An Event-B Plug-in for Creating Deadlock-Freeness Theorems

Faqing Yang and Jean-Pierre Jacquot

LORIA – DEDALE Team – Nancy Université
Vandoeuvre-Lès-Nancy, France
{firstname.lastname}@loria.fr

Abstract. This paper presents DFT-generator, a small tool to generate Deadlock-Freeness Theorems (DFT) in Event-B specifications. Event-B, a companion to the B-method, allows specifiers to model systems and environments with the help states, invariants, and events. Events are guarded generalized substitutions which are fired non-deterministically. Assessing temporal properties such as termination or as non-blocking cycle is then a necessity. To overcome the lack of deadlock checking in the core of Event-B and of its supporting environment, Rodin, we have developed a practical little tool which generates the necessary theorems to prove that a model is free of deadlocks. We explain what are the deadlock theorems, why we need a tool to help generating the theorems, what problems were encountered during development. We conclude on a quick comparison with model-checking.

1 Introduction

Event-B [8,2] is an evolution of the classic B method [1]. It has been designed for modeling complex systems such as concurrent, reactive systems, or complex algorithms. Event-B is based on three principles:

- the model's base is a state: a mapping from name to values constrained by an invariant. Values are any set-theoretic construct based on integers, symbols, sets (including powersets and cartesian products) or relations (including functions), the invariant is a first-order logic formula on the values.
- events describe atomic evolution of the state. An event is a guarded substitution on the state; it can be triggered any time its guard is true; the choice of the event to trigger when several are enabled is non-deterministic.
- the development of the model is made through formal refinements. A refinement can either reify an element of the state, introduce new events, or both.

Event-B, like B, has been designed with a strong notion of "correctness" in mind. A model is correct if three conditions are met: (1) an actual initial state can be set-up, (2) each event maintains the invariant, and (3) each refinement maintains the abstract invariant as well as the concrete invariant. These conditions can be expressed in terms of proof obligations that can be generated by the support tools [9] and can be discharged by theorem-provers.

Using B or Event-B, it is then possible to develop a system which can be proved "correct," i.e, a system whose implementation is guaranteed to maintain the invariant stated at the most abstract level of its specification.

From an operational point of view, a computation modeled in Event-B is started by firing the *INITIALISATION* event; it halts when no event has its guard true. There are two opposing situations related to the halting problem. In the first, we want the system to reach a terminal state. Then, we must prove that the computation will halt. Event-B provides a notion of *variant* which is associated to proof obligations and can be used to prove the termination. In the second, we expect the system to cycle endlessly. We must then prove that the computation will not halt. Event-B does not provide us with constructs or proof obligations for this property.

Deadlock-freeness is not well integrated into the Event-B framework. As many other temporal properties, we can use "tricks." For deadlock-freeness, we build the disjunction of the guards of all events other than *INITIALISATION* and we prove it is a theorem. So, we can be sure that one event at least can always been fired.

There exists other techniques which do the same work, like:

- Animation with an interactive tool such as Brama[1] allows one to observe the occurrence of deadlocks.
- Modeling checking with a tool like ProB [6] allows one to check for (absence of) deadlocks in a finite space and partial transitions.

These techniques can help us to find a deadlock, but can't prove that the system is free of deadlocks. On the other hand, using theorem proving techniques on some theorems which express deadlock-freeness, we can formally assess that the system is actually free of deadlocks. In this paper, we present a little plug-in DFT-generator to create the deadlock-freeness theorems for Rodin platform[2]. It is useful:

- for helping to specify systems that must be free of deadlocks,
- for helping to get large scale Event-B specifications correct, in particular by replacing the error-prone manual writing of ad-hoc formulae,
- for identifying missing properties or invariants in the specification.

2 Deadlock-Freeness in Event-B Language

2.1 Deadlock-Freeness of Systems

Given a model [2] with carrier sets s, constants c, axioms $A(s,c)$, variables v, and invariants $I(s,c,v)$, the deadlock-freeness theorem states that one of the guards $G_1(s,c,v)$, ... , $G_m(s,c,v)$ of the events, exception *INITIALISATION,* is always true. The expression is the following:

$$A(s,\ c)\ \wedge I(s,\ c,\ v)\ )\ \Rightarrow G1(s,\ c,\ v)\ \vee\ \ldots\ \vee\ Gm(s,\ c,\ v)$$

[1] http://www.brama.fr

[2] http://sourceforge.net/projects/rodin-b-sharp/

In the Rodin platform, the provers can automatically reference the axioms $A(s,c)$ and invariants $I(s,c,v)$, so the form of deadlock-freeness theorem can be simplify to

$$G1(s, c, v) \vee \ldots \vee Gm(s, c, v)$$

Note that there is only one general deadlock-freeness theorem for a given model.

