Usability of AutoProof: a case study of software verification

Mansur Khazeev <m.khazeev@innopolis.ru>
Victor Rivera <v.rivera@innopolis.ru>
Manuel Mazzara <m.mazzara@innopolis.ru>
Alexander Tchitchigin <a.chichigin@innopolis.ru>
Innopolis University, Software Engineering Lab.
420500, Russia, Innopolis, Universitetskaya Str. 1

Abstract. Verification tools are often the result of several years of research effort. The development happens as a distributed effort inside academic institutes relying on the ability of senior investigators to ensure continuity. Quality attributes such as usability are unlikely to be targeted with the same accuracy required for commercial software where those factors make a financial difference. In order for such tools to become of widespread use, it is therefore necessary to spend an extra effort and attention on users' experience, and allow software engineers to benefit out of them without the necessity of understanding the mathematical machinery in full detail. In order to put the spotlight on usability of verification tools we chose an automated verifier for the Eiffel programming language, AutoProof, and a well-known benchmark, the Tokeneer problem. The tool is used to verify parts of the implementation of the Tokeneer so to identify AutoProof's strengths and weaknesses, and finally propose the necessary updates. The results show the efficacy of the tool in verifying a real piece of software and automatically discharging nearly two thirds of verification conditions. At the same time, the case study shows the demand for improved documentation and emphasizes the need for improvement in the tool itself and in the Eiffel IDE.

Keywords: static verification; formal specification; Eiffel, Autoproof; Design by Contract

DOI: 10.15514/ISPRAS-2016-28(2)-7

For citation: Khazeev Mansur, Rivera Victor, Mazzara Manuel, Tchitchigin Alexander. Usability of AutoProof: a case study of software verification. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 111-126. DOI: 10.15514/ISPRAS-2016-28(2)-7

1. Introduction

Tools for software verification allow the application of theoretical principles in practice, in order to ensure that nothing bad will ever happen (safety). The extra effort required by the use of these tools is certainly not for free and comes with increased development costs [1]. There is a common belief in industry that developing software

with high level of assurance is too expensive, therefore not acceptable, especially for non safety-critical or financially-critical applications.

Tools and techniques for the formal development of software have played a key role on demystifying this belief. There are several approaches, for instances abstract interpretation and model checking [2], [3] that seek the automation to formally proving certain conditions of systems. However, these techniques tend to verify simple properties only. On the other end of the spectrum, there are interactive techniques for verification such theorem provers [4]. These techniques aim at more complex properties but demand the interaction of users to help the verification.

Nowadays, there are new approaches that aim at finding a good trade-off between both techniques, e.g. auto-active: users are not needed during the verification process (it is automatically performed); they are required instead to provide guidance to the proof using annotations. AutoProof [5], is a static auto-active verifier for functional properties of object-oriented programs. Using AutoProof, users write code and equip classes with contracts and annotations to help the tool to prove certain properties.

The main goal resented in this paper is to provide insights on how easy/difficult is for users (mainly engineers without deep knowledge of formal verification) to use current methodologies and tools for the development of software with high level of assurance, in particular on the use of the AutoProof tool.

Generally, to prove the correctness of a program one needs some mechanisms to express what the program is supposed to do and clearly state it in the specifications that are used later to verify the program. Eiffel programming language natively supports these mechanisms by means of contracts. Eiffel is an object-oriented programming language, which directly implements the concepts of Design-by-Contract (DbC) [1], [6]. The key concept is viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations. This is realized equipping methods with pre- and post-conditions, and classes with invariants. The key feature of the Eiffel language is indeed the idea that all the methods might and should contain contracts.

Contracts and annotations used in Eiffel are used by AutoProof to statically verify the consistency of the classes. To demonstrate the usability of the tool, the Tokeneer project [7] was implemented in Eiffel and AutoProof was used to verify the consistency of the code. The Tokeneer project is a system specified and implemented by National Security Agency (NSA). Initially, NSA carried out this challenge to prove that it is possible to develop secure systems rigorously in a cost effective manner. Since its development, it became a testing range for different software development methodologies and verification tools. Results of the project are publicly available. This paper reports on the use of AutoProof to verify an Eiffel implementation of Tokeneer and also reports on how easy/difficult is for users to use the tool, e.g. the burden of helping the tool by means of annotations in the code.

