Softw Syst Model (2017) 16:625–626
DOI 10.1007/s10270-017-0605-x

EDITORIAL

Logic formulas in models

Jeff Gray[1] · **Bernhard Rumpe**[2]

Published online: 20 June 2017

Many modeling languages, especially graphical ones, concentrate on the ease of expression in specifying certain aspects of a system that need to be developed. However, this often leads to a reduced ability to express complex relations between particular elements in the model. This is certainly obvious when using UML's class diagrams, but also occurs in specification-oriented versions of state machines, activity diagrams, business process models, and variants of architectural description languages. The restricted ability to describe additional constraints usually leads to the demand for an expressive logic that is used on top of the underlying modeling language. The Object Constraint Language (OCL), as part of the UML, is such an example. In the context of class diagrams, it allows the description of invariants in the form of relations between attributes, as well as structural restrictions between objects during runtime.

A logic formula is either true or false. In terms of logic, there is no other possibility. The *Law of the Excluded Middle* (i.e., either a formula is true, or its negation is true—there is no third possible value) is a core axiom in logic and has been accepted by logicians throughout the deep history of development of mathematics and philosophy. The truth values are of such importance that many programming languages have a built-in type that represents these two values; for example, a "Boolean" (going back to George Boole, 1815–1864) type that has values "true" and "false".

Computers, like humans, have restrictions in their abilities to deduce truth. In particular, the theory of computability allows us to understand when a logic formula is true, but the computer does not recognize this when "executing" the formula. Hence, logicians have tried to describe this behavior of computers by introducing denotational semantics as a form of mapping each syntactic construct to the result obtained when the computer executes the formula. In the case of a logic formula, there are three potential values in the notation and semantics: True, false and "undef". This third value, called "undef", is a virtual construct of humans that defines a denotational semantics for languages. "Undef" occurs when the computer does not terminate, when emulating the formula, or terminates in an extraordinary state (e.g., an exception). Therefore, "undef" in its purest sense is not part of the syntax of the language itself.

"Undef" is an interesting construct when reasoning about the behavior of programs. Many verification tools have explicitly added "undef" as an element of any type, because the execution of any expression may fail. Thus, the Boolean type suddenly becomes three-valued. This leads to interesting challenges at the logic level. For example, the traditional logic laws (e.g., commutativity, associativity, and idempotence of "and" and "or") should be retained. The number of cases to be handled in a proof, as well as when intellectually designing a system, should be small; handling of "undef" should not introduce too many exceptions and extra cases to handle. Please note that "and" has four cases in 2-valued logics, but nine cases in 3-valued logics, which is more than twice the potential values in a 2-valued logic.

Different solutions have been proposed (e.g., extending logics to three values, like in Kleene logic, or strictly separating the Boolean type to 3 elements and the Truth values to 2 elements). All of the solutions have individual advantages and disadvantages, sometimes very much dependent on the purpose and use of the formula, such as: when a formula is used mainly for programming, when a formula is for specifying in an abstract way and then, for example, deriving tests

✉ Bernhard Rumpe
bernhard.rumpe@sosym.org

[1] University of Alabama, Tuscaloosa, AL, USA

[2] RWTH Aachen University, Aachen, Germany

from a specification, or for verifying certain properties in critical parts of a system.

"Undef" is also an interesting construct when modeling the desired behavior from a safety or security point of view. For example, robustness to some extent explicitly refers to the definedness of functionalities. This led to a need to add "undef" as an explicit value to a modeling language, but also to certain programming languages. The very famous "null" in many programming languages is such a construction. The logic of OCL also provides explicit mechanisms to describe the absence of a truth value by "null" and invalidity of a formula by "invalid". This allows the convenient expression of certain effects, for example when retrieving data from a database, where "null" describes missing data. Thus, it can be seen as a valid mechanism to extend forms of types to these values. However, from the denotational semantics point of view, these are just ordinary real values and again the virtual value "undef" is necessary to describe nonterminating execution. The OCL Boolean type with all of these options has roughly five values ("true", "false", "invalid", "null" and of course the virtual "undef").

It is necessary to clearly separate a Boolean type, which is part of the underlying programming language and thus of the system under development, from the Truth values. The Truth values are not part of the system under development and need not be used to type attributes or variables, but describe the logical outcome of the meaning of a formula. Tertium non datur.

Thus we postulate:

> The truth shall not be compromised by alternate truth and false values.

When we use a logic formula during the execution of a test, then we are confronted again with the restricted ability to deduce truth computationally (especially in non-terminating evaluations). However, we believe that in a modern language, such as Java, where many potential sources for non-termination are eliminated, this is not a practical problem anymore. For example, infinite loops are easy to detect by humans and infinite recursion may terminate eventually with a StackOverflow exception. If the execution of a logic formula terminates with an exception, the formula is simply false.

Thus, we think it is necessary for models, but first of all for designers of modeling languages including logics, to explicitly separate the truth values from the Boolean type. It also does not make sense to try to inject everything into computational types, because it complicates the logic formulas to be defined beyond the intrinsic necessity.

Content of this issue

This issue contains the theme section of *Business Process Modeling, Development, and Support* (BPMDS'2014): "The Human Perspective in Business Processes", with Selmin Nurcan and Rainer Schmidt as Guest Editors. The included papers are described in the Guest Editorial.

In addition, this issue contains the following regular papers:

- "A case study about the improvement of business process models driven by indicators" by Felix Garcia, Laura Sanchez-Gonzalez, Francisco Ruiz, and Mario Piattini.
- "Refinement-based Validation of Event-B Specifications" by Atif Mashkoor, Faqing Yang, and Jean-Pierre Jacquot.
- "Design notations for secure software: a systematic literature review" by Alexander van den Berghe, Riccardo Scandariato, Koen Yskout, and Wouter Joosen.
- "A graph-theoretic method for the inductive development of reference process models" by Jana Rehse, Peter Fettke, and Peter Loos.
- "An integrated semantics for reasoning about SysML design models using refinement" by Lucas Lima, Alvaro Miyazawa, Ana Cavalcanti, Márcio Cornélio, Juliano Iyoda, Augusto Sampaio, Ralph Hains, Adrian Larkham, and Vaughan Lewis.
- "Supporting aspect orientation in business process management—From process modelling to process enactment" by Amin Jalali, Amin, Chun Ouyang, Petia Wohed, and Paul Johannesson.