2.2 Deadlock-Freeness of Subset of Events

Some systems may be structured in such a way that some events are always enabled. For instance, an automated vehicle may allow its users to issue commands at any time, this would be modeled as an event with a guard which is always enabled. So, the general deadlock-freeness theorem stated above is vacuously true. But, for the model to be admissible, the part which specify the control of the vehicle must free from deadlocks. So it is important to provide specifiers a way to express the deadlock-freeness theorem of the subset of events. This can be easily done with a small user-interface. The theorem itself has the same form as the DFT for the system.

It can be noted that several partial deadlock-freeness theorems can be generated for the same model.

3 Rationale for a Plug-in

The deadlock-freeness theorems presented in Sect. 2 can be written manually for a small Event-B specification, however, this becomes soon unpractical as the size of the specification grows. Let us consider the specification which prompted this work.

The specification in [5] proposes an Event-B model of a control algorithm for automated vehicles moving in a platoon. The aim of the work was to prove that a known algorithm based on multi-agent system modeling [3,4] was safe, *i.e.,* vehicles never collide. The safety property was modeled as an invariant and all proof-obligations where discharged. So the specification was though to be "correct," yet, some simulations lead to collisions. Deadlocks between events were discovered to be the cause of the problem. So, we needed to find the source of the deadlocks, and failures in the proofs of deadlock-freeness theorems are a good way to approach it, and then, to modify the model so that it could be proven free from deadlocks.

The specification is a simplified model in 1-Dimension (vehicles moving on a "rail"). It consists in four refinements; the last one contains 15 events, excluding *INITIALISATION*. Table 1 gives some figures showing the evolution of the sizes:

A correction at the level of *platoon_2* allowed us to discharge the DFT and to produce a safe model.

Two points should be noted. The size of the theorem is roughly twice the sum of the guards of the events. So, it grows as fast as the specification does and it counts quickly several tens of lines. Deadlock-freeness is not a monotonous property with respect to refinement: it should be established for each refinement.

Automating the generation of the theorems is needed for several reasons:

Table 1. 1D initial platooning model

Machine	Events	Guards	DFT lines	Proofs of DFT	Animation Behavior
platoon	1	2	6	discharged	normal
platoon_1	3	8	16	discharged	normal
platoon_2	7	34	54	No	deadlock
platoon_3_0	9	42	68	No	deadlock
platoon_4_0	15	79	121	No	deadlock

- using "copy and paste" procedures is highly non trustable. The probability to introduce an error is high and the length of the formula makes spotting errors hard,
- any modification of the model requires a modification of the theorems. In particular, it should be easy to re-generate the formulae while correcting the model.
- manually generating the theorems is not intellectually challenging (it is boring in fact) and takes a lot of useful time.

The core of the generation is a syntactic manipulation of the model: extracting the guards, gluing them together in a single formula, and inserting it in the specification. Programming tools and techniques to cover such tasks are well known.

A small user-interface should be build for selecting subset of events in the case of partial deadlock-freeness generation. This relies also on standard technologies.

4 Development

Event-B is supported by Rodin, an open-source platform implemented in the Eclipse framework. Rodin supports the development of extension plug-ins and we have chosen to implement our tool, DFT-generator, as such.

DFT-generator is implemented in Java within the Eclipse Plug-in Development Environment. It uses the version of the programming interface provided by Rodin version 2. It is installed as a single component without dependencies from other external components.

The deadlock-freeness theorems are implemented as part of the machine invariant and flagged as "theorem." A label is generated for each theorem. It is used later to reference the formulae in the re-generation process.

While the sub-formulae of DFT are a simple conjugation of event guards, the construction of the formula requires a careful treatment of names. Existential quantifiers must be introduced for each parameters of events. Since Rodin version 2.0, it begins to support the use of the same identifier for quantified variables in different part of formulae, the development is simplified.