The rest of the paper is organized as follows: Section II introduces the Tokeneer project, Eiffel and the AutoProof tool. Section III describes the methodology used to verify the implementation of the Tokeneer project. Section IV presents empirical

results helping to draw conclusions. Section V is devoted to related work and Section VI concludes and mentions future work.

2. Preliminaries

2.1 The Tokeneer Project

In 2002, with the aim to prove/disprove the common believe in industry that development of software of high level of assurance is too expensive and therefore not feasible, the National Security Agency (NSA) asked Altran to undertake a research project to develop part of an existing secure system, the Tokeneer System, in accordance with Altran's Correctness by Construction development process. The system was specified using Z notation [8] and implemented in Ada [9]. The project was successfully delivered in 2003 within 260 days of effort, and later, in 2008, all the results were made available by NSA to the software development and security communities in order to demonstrate the possibility to develop secure systems in a cost effective manner. It includes the ``Core'' Tokeneer ID System Software, test cases derived from the system test specification, ``Support'' Tokeneer ID System Software and test tokens and biometric data, project documents. Since the delivery, the Tokeneer project has become a milestone point and a testing range for different verification tools before applying them in industrial projects. Despite the fact that after delivery 4 bugs[1] were found, the system is still deemed to be very secure.
Tokeneer is a secure enclave consisting of a set of system components, some housed inside the enclave and some outside, as depicted in Figure 1.

[1] According to [7]

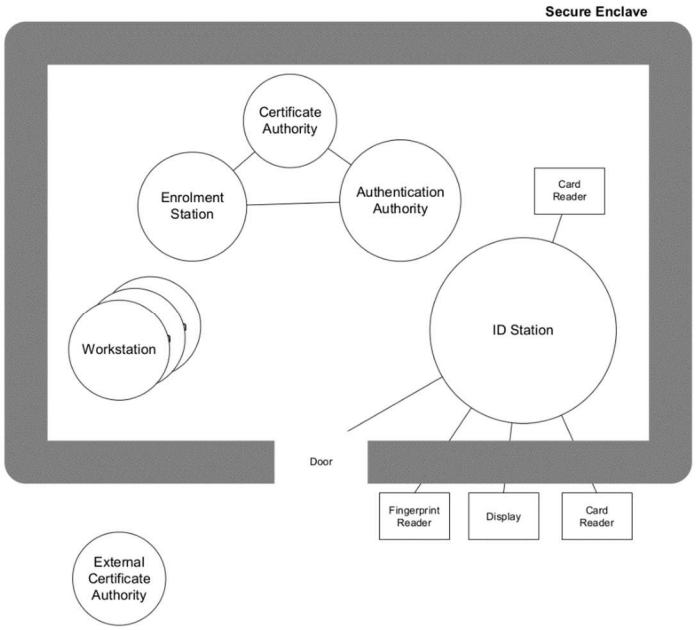

Fig. 1. The Tokeneer System.

The ID Station (TIS) is part of the larger Tokeneer system. It has four connected peripherals, namely, a fingerprint reader, a smartcard reader (users use Tokens - smartcards- as identification), a door and visual display. The objective of the enclave is to ensure that anyone who enters the enclave has a proper access, and no one else can access to the enclave.
In order to ensure the entrance of users to the enclave, TIS implements a series of protocols and checks (the use of smart cards and biometrics) to grant or deny the entrance to it. This paper discusses one of these protocols: the enrollment to the ID Station. The protocol starts in a state where the user is not enrolled. Users can request enrollment and then insert a **FLOPPY** (it retains an internal view of the last data written) for the system to proceed. The system reads the information in the floppy and either fails the enrollment process, in which case takes the process to the initial state, or correctly validates the data in the floppy.

2.2 Eiffel

Eiffel is a real complex object oriented programming language that natively supports Design-by-Contract methodology. Users can specify the behavior of Eiffel classes by equipping them with contracts: pre- and post-conditions and class invariants that are represented as assertions.

```
class
    ACCOUNT
create make

feature      -- Initialization
    make     -- Initialize empty account
        do
            balance := 0
        ensure
            balance_set: balance = 0
        end

feature      -- Access
    balance : INTEGER   -- Balance of account

feature      -- Element change
    deposit (amount : INTEGER)  -- Deposit 'amount' on account
        require
            amount not negative : amount >= 0
        do
            balance := balance + amount
        ensure
            balance_increased : balance = old balance + amount
        end

    withdraw (amount : INTEGER) --Withdraw 'amount' from account
        require
            enough_balance : amount <= balance
        do
            balance := balance - amount
        ensure
            balance_decreased : balance = old balance - amount
        end

invariant
    non_negative_balance : balance >= 0
end
```

Fig.2 ACCOUNT Eiffel class.

