Information and Software Technology 53 (2011) 14–33

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A formal approach for the development of reactive systems

Olfa Mosbahi [a,b,*], Leila Jemni Ben Ayed [b], Mohamed Khalgui [c]

[a] LORIA, INRIA Lorraine, Campus Scientifique - BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex, France
[b] Faculty of Sciences in Tunis, Campus Universitaire 2092 El Manar Tunis, Tunisia
[c] Martin Luther University, Halle-Wittenberg Zentrale 06108 Halle, Germany

ARTICLE INFO

Article history:
Received 9 September 2009
Received in revised form 17 July 2010
Accepted 19 July 2010
Available online 19 August 2010

Keywords:
Reactive systems
Event-B method
Refinement
Language TLA+
Liveness properties
Verification

ABSTRACT

Context: This paper deals with the development and verification of liveness properties on reactive systems using the Event-B method. By considering the limitation of the Event-B method to invariance properties, we propose to apply the language TLA+ to verify liveness properties on Event-B models.
Objective: This paper deals with the use of two verification approaches: theorem proving and model-checking, in the construction and verification of safe reactive systems. The theorem prover concerned is part of the Click_n_Prove tool associated to the Event-B method and the model checker is TLC for TLA+ models.
Method: To verify liveness properties on Event-B systems, we extend first the expressivity and the semantics of a B model (called temporal B model) to deal with the specification of fairness and eventuality properties. Second, we propose semantics of the extension over traces, in the same spirit as TLA+ does. Third, we give verification rules in the axiomatic way of the Event-B method. Finally, we give transformation rules from a temporal B model into a TLA+ module. We present in particular, our prototype system called B2TLA+, that we have developed to support this transformation; then we can verify liveness properties thanks to the model checker TLC on finite state systems. For the verification of infinite-state systems, we propose the use of the predicate diagrams and its associated tool DIXIT. As the B refinement preserves invariance properties through refinement steps, we propose some rules to get the preservation of liveness properties by the B refinement.
Results: The proposed approach is applied for the development of some reactive systems examples and our prototype system B2TLA+ is successfully used to transform a temporal B model into a TLA+ module.
Conclusion: The paper successfully defines an approach for the specification and verification of safety and liveness properties for the development of reactive systems using the Event-B method, the language TLA+ and the predicate diagrams with their associated tools. The approach is illustrated on a case study of a parcel sorting system.

1. Introduction

Reactive systems are systems whose role is to maintain an ongoing interaction with their environment rather than produce some final values upon termination. Typical examples of reactive systems are air traffic control systems, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor. Formal methods is the term used to describe the specification and verification of these systems using mathematical and logical techniques. The main advantages of the formal approach to software construction [1–4] is that, whenever applicable, it can lead to an increase of the reliability and correctness of the resulting programs by several orders of magnitude.

Several approaches for the verification of reactive systems are available, the most prominent are algorithmic (model-checking [5,6]) and deductive verification (theorem-proving techniques [7,8]). These approaches are used to establish the correctness of reactive programs relative to their temporal specifications. Verifying the correctness of a program involves formulating a property to be verified using a suitable logic such as first order logic or temporal logic. The model-checking approach [6] involves the construction of an abstract model M, in the form of variations on finite state automata, and the construction of specification formulas ϕ, in the form of variations on temporal logic. The verification algorithm used in model-checking involves exploring the set of reachable states of the model M to ensure that the formula ϕ holds. It has gained popularity in industry because the verification procedure can be fully automated [6] and counter-examples are automatically generated if the property being verified does not hold. Furthermore, model checkers rely on exhaustive state space enumeration to

* Corresponding author at: Faculty of Sciences in Tunis, Campus Universitaire 2092 El Manar Tunis, Tunisia.
E-mail address: olfamosbahi@gmail.com (O. Mosbahi).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.07.004

establish whether a property holds or does not hold. This approach to verification puts immediate limits on the state space of problems that can be explored by model checkers. This common problem, known as the state explosion problem, is an often cited drawback of verification by model-checking.

Theorem proving is a very tedious process involving keeping in mind a multitude of assumptions and transformation rules. The calculi used in theorem proving is based on Hoare and Dijkstra theories. The first one describes a calculus to reason about program correctness in terms of *pre* and *post* conditions [7]. Dijkstra extended Hoare's ideas in the concept of "*predicate transformers*", which instead of starting with a pre-condition and post-condition, starts with a post-condition and uses the program code to determine the pre-condition that needs to hold to establish the post-condition [8]. Hoare's approach to proving correctness introduced the concept of a "Hoare triple", which is a formula in the form: $\{\phi_{pre}\}P\{\phi_{post}\}$. This formula can be read as "if property $\{\phi_{pre}\}$ holds before program P starts, $\{\phi_{post}\}$ holds after the execution of P". The program P can refer to an entire program or to a single function call, depending on the unit that is being verified. In Hoare's calculus, axioms and rules of inference are used to derive $\{\phi_{post}\}$ based on $\{\phi_{pre}\}$ and P. The syntax of P described by Hoare corresponds to a simple imperative language with the usual constructs (assignment, conditional branching, looping, and sequential statements).

