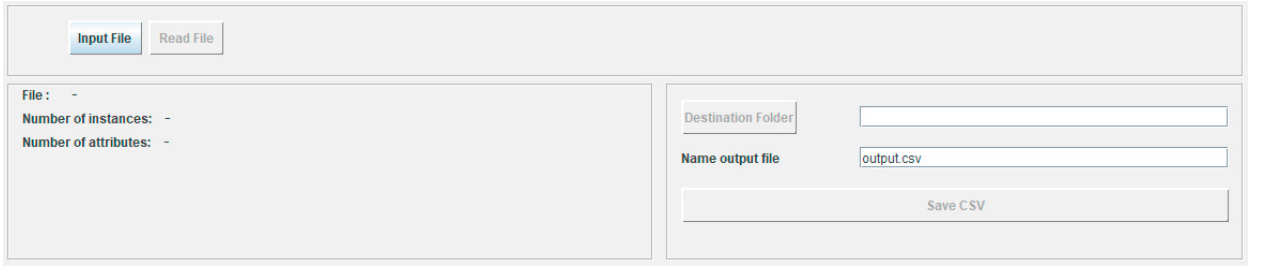

Fig. 1: Tab of *Dunuen* for preprocessing operation.

2.2. Discretization

The second tab of *Dunuen* is about the variable discretization in the loaded CSV, exploiting two different discretization methods: equal frequency or equal width method. The interface allows the user to specify the indices of the attributes to discretize, the number of the bins and the discretization method, as shown in Figure 2.

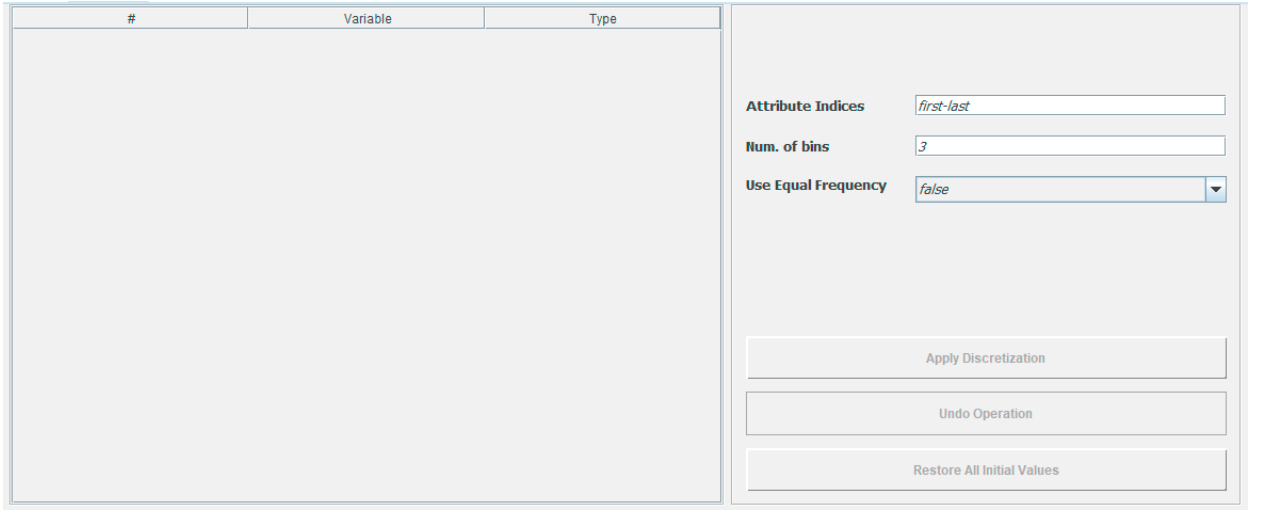

Fig. 2: Tab of *Dunuen* for discretization operation.