The development of the plug-in raised three issues. The first is the user-interface. We have kept it to a minimum: a pane on which users select first the type of generation (system DFT, partial DFT, re-generation) and, for partial DFT, the list of events to be selected. The second issue dealt with keeping memory of which events are used in a partial DFT. The initial idea to let the user erase old partial DFT and generate them

again does not work well. Actually, it defeats the goal of the plug-in to make proving deadlock-freeness as easy as possible. Events used for the re-generation are stored in the form of a comment on the theorem. Although this implementation requires a little extra-programming to extract the list of events, it has the advantage to work without any extension to Rodin database.

The last issue concerns the choice of using Event-B machine file to generate the theorems: unchecked machine file or statically checked machine file. When a machine is edited and saved, three tools (the Static Checker, the Proof Obligation Generator and the Auto-Prover) are called automatically for this edited machine and the following refinements. It can take a certain time. Modifying a early refinement becomes more time-consuming for the whole project building as the specification's length increases, many user's time would be wasted waiting. To overcome this, a "fast generation" version of the plug-in using unchecked machine file is provided for users. The limitations of the fast version is that no "extended" event is present in the specification because it becomes very difficult to access to the guards of the events which are declared "extended." A small advantage of the fast version is that the typesetting of the formula is exactly kept as users entered it . Since the guards of events become easily accessible after the full static check, a normal version of the plug-in using statically checked machine file is provided for users without the limitations of specification. We suggest to apply the normal version of the plug-in beginning the most abstract Event-B machine in the early development stage.

5 Applications of DFT-generator

A more realistic model of the platooning algorithm in 2-Dimension served as a test-bed for the DFT-generator. The specification has the same basic structure (four refinements) but contains much more events. Table 2 gives some figures about the generation:

Table 2. 2D platooning model

Machine	Events	Guards	DFT lines
platoon	1	5	19
platoon_1	3	10	28
platoon_2	19	184	249
platoon_3_0	21	177	264
platoon_4_0	39	354	637

Thanks to Rodin, which is able to manage large formulae and specification, the generation goes without any difficulty. Discharging the proof obligations of DFT is of course the biggest time-consuming activity.

6 Conclusion and Future Works

The lack of means to express temporal properties in Event-B is a well-recognized limitation in the general use the method in software development. DFT-generator is a practical attempt to provide specifiers with a way to express "simply" a very important property.

A very positive aspect of our experience is the assessment of the maturity of Rodin. The development effort we had to put into the project was reasonable. As is always the case with large API, the first use requires an important learning effort. This investment pays back afterwards since the API is easy to use and quite powerful. It is then reasonable the try to overcome limitations of Event-B by developing small assisting tools rather than waiting major evolution of the formal framework.

The size of the generated formulae, while not a surprise, still raises the issue of their manageability. The problem is actually not specific to our tool but comes from the formal core of Event-B. Assessing formal properties of large B models is a very active research domain. One way to avoid proving huge formulae is to use dynamic techniques such as model-checking; ProB [7] allows this. ProB has been used on our specifications. The model-checker did identify the deadlocks in the 1D specification of platooning. However, it failed on the 2D model due to the complexity of the geometric space (a plane). So, at present, we do not have any other mean than to discharge the 637 line-long DFT formula to assess the safety of our model.

An experimental version of DFT-generator is accessible at the website http://dedale.loria.fr/?q=en/plug-in-dft-generator.

References

1. Abrial, J.R.: The B Book. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Professional (1999)
4. Ferber, J., Muller, J.P.: Influences and Reaction : a Model of Situated Multiagent Systems. In: 2nd Int. Conf. on Multi-agent Systems. pp. 72–79 (1996)
5. Lanoix, A.: Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In: 2nd International Symposium on Theoretical Aspects of Software Engineering (TASE'08). Nanjing, China (2008)
6. Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. Journal Software Tools for Technology Transfer 10(2), 185–203 (2008)
7. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale b models with prob. Formal Aspects of Computing pp. 1–27 (2011), http://dx.doi.org/10.1007/s00165-010-0172-1, 10.1007/s00165-010-0172-1
8. Metayer, C., Voisin, L.: The Event-B Mathematical Language (Oct 2007)
9. RODIN: Rigorous Open Development Environment for Complex Systems. website (Aug 2007), http://rodin-b-sharp.sourceforge.net