Figure 2 depicts a reduced implementation of a Bank Account. In Eiffel, creation procedures are listed under the keyword create, for class **ACCOUNT**, routine make is used as a creation procedure. The class defines a class attribute balance to represent the current balance of the account. It also defines two routines (methods), deposit and withdraw. deposit implements a deposit of amount of money to the account and withdraw implements withdrawing money. Eiffel encourages software developers to express formal properties of classes by writing assertions. Routine pre-conditions express the requirements that clients must satisfy whenever they call a routine. They are introduced in Eiffel by the keyword require. Routine deposit imposes a pre-condition on the call, the client must pass as an argument a non-negative number (i.e. **amount_not_negative**: **amount >= 0**) for the routine to work correctly: a negative value might invalidate the invariant of the class. Routine post-conditions, introduced in Eiffel by the keyword ensure, express conditions that the routine (the supplier) guarantees on method exit, assuming the pre-condition. Routine deposit guarantees that the balance of the account will be the previous value of the balance (expressed in Eiffel by the keyword old: the value on entrance of the routine) plus the amount being deposited. Routine withdraw imposes the constraint to the caller that the argument must be less than or equal to the current balance of the account to avoid having

negative value in the balance. The routine ensures that, after execution, the new value of balance will be the value on routine entry minus the amount withdrawn.

A class invariant must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class (public routines). The invariant appears in a clause introduced by the keyword invariant. Class **ACCOUNT**'s invariant imposes the restriction that class attribute balance can never be negative (i.e. **non_negative_balance: balance >= 0**).

2.3 AutoProof

AutoProof [5] is a static verifier of contracts for Eiffel programs. It follows the auto-active paradigm where verification is done completely automated, similar to model checking [3], but users are expected to feed the classes providing additional information in the form of annotations to help the proof. AutoProof identifies software issues without the need of executing the code, therefore opening a new frontier for ``static debugging'', software verification and reliability, and in general for software quality.

AutoProof verifies the functional correctness of Eiffel classes. It translates Eiffel code to Boogie programs [10] and calls the Boogie tool to generate verification conditions: logic formulas whose validity entails correctness of the input programs. Finally, retrieves the answer back to Eiffel. AutoProof verifies that routines satisfy pre- and post-conditions, maintenance of class invariants, loops and recursive calls termination, integer overflow and non-Void (*null* in other programming languages) references calls. The tool also supports most of the Eiffel language constructs: in-lined assertions such as check (*assert* in other programming languages), types, multi-inheritance, polymorphism.

3. Verification of Tokeneer using AutoProof

The Tokeneer project was implemented in Eiffel following the specifications file 41_2.pdf (see [7]) of the Tokeneer System and equipping classes with contracts. This research work encompasses only the enrolment process of the whole Tokeneer System therefore it implements only the entities involved in this process.

One of the main parts of TIS is the **ID_STATION** (see Figure 8) – it describes how all components of the system are related to each other: one of the components is implemented in class **INTERNAL_**S (not shown here) whose responsibility is to keep knowledge of the status of user entry and the enclave and to hold a timeout when relevant; another component is implemented on class **FLOPPY** (not shown here) that retains an internal view of the last data written to the floppy as well as the current data on the floppy. **ID_STATION** displays the configuration data on the screen which is implemented in **SCREEN_DISPLAY**. There are a number of messages that may appear on the TIS screen. The Real World types (described in [7] Specification document, section 2.7.1) of the system such as messages that appear on the display and screen, were implemented all together in class CONST which implements the

constants used in the TIS. And finally, a number of interactions between all these entities within the enclave are implemented in **ENCLAVE_OPERS**.
AutoProof does not make any assumptions out of box therefore users are expected to feed the Eiffel classes for a succeed verification.

```
class
    ID_STATION
    -- Some lines were omitted--

create
    make

feature --Initialization
    make
        note
            status : creator
        do
            -- Some lines were omitted --
        end
```

Fig. 3. Initialization of ID STATION Eiffel class.