2.3. Model Building

The third tab of *Dunuen* is aimed to build the CCS model and it allows the user to:

- **Loading a .CSV file:** in this scenario the tool computes all possible permutation of the variables under analysis to build the CCS model;
- **Loading a .CCS file:** in this scenario the user can load directly a .CCS file representing the model that he/she wants to check;
- **Loading a .mu file:** through this option the user can load a .mu file containing the logical temporal properties to verify on the built CCS model.

We briefly describe the model building construction: for this aim, we use Milner's Calculus of Communicating Systems (CCS)[17]. CCS is one of the most well known process algebras.

time	$F1$	$F2$	$F3$
$t1$	$V3\_F1$	$V1\_F2$	$V2\_F3$
$t2$	$V0\_F1$	$V0\_F2$	$V3\_F3$
$t3$	$V1\_F1$	$V2\_F2$	$V3\_F3$
$t4$	...	...	...

Table 1: Example of CSV file.

P	$\stackrel{\text{def}}{=}$	1.	$V3\_F1.DONE \parallel V1\_F2.DONE \parallel V2\_F3.DONE$;
		2.	$V0\_F1.DONE \parallel V0\_F2.DONE \parallel V3\_F3.DONE$;
		3.	$V1\_F1.DONE \parallel V2\_F2.DONE \parallel V3\_F3.DONE$;
		4.	...;

Table 2: CCS model Fragment.

CCS contains basic operators to build finite processes, communication operators to express concurrency, and some notion of recursion to capture infinite behaviour. The semantics of a CCS process p is formally defined using the structural operational semantics. The semantic definition is given by a set of conditional rules describing the transition relation of the automaton corresponding to the behavior expression defining p. This automaton is called *standard transition system* for p. Readers unfamiliar with CCS are referred to [17] for further details. To the building model we use the following CCS operators:

- "+": the process $p + q$ is a process that non-deterministically behaves either as p or as q.
- ";": this operator is suitably used to express the sequentialization between two processes. The process $p; q$ means that p must terminate before the process q can start its execution.
- "||": the process $p||q$ represents the parallel execution of p and q and terminates only if both processes terminate.
- "*DONE*": is the constant that corresponds to a process whose task is to terminate without further moves.

An example of CSV containing time-series data is shown in Table 1, where the features are *F1*, *F2* and *F3* and we considered four discretization intervals i.e., *V0*, *V1*, *V2* and *V3*.

The CCS model describing the time series is shown in Table 2.

3. Case Study

In this section we present a scenario aimed to show the main functionalities provided by the *Dunuen* tool.

A real-world dataset related to a series of patients with diabetes [12,16] whose measurement of insulin are stored pre and post breakfast, lunch, dinner and bedtime is considered. The dataset is freely available for research purpose[1]. The dataset is available in CSV file related to diabetes information characterized by 63 instances and 3 attributes. In the follow we describe the step-by-step procedure to build the CSS model with *Dunuen* and we show an example of property verification through its user interface.

Firstly, the user have to upload the CSV file and read it.

Once loaded the file, the user performs a discretization operation on the three attributes i.e., *F1*, *F2*, *F3*, using the equal frequency discretization with 4 bins, as shown in Figure 5. The tool allows the user to see the information related to each attribute of the file.

In Figure 4, the user has specified the CSV file, from which the tool has constructed the CCS model, and the mu file containing the property that has to be checked on the CCS model.

Once built the CCS model from the discretized CSV file, the user can evaluate a property. In this case study, we evaluate the property:

$$\varphi = \nu X.[V3\_F2]\,\texttt{ff} \wedge [-V0\_F1]\,X$$

Considering that the *F1* feature is related to *Pre-breakfast blood glucose measurement*, while *F2* is related to the *Regular insulin dose*, the property is aimed to verify that: "no insulin dose has been injected (i.e., *F2* exhibits a high