A key difference between the theorem-proving approach to software verification and the model-checking approach to software verification is that theorem provers do not need to exhaustively visit the program's state space to verify properties [9]. Consequently, a theorem-proving approach can reason about infinite state spaces and state spaces involving complex datatypes and recursion. This can be achieved because a theorem prover reasons about constraints on states, not instances of states. While theorem provers have distinct advantages over model checkers, namely in the superior size of the systems they can be applied to and their ability to reason inductively, deductive systems also have their drawbacks [10]. An often cited drawback of theorem provers is that they require a great deal of user expertise and effort [11]. This requirement presents perhaps the greatest barrier to widespread adoption and usage of theorem provers. Although theorem proving and model-checking appear to be contradictory approaches to software verification, there has been considerable effort in the past 15 years to incorporate model-checking and theorem proving [12,13]. Because theorem provers and model checkers each provide complementary benefits in terms of automation and scalability, it is likely that this trend will follow and that model checkers will continue to be useful on systems of manageable size while theorem provers will be used on large systems [14].

In this paper, we have chosen the use of the Event-B method [15–17] in the development of reactive systems because it supports development of programming language code from specifications. It has been used in major safety-critical system applications in Europe (such as the Paris Metro Line 14), and is attracting increasing interest in industry. It has robust, commercially available tool support for specification, design, proof and code generation, for example Click_n_Prove and Rodin Platform [18]. It focuses on refinement to code rather than just formal specification. The basic idea of refinement [19–21] consists in successively adding implementation detail while preserving the properties required at an abstract level. In a refinement-based approach to system development, one proceeds by writing successive models, each of which introduces some additional detail while preserving the essential properties of the preceding model. Fundamental properties of a system can thus be established at high levels of abstraction, errors can be detected in early phases, and the complexity of formal assurance is spread over the entire development process.

Our aim is to check that the model has an expected behavior, i.e., satisfies the safety and liveness requirements of an informal specification. For that, one can express the requirement as formal properties that are checked on the model. The notions of safety and liveness properties have been first introduced by Lamport [22]. Informally, a safety property expresses that "*something (bad) will not happen*" during a system execution. Mutual exclusion and partial correctness are two prominent examples of safety properties. A liveness property expresses that eventually "*something (good) must happen*" during an execution. The most prominent example of a liveness property is termination. Lamport [22] distinguishes two types of liveness properties: Eventuality and Fairness. Eventuality asserts that something good must eventually happen. Fairness means that if a certain event is enabled, then the program must eventually execute it.

The Event-B method provides us with techniques and tools for specifying, refining, verifying invariant properties and implementing systems; its scope is limited to invariant properties and it is not well suited to deal with liveness properties in reactive systems. By considering the limitation of the Event-B method to safety properties, we propose to apply the language TLA$^+$ to verify liveness properties on a software behavior. The language TLA$^+$ provides us with an abstract and powerful framework for modeling, specifying and verifying safety, eventuality and fairness properties of reactive systems. The combination of the Event-B method and the language TLA$^+$ allows us to take benefits of the powerful tool of B to verify safety properties and to formulate in TLA$^+$ more natural properties that are not straightforward to express with the Event-B method and to verify them with the TLC model checker on finite state systems. For the verification of liveness properties on infinite-state systems, we propose the use of the predicate diagrams. We propose in this paper to combine and apply two techniques with the goal being to take advantages of both: theorem proving, when possible, and model-checking otherwise, in the construction and verification of safe reactive systems. The theorem prover concerned is part of the Click_n_Prove tool associated to the Event-B method and the model checker is TLC for TLA$^+$ models.

The paper is organized as follows: Section 2 gives an overview of some related works concerning B extensions for capturing and proving liveness properties, Section 3 presents a background: the Event-B method, the language TLA$^+$ and the predicate diagrams. Section 4 gives a description of the proposed approach: we extend the syntax of the Event-B method to deal with liveness properties, we give the semantics of these properties in terms of traces (*temporal B model*) and we give transformation rules from a temporal B model into a TLA$^+$ module on which we can verify liveness properties. Section 5 presents an example to illustrate our approach. Finally, Section 6 ends with a conclusion and future work.

2. Related work

In this section we review some related works and we discuss the positioning of our work and the contribution of this paper. The integration of temporal modalities to specify and to model reactive systems started with the work of Pnueli [23] and work on temporal logics (linear-time temporal and branching-time temporal logics [24], model-checking [25–27], theorem proving [28,29]). UNITY [30] is another framework, which is interesting with respect to the design of parallel programs and also to the development of distributed programs and real time aspects. The UNITY formalism is made up of a programming notation based on action systems and a specification language, which is a fragment of the linear temporal logics and a proof system. The safety and liveness properties of an algorithm are specified using a particular temporal logic then, a UNITY program is derived by stepwise refinement of the specification. Temporal modalities express safety