This is expressed by means of Eiffel's **note** clause. **note** clause enables users to attach addition information to the class that is ignored by the Eiffel's compiler. AutoProof uses this information to succeed in the verification. For instance, AutoProof's annotation **status** defines which procedure is used to initialize newly created objects: Figure 3 depicts procedure make with annotation **note** (e.g. **note** status: creator) to help Autoproof to discharge the corresponding proof obligations related to creation procedures: the procedure will be called only when an object of this class is being created, AutoProof needs to verify a creation routine only once.
note clause is also used to define models queries to express the abstract state space of a classes. Model queries are part of model-based contracts to help users to write abstract and concise specifications [11], they are used to specify the behavior of the class. In Eiffel, this is specified by adding a **note** clause at the beginning of the class followed with a keyword model: and listing one or more attributes of the class. Model queries are also used to describe frame conditions: which allocations are allowed to be modified by procedures.
In Eiffel, frame conditions are listed using the **modify** clause, which lists the model queries that the feature is allowed to modify, as shown in Figure 7 (i.e. **modify_model("current_display", Current)**).

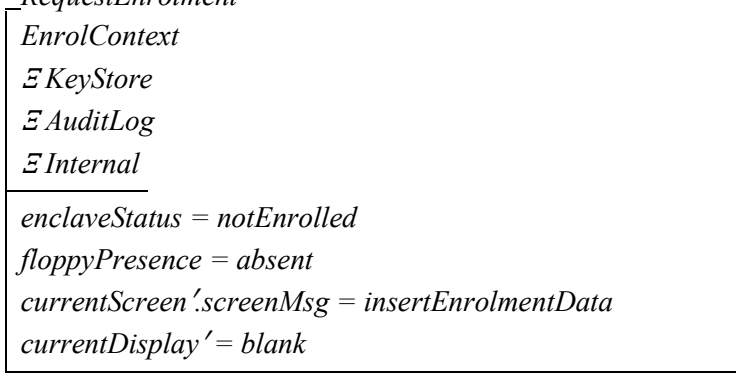

*Fig. 4. Z schema of **RequestEnrolment.***

According to **RequestEnrolment** (a Z-schema that is a part of the formal specification of the project Tokeneer), which is presented in Figure 4, requesting enrolment involves **EnrolContext**, **KeyStore**, **AuditLog**, Internal. Schemas in Z consist of an upper part, in which some variables are declared, and a lower part, which describes the relationship between values and variables. The notation **Ξ** indicates an operation in which the state does not change, and the apostrophe indicates the state of the variable after the change [12]. **RequestEnrolment** specifies that the ID station will request enrolment by displaying a request string on the screen and keeping the display blank. This is only possible while there is no Floppy present. Therefore, initially **floppyPresence = absent** and **enclaveStatus** set to **notEnrolled**. An ensure clause was used in the creation procedure to guarantee this after the initialization of **ID_STATION** object:

```
make
        -- Some lines were omitted --
    ensure
        enclave status = cons floppy.not enrolled
        floppy presence = cons internal.absent
        token removal timeout = 0
    end
```

Fig. 5. ensure clause in feature make.

Figure 6 depicts the class invariant for class **ID_STATION**. It states that a message displayed on the display outside the enclave is one of the available from the list of messages (i.e. **constants.display_message.has(current_display)**) and that class attribute constants is attached to an object (i.e. **constants /= Void**).

```
invariant
    constants.display message.has(current display)
    constants /= Void
```

Fig. 6. Invariants of ID STATION Eiffel class.

Figure 7 shows the implementation of procedure **set_current_display**. Its first pre-condition was added to satisfy the invariant ensuring that argument *v* belongs to the allowed displayed messages. The second pre-condition restricts the procedure to change values only to model query **current_display**.

```
feature     -- Element Change
    set_current_display (v: STRING)
        require
            constants.display message.has(v)
            modify model("current display", Current)
        do
            current display := v
        ensure
            current display = v
        end
```

Fig. 7. Feature equipped with modify clause.