[1] https://archive.ics.uci.edu/ml/datasets/diabetes

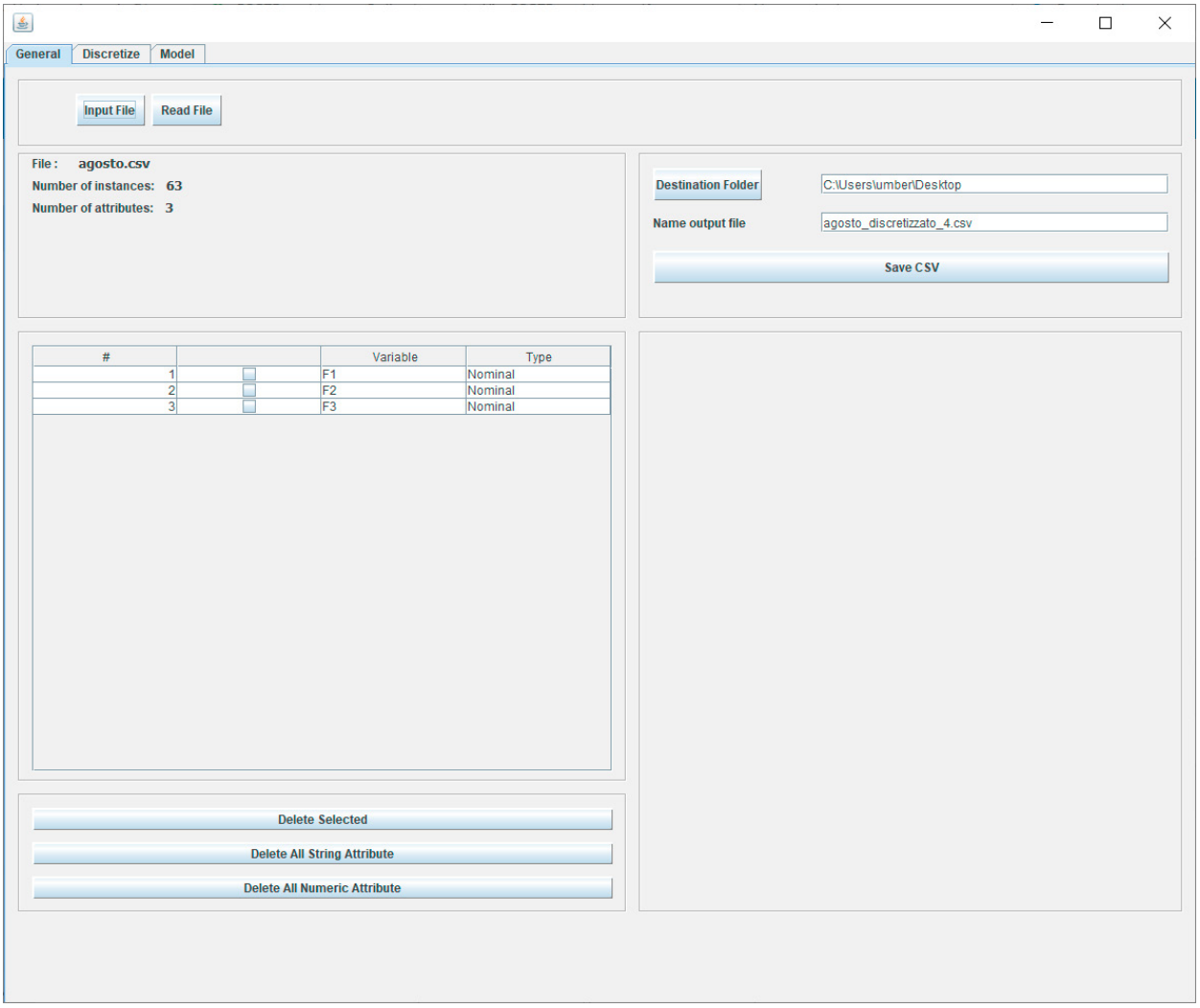

Fig. 3: Example of loading CSV file in *Dunuen*.

value) if the blood glucose pre-breakfast blood glucose measurement is low (i.e., the *F1* feature exhibits a low value)" .

In this case, the property satisfies the model, as a matter of fact the *Dunuen* tool, once verified the property by invoking the CWB-NC, outputs *TRUE*.

4. Conclusion and Future Work

Considering the limitations to the adoption of formal methods due to the deep knowledge of the specification languages, in this paper we propose *Dunuen*, a tool aimed to automatically build CSS models starting from time-series dataset stored in CSV files. *Dunuen* is developed using the Java programming language and it is able to invoke the CWB-NC using the automatically generated CCS model. Furthermore, it presents to the user a practical graphical interface to verify properties. A case study belonging to health-care domain in presented, showing the usability of the *Dunuen* tool. As future work, we plan to extend *Dunuen* to automatically generate CCS models also starting from non time-series dataset. Furthermore, we will investigate whether the built models with a set of properties can be useful to perform classification tasks, as alternative to machine learning based algorithms. From the implementation

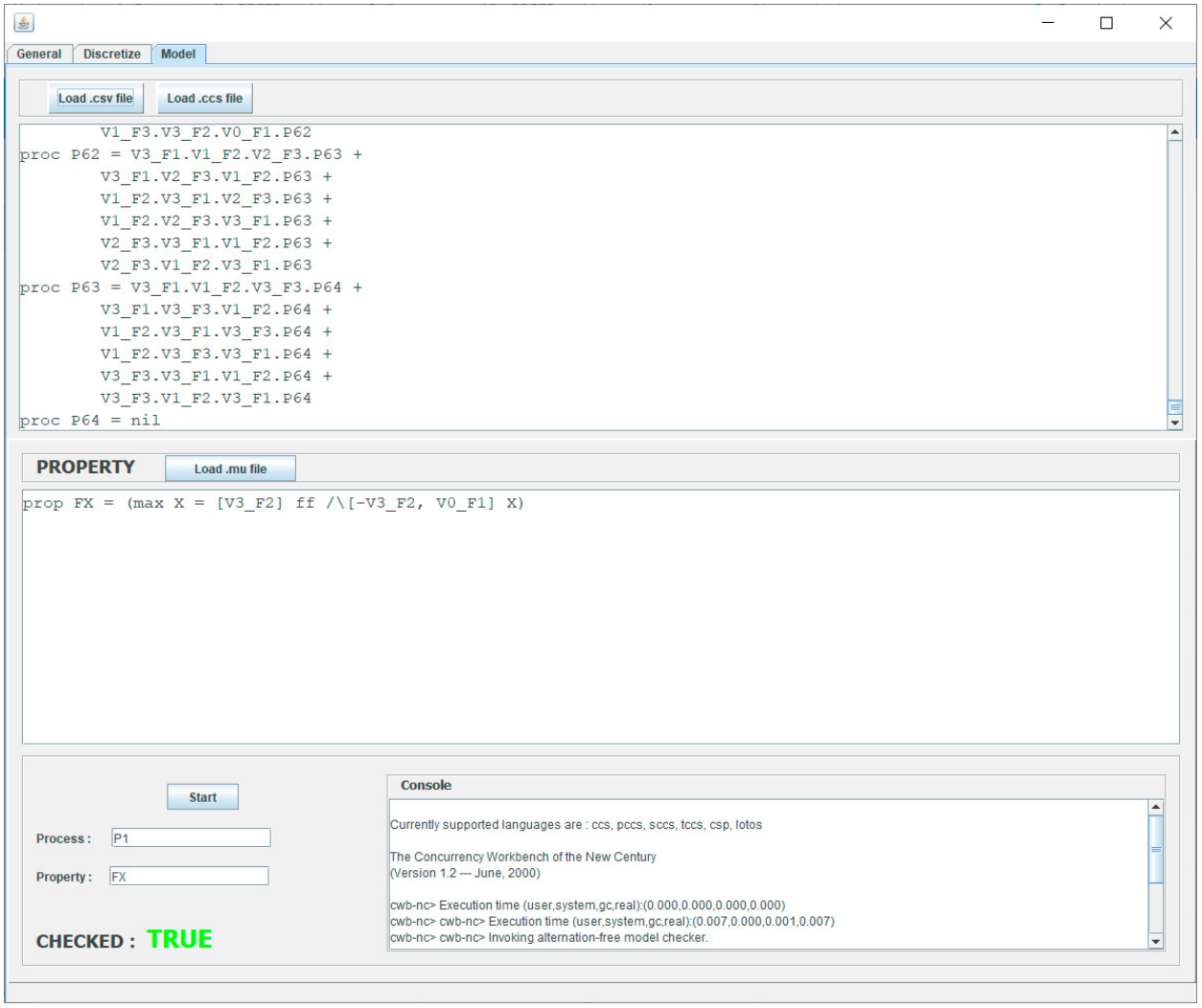

Fig. 4: Example of model construction and property checking in *Dunuen*.

point fo view, in the next *Dunuen* release we will integrate a SQL bridge to make the tool able to accept dataset from databases.

References

1. A. Cimatti, E.C., Roveri, M., 2000. Nusmv: A new symbolic model verifier, pp. 410–425.
2. Baier, C., Katoen, J.P., 2008. Principles of Model Checking.
3. Bowen, J., 1993. Formal methods in safety-critical standards, in: Proceedings 1993 Software Engineering Standards Symposium, IEEE. pp. 168–177.
4. Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2018. Leila: formal tool for identifying mobile malicious behaviour. IEEE Transactions on Software Engineering .
5. Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., 2017a. Formal methods meet mobile code obfuscation identification of code reordering technique, in: 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), IEEE. pp. 263–268.
6. Cimitile, A., Mercaldo, F., Martinelli, F., Nardone, V., Santone, A., Vaglini, G., 2017b. Model checking for mobile android malware evolution, in: Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering, IEEE Press. pp. 24–30.
7. Cimitile, A., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2018. Talos: no more ransomware victims with formal methods. International Journal of Information Security 17, 719–738.