Figure 8 shows the final version of class **ID_STATION**: with the respective annotations for AutoProof to successfully verify the class. In class **ID_STATION**, class attributes **current_screen** and **current_display** implements the physical screen and display, respectively, of the enclave.

```
class
    ID STATION
    -- Some lines were omitted --
create
    make

feature     -- Initialization
    make
        note
            status: creator
        do
            create constants
            current display := constants.blank
            create current screen.make

            create cons floppy
            enclave status := cons floppy.not enrolled
            token removal timeout := 0
```



```
            create cons internal
            floppy presence := cons internal.absent
        ensure
            enclave status = cons floppy.not enrolled
            floppy presence = cons internal.absent
            token removal timeout = 0
        end

feature     -- Element Change
    set_current_display(v: STRING)
        require
            constants.display_message.has(v)
            modify model("current display", Current)
        do
            current display := v
        ensure
            current display = v
        end

feature     -- Access
    constants : CONST
    current screen : SCREEN DISPLAY
    current display : STRING

invariant
    constants.display_message.has(current display)
    constants /= Void
end
```

Fig. 8. Verified ID STATION Eiffel class.

4. Empirical Results

The usability of a verification tool cannot be considered in isolation and, in particular, cannot be hived off by the effectiveness of the tool itself. First, as a general observation, the cost of using an instrument can only be justified by its return, which can ultimately be linked to financial consideration by top management. Second, and this aspect is less general and more peculiar to the auto-active verification approach, a tool like AutoProof is as much effective and usable as is its ability to discharge verification conditions completely automatically, without feeding the code of annotation overhead or requiring particular tweaking. Finally, the necessity for users to add further annotations and dedicate extra effort (and considerable time) is, by itself, an obstacle to adoption and (technically) a usability issue. Verification tools should require minimal annotational effort and give valuable feedback when verification fails.

The case study analyzed in this paper presented good results in term of automatic discharge of verification conditions, though not comparable to others seen in literature [13].

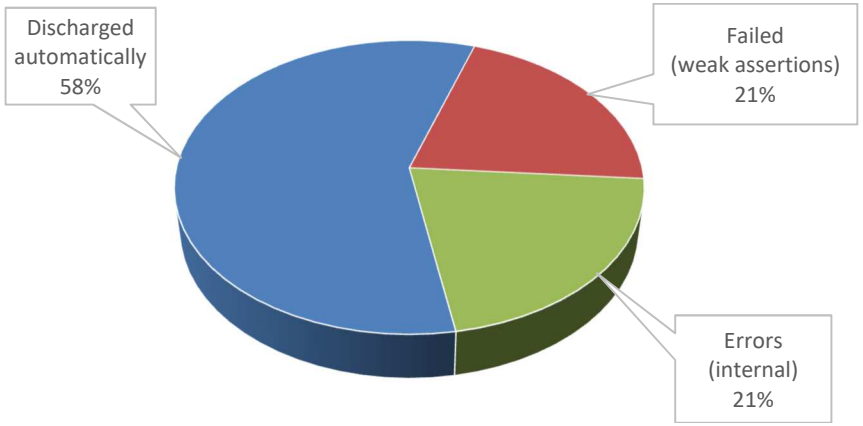

Fig. 9. Verifications results.

In total there were 38 generated proofs. Of these, 22 (58\%) were discharged automatically (see figure 9), 8 (21\%) could not be satisfied, and the rest (21\%) failed due to internal errors, which in our case were basically caused by the attempt to create objects in the contract, and that is not allowed by the tool. As observed before, the success of verification is unsurprisingly linked to the complexity of programs [13]. Previous literature mostly dealt with students users and university projects. The use of Tokeneer as a benchmark demands for detailed comparisons with different verification efforts (for example, [14]).

5. Related Work

Formal/mathematical notations have existed for a long time and have been used to specify and verify systems. Examples are process algebras [15], specification languages like Z [16], B [17] and Event-B [18]. The Vienna Development Method (VDM) is one of the earliest attempts to establish a formal method for the development of computer systems [19]. A survey of these (and others) formalisms can be found in [20] while a discussion on the methodological issues of a number of formal methods is presented in [21].

All these approaches (and others described in the literature) still leave an open issue, i.e., they are built around strict formal notations which affect the development process from the very beginning. These approaches demonstrate a low level of flexibility. To overcome this problem, a seamless methodological connection built on top of a portfolio of diverse notations and methods is presented in [22]. Another approach is presented in [14], [23] using [24], where users start the development of system from a strict formal notation (i.e. Event-B), to then automatically translate it to Java code with JML [25] specifications embedded (following Design-by-Contract methodology). Even though this approach enables users with less mathematical

expertise to work on formal development, it does not give a seamlessly methodology for the development as presented in this paper.

On the other side, Design-by-contract [6] when combined with AutoProof technology offers the pros of both rigorous methodologies and supporting tools able to semi-automate the process. Before this to be available for the average developer it is however necessary to improve the users' experience. A comparison between different approaches (for example Event-b/Rodin and Design-by-contract/AutoProof) is beyond the scope of this paper and it is left as future work.

6. Conclusion

AutoProof allows for “static debugging”, i.e. debugging becomes possible without the need of executing the program. The most effective way to release correct software is a combination of static debugging and traditional run-time debugging. Being all human activities (therefore including programming and testing itself) error-prone, there is no magic or free lunches out there. Abandoning testing and adopting a proof-oriented approach does not make miracles, debugging remains a trial-and-error long and laborious process. AutoProof does not change the rules of the game: developers will have to try, observe the results and make changes as a consequence. A proof-oriented approach does not make the process smoother and necessarily simpler. However, it makes it more accurate and robust, therefore effective. Adjustment can be now focused on the implementation side (possibly sinergically with run-time debugging), on the specification side (the contracts used to annotate the code as integral part of the code itself), or in the proof itself (fine-tuning may be necessary for AutoProof and its behind-the-curtains machinery to be able to prove correctly).

All this comes with a cost: the willingness and ability of the user to use extra tools and being able to master them, and possibly invest extra time in the process. On the other side, it is necessary for the tools to be simple to master and to provide intelligible feedback.

The Tokeneer project case study showed the efficacy of AutoProof in verifying a real piece of software, the complexity of which can be compared not only with most of the commercial Off-the-Shelf software, but also with safety and financial-critical applications, both in terms of computational logic and architectural organization. AutoProof is capable to verify industrial software and may well be adopted in commercial companies and its use injected into the development process. However, some obstacles have been identified that could prevent its broader adoption.

As result of an academic effort, documentation is not at par with commercial software, in particular for what concerns the size of the library of correctly verified examples: tutorials on the official website are quite useful, but not enough. On top of this, the tool itself has limitations. First, existing implementations need to be modified in order to be verified. This would represent an unsurmountable obstacle in most institutions since the overall cost of code adaptation may overrun the saves occurring to the testing phase. This consideration may be different, however, for safety-critical

systems. Second, the Eiffel IDE - necessary for functioning - calls for increased stability and robustness.

7. Acknowledgments

We would like to thank Innopolis University for logistic and financial support, and the laboratories of Software Engineering (SE) and Service Science and Engineering (SSE) for the intellectual engagement and vivid discussions.

References

[1]. B. Meyer, Touch of Class: Learning to Program Well with Objects and Contracts. Springer Publishing Company, Incorporated, 1 ed., 2009.
[2]. P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL ’77, (New York, NY, USA), pp. 238–252, ACM, 1977.
[3]. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, Model Checking. Cambridge, MA, USA: MIT Press, 1999.
[4]. D. W. Loveland, Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science). sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.
[5]. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, “AutoProof: Auto-active functional verification of object-oriented programs,” in 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings, pp. 566–580, 2015.
[6]. B. Meyer, Object-oriented software construction, ch. 11: Design by Contract: building reliable software. Prentice Hall PTR, 1997.
[7]. AdaCore, “Tokeneer.” http://www.adacore.com/sparkpro/tokeneer/download, accessed in April 2016.
[8]. J.-R. Abrial, S. Schuman, and B. Meyer, “Specification Language,” in On the Construction of Programs, R. M. McKeag and A. M. Macnaghten, editors, pp. 343–410, Cambridge University Press, 1980.
[9]. J. Barnes, High Integrity Software: The SPARK Approach to Safety and Security. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
[10]. K. R. M. Leino, “This is boogie 2,” tech. rep., June 2008.
[11]. N. Polikarpova, C. A. Furia, and B. Meyer, “Specifying reusable components,” in Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE’10) (G. T. Leavens, P. O’Hearn, and S. Rajamani, eds.), vol. 6217 of Lecture Notes in Computer Science, pp. 127–141, Springer, August 2010.
[12]. J. Spivey, “An introduction to Z and formal specifications,” Software Engineering Journal, 1989.
[13]. C. A. Furia, C. M. Poskitt, and J. Tschannen, “The AutoProof verifier: Usability by non-experts and on standard code,” in Proc. Formal Integrated Development Environment (F-IDE 2015), vol. 187, pp. 42–55, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.
[14]. V. Rivera, S. Bhattacharya, and N. Cataˇ no, “Undertaking the tokeneer challenge in Event-B,” To appear in 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2016.