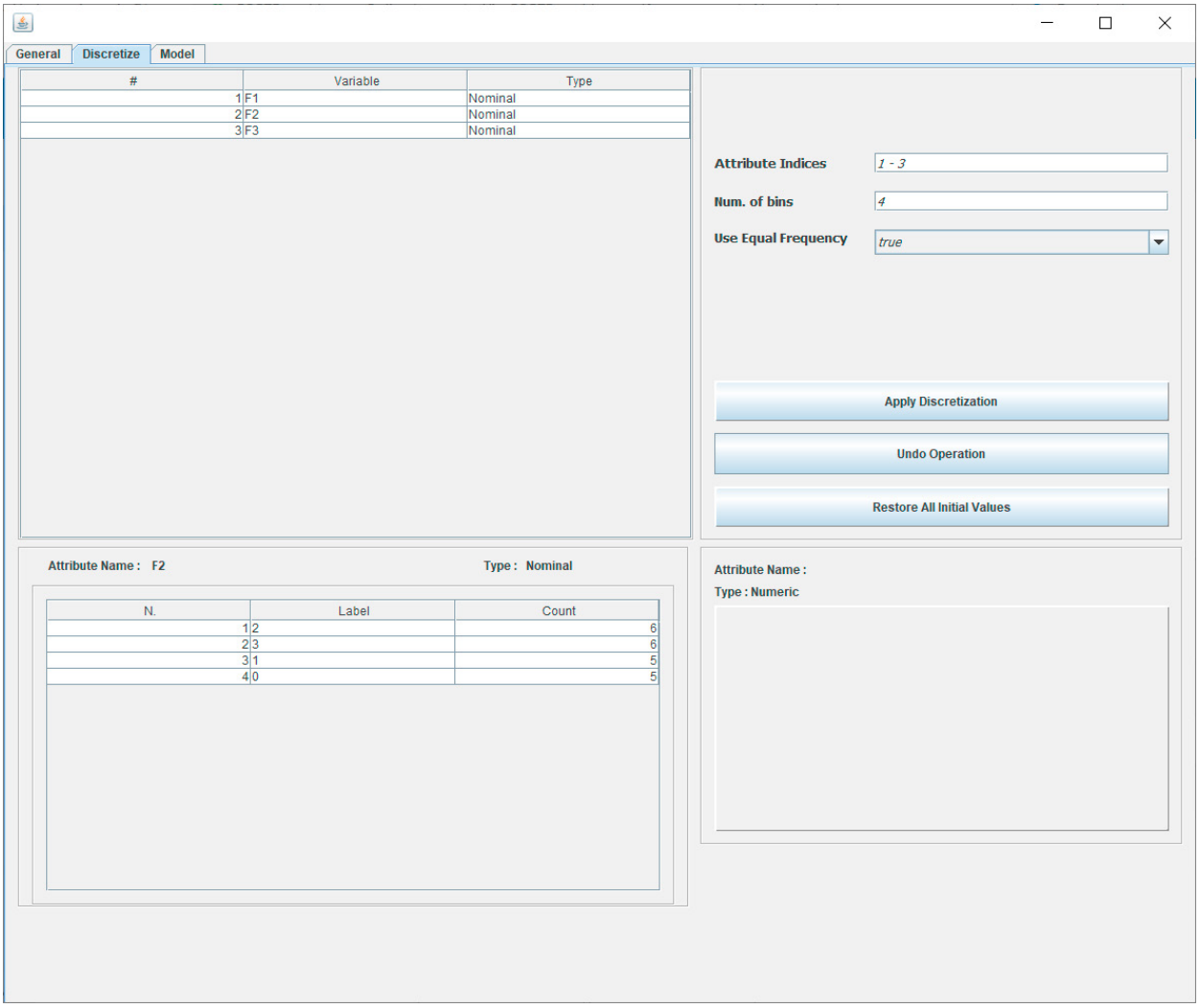

Fig. 5: Example of discretization in *Dunuen*.

8. Gerhart, S., Craigen, D., Ralston, T., 1994. Experience with formal methods in critical systems. IEEE Software 11, 21–28.
9. Heitmeyer, C., 1998. On the need for practical formal methods, in: International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer. pp. 18–26.
10. Holloway, C.M., 1997. Why engineers should consider formal methods, in: 16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings, IEEE. pp. 1–3.
11. Hubert Garavel, Frdric Lang, R.M., Serwe, W., 2012. Cadp 2011: a toolbox for the construction and analysis of distributed processes, p. 89107.
12. Kavakiotis, I., Tsave, O., Salifoglou, A., Maglaveras, N., Vlahavas, I., Chouvarda, I., 2017. Machine learning and data mining methods in diabetes research. Computational and structural biotechnology journal 15, 104–116.
13. Knight, J.C., DeJong, C.L., Gibble, M.S., Nakano, L.G., 1997. Why are formal methods not used more widely?, in: Fourth NASA formal methods workshop, Citeseer.
14. Marta Kwiatkowska, Gethin Norman, D.P., 2002. Prism: Probabilistic symbolic model checker, pp. 200–204.
15. Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., Sangaiah, A.K., Cimitile, A., 2018. Evaluating model checking for cyber threats code obfuscation identification. Journal of Parallel and Distributed Computing 119, 203–218.
16. Mercaldo, F., Nardone, V., Santone, A., 2017. Diabetes mellitus affected patients classification and diagnosis through machine learning techniques. Procedia computer science 112, 2519–2528.
17. Milner, R., 1989. Communication and concurrency. PHI Series in computer science, Prentice Hall.
18. Monin, J.F., 2012. Understanding formal methods. Springer Science & Business Media.
19. Saiedian, H., 1996. An invitation to formal methods. Computer 29, 16–17.

20. Wing, J.M., 1990. A specifier's introduction to formal methods. Computer 23, 8–22.
21. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J., 2009. Formal methods: Practice and experience. ACM computing surveys (CSUR) 41, 19.