[15]. J. C. M. Baeten, “A brief history of process algebra,” Theor. Comput. Sci., vol. 335, no. 2-3, pp. 131–146, 2005.
[16]. J. Abrial, S. A. Schuman, and B. Meyer, “Specification language,” in On the Construction of Programs, pp. 343–410, 1980.
[17]. J. Abrial, The B-book - assigning programs to meanings. Cambridge University Press, 2005.
[18]. J.-R. Abrial, Modeling in Event-B: System and Software Engineering. New York, NY, USA: Cambridge University Press, 1st ed., 2010.
[19]. C. B. Jones, Software Development: A Rigorous Approach. Englewood Cliffs, N.J., USA: Prentice Hall International, 1980.
[20]. “On modelling and analysis of dynamic reconfiguration of dependable real-time systems,” in Proceedings of the 2010 Third International Conference on Dependability, DEPEND ’10, (Washington, DC, USA), pp. 173–181, IEEE Computer Society, 2010.
[21]. M. Mazzara, “Deriving specifications of dependable systems: toward a method,” in Proceedings of the 12th European Workshop on Dependable Computing, EWDC, 2009.
[22]. R. Gmehlich, K. Grau, A. Iliasov, M. Jackson, F. Loesch, and M. Mazzara, “Towards a formalism-based toolkit for automotive applications,” 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2013.
[23]. V. Rivera, N. Cataˇ no, T. Wahls, and C. Rueda, “Code generation for Event-B.” To appear in International Journal on STTT, 2016.
[24]. V. Rivera and N. Cataˇ no, “Translating Event-B to JML-Specified Java programs,” in 29th ACM SAC, (Gyeongju, South Korea), March 24-28, 2014.
[25]. G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of jml: A behavioral interface specification language for java,” SIGSOFT Softw. Eng. Notes, vol. 31, pp. 1–38, May 2006.

Применимость AutoProof: учебный пример верификации ПО

Мансур Хазеев <m.khazeev@innopolis.ru>
Виктор Ривера <v.rivera@innopolis.ru>
Мануэль Маццара <m.mazzara@innopolis.ru>
Александр Чичигин <a.chichigin@innopolis.ru>
Университет Иннополис,
420500, Россия, респ. Татарстан, г. Иннополис, ул. Университетская, д.1.

Аннотация. Очень часто инструменты статической верификации являются результатом многолетних научно-исследовательских работ. По этой причине разработки ведутся с распределением задач внутри учебных заведений и с расчетом на способность старших исследователей обеспечивать её непрерывность. В такой ситуации некоторые атрибуты качества, такие как удобство и простота использования программного обеспечения, чаще всего, не рассматриваются на должном уровне, что плохо сказывается на возможности дальнейшей коммерциализации продукта. Для того, чтобы данные инструменты получили широкое применение необходимо обратить внимание и направить усилия при дальнейшей доработке на упрощение механизма взаимодействия пользователей с приложением, для того, чтобы дать инженерам программного

обеспечения возможность пользоваться инструментом без необходимости полного понимания всех математических механизмов во всех деталях. Для того, чтобы обратить внимание общественности на важность удобства использования инструментов верификации, мы применили инструмент AutoProof к хорошо известному проекту Tokeneer. Данный инструмент использовался для верификации части имплементации реального проекта Tokeneer, в ходе чего были выявлены сильные и слабые стороны AutoProof, и, как результат, был составлен список необходимых улучшений. Результат данной работы иллюстрирует эффективность инструмента при верификации фрагмента реального программного обеспечения: он позволил автоматически проверить практически две трети всех свойств. В то же время, данное исследование показало потребность в доработке документации к данному инструменту и подчеркнуло необходимость улучшения как самого инструмента, так и среды Eiffel IDE.

Ключевые слова: статическая верификация, формальная спецификация, Eiffel, Autoproof, контрактное программирование

DOI: 10.15514/ISPRAS-2016-28(2)-7

Для цитирования: Хазеев Мансур, Ривера Виктор, Маццара Мануэль, Чичигин Александр. Применимость AutoProof: учебный пример верификации ПО. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 111-126 (на английском). DOI: 10.15514/ISPRAS-2016-28(2)-7

Список литературы

[1]. B. Meyer, Touch of Class: Learning to Program Well with Objects and Contracts. Springer Publishing Company, Incorporated, 1 ed., 2009.
[2]. P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL ’77, (New York, NY, USA), pp. 238–252, ACM, 1977.
[3]. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, Model Checking. Cambridge, MA, USA: MIT Press, 1999.
[4]. D. W. Loveland, Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science). sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.
[5]. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, “AutoProof: Auto-active functional verification of object-oriented programs,” in 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings, pp. 566–580, 2015.
[6]. B. Meyer, Object-oriented software construction, ch. 11: Design by Contract: building reliable software. Prentice Hall PTR, 1997.
[7]. AdaCore, “Tokeneer.” http://www.adacore.com/sparkpro/tokeneer/download, accessed in April 2016.
[8]. J.-R. Abrial, S. Schuman, and B. Meyer, “Specification Language,” in On the Construction of Programs, R. M. McKeag and A. M. Macnaghten, editors, pp. 343–410, Cambridge University Press, 1980.
[9]. J. Barnes, High Integrity Software: The SPARK Approach to Safety and Security. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
[10]. K. R. M. Leino, “This is boogie 2,” tech. rep., June 2008.

[11]. N. Polikarpova, C. A. Furia, and B. Meyer, “Specifying reusable components,” in Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE’10) (G. T. Leavens, P. O’Hearn, and S. Rajamani, eds.), vol. 6217 of Lecture Notes in Computer Science, pp. 127–141, Springer, August 2010.
[12]. J. Spivey, “An introduction to Z and formal specifications,” Software Engineering Journal, 1989.
[13]. C. A. Furia, C. M. Poskitt, and J. Tschannen, “The AutoProof verifier: Usability by non-experts and on standard code,” in Proc. Formal Integrated Development Environment (F-IDE 2015), vol. 187, pp. 42–55, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.
[14]. V. Rivera, S. Bhattacharya, and N. Cata˜ no, “Undertaking the tokeneer challenge in Event-B,” To appear in 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2016.
[15]. J. C. M. Baeten, “A brief history of process algebra,” Theor. Comput. Sci., vol. 335, no. 2-3, pp. 131–146, 2005.
[16]. J. Abrial, S. A. Schuman, and B. Meyer, “Specification language,” in On the Construction of Programs, pp. 343–410, 1980.
[17]. J. Abrial, The B-book - assigning programs to meanings. Cambridge University Press, 2005.
[18]. J.-R. Abrial, Modeling in Event-B: System and Software Engineering. New York, NY, USA: Cambridge University Press, 1st ed., 2010.
[19]. C. B. Jones, Software Development: A Rigorous Approach. Englewood Cliffs, N.J., USA: Prentice Hall International, 1980.
[20]. “On modelling and analysis of dynamic reconfiguration of dependable real-time systems,” in Proceedings of the 2010 Third International Conference on Dependability, DEPEND ’10, (Washington, DC, USA), pp. 173–181, IEEE Computer Society, 2010.
[21]. M. Mazzara, “Deriving specifications of dependable systems: toward a method,” in Proceedings of the 12th European Workshop on Dependable Computing, EWDC, 2009.
[22]. R. Gmehlich, K. Grau, A. Iliasov, M. Jackson, F. Loesch, and M. Mazzara, “Towards a formalism-based toolkit for automotive applications,” 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2013.
[23]. V. Rivera, N. Cata˜ no, T. Wahls, and C. Rueda, “Code generation for Event-B.” To appear in International Journal on STTT, 2016.
[24]. V. Rivera and N. Cata˜ no, “Translating Event-B to JML-Specified Java programs,” in 29th ACM SAC, (Gyeongju, South Korea), March 24-28, 2014.
[25]. G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of jml: A behavioral interface specification language for java,” SIGSOFT Softw. Eng. Notes, vol. 31, pp. 1–38, May 2006.