arXiv:0805.3256v2 [cs.LO] 30 May 2008

Model Checking Event-B by Encoding into Alloy*

Paulo J. Matos João Marques-Silva
Electronics and Computer Science, University of Southampton
{pocm, jpms}@ecs.soton.ac.uk

May 30, 2008

Abstract

As systems become ever more complex, verification becomes more main stream. Event-B and Alloy are two formal specification languages based on fairly different methodologies. While Event-B uses theorem provers to prove that invariants hold for a given specification, Alloy uses a SAT-based model finder. In some settings, Event-B invariants may not be proved automatically, and so the often difficult step of interactive proof is required. One solution for this problem is to validate invariants with model checking. This work studies the encoding of Event-B machines and contexts to Alloy in order to perform temporal model checking with Alloy's SAT-based engine.

1 Introduction

Current day systems are ever more detailed and complex leading to the necessity of developing models that abstract unimportant implementation details while emphasizing their structure. These models are developed in order to be easily verified either by theorem provers or model checkers.

Event-B is a language based on B-method (Abrial [1996]), and supported by the open tool RODIN [1]. Part of the language is developed visually and there is a syntax for predicates and expressions(Consortium [2005]). The RODIN tool is shipped with theorem provers which allow the user to prove the model invariants either automatically or interactively.

Alloy (Jackson [2002]) is a structural modeling language based in first order logic. It is a textual language that consists of signatures, which introduce flat relations, functions, predicates and assertions that deal with the relations. By specifying predicates and assertions it is possible to perform model finding or model checking. The tool uses KodKod (Torlak and Jackson [2007]), a model

*This work is partially supported by EPSRC grant EP/E012973/1, and by EU grants IST/033709 and ICT/217069.

[1]The RODIN toolkit was developed under research project IST 511599.

finder, to convert the model within given bounds to SAT and return an instance of the model or a counter-example.

In previous work (Mikhailov and Butler [2002]) B-METHOD was combined with ALLOY. since then ALLOY was considerably improved and extended by, not only, adding new constructs but also by making the verification much more efficient. The restrictions described in the work either do not exist anymore or they can be overcome.

Until recently it was only possible to perform temporal model checking in an EVENT-B model by using a two step process: converting the model to B-METHOD and then using ProB(Leuschel and Butler [2003]). However, the solution was not straightforward as the conversion to B-METHOD was not always fully automated. More recently, a prototype ProB plugin(Ligot et al. [2007]) for the RODIN tool has been developed that provides an alternative solution for model checking EVENT-B models using the ProB model checker. Nevertheless, encoding EVENT-B to ALLOY allows building on top of the ALLOY model finding engine therefore benefiting from all of its optimizations. As mentioned elsewhere (Ligot et al. [2007]), encoding EVENT-B is not straightforward but this work shows it is possible. This work has been published in (Matos and Marques-Silva [2008]).

2 Encoding

This section summarizes the process of encoding an EVENT-B model into ALLOY. Due to space constraints the presentation will be informal and far from complete, which means that there are EVENT-B operators for which an encoding has been developed but which are not described. A full example, based on processes and mutexes, including the actual encoding and additional comments will be shown in section 3. For a detailed description of the structure and rules of EVENT-B models the reader is referred to (Consortium [2005]).

There are three aspects to the encoding: encoding of the model structures, expressions, and predicates (which are straightforward due given the existence of the logical operators in both languages).

The execution model needs to be emulated by the final ALLOY model. To this end all of the encoded models define a "State" signature which is ordered by using the ordering module and with as many fields are there are variables. The variable types are extracted from the set of EVENT-B invariants and encoded into signatures which become the type of the respective fields. A fact defines the initial state which is encoded from the "Initialisation" event. Each event is encoded in a predicate with two arguments: the current state and the next state and it evaluates to true whenever the next state reflects the triggering of an event from the current state. A final fact asserts that at every state one of the events needs to be triggered.

A carrier set is encoded as a signature with no fields and an enumerated set is encoded as signature, one per enumeration, with no fields that extend a base signature that represents the type of the enumerated set. One special enumerated set “Events” has one enumeration per event, plus an “Undef” enumeration for the initial state which is defined to be triggered by an “Undef” event. “Ev” is the type of a special field in the “State” signature. If “Ev” is x in state s', this means that it was the triggering of x that caused the transition from s to s'. Although this information is not necessary for the model checking itself, it makes the state trace of a failed invariant much more readable.

Expressions are the hardest part to encode. There is not only a myriad of complex expressions in EVENT-B but given that ALLOY uses only flat relations, some EVENT-B expressions that introduce relations with nested sets generate many (and potentially large) ALLOY expressions. Some expressions are straightforward like the domain, range, domain and range restriction and their subtraction counterparts, since they are either already defined in modules shipped as part of the current ALLOY distribution or they are very easily defined as small functions. Operators like prj_1, prj_2 and id need to be defined as ALLOY functions in order to be used. All arithmetic operators except power are defined but still, power can be defined explicitly during encoding-time depending on the defined bit width passed on to the ALLOY engine for checking. For example, a^b could be defined as $a = 0 \Rightarrow 1$ else $b = 0 \Rightarrow 0$ else $b = 1 \Rightarrow a$ else $b = 2 \Rightarrow a.mult[a] \cdots$. Function expressions can be encoded as relations and then facts can be added to the model as to assure the semantics is preserved. So, to encode $(A \rightarrow B) \leftrightarrow C$, a signature with a relation from A to B would be defined followed by a fact asserting the relation to be a total function and then yet another signature is defined with a relation from the previously defined signature to C. Although function nesting requires a new signature, it seems this is the best general solution given that ALLOY only works with flat relations.

3 Example

This section presents an example of the encoding. Although the example is simple, it is enough to have an idea of how it works. The example is based on the Alloy model `dijkstra.als` distributed with Alloy 4.1.

The EVENT-B specification is shown in table 1. It introduces two carrier sets, the processes set and the mutexes set, and two relations between processes and mutexes. The pair $\{p \mapsto m\}$ is in the relation $Holds$ iff the process p holds the mutex m and it is in the relation $Waits$ if p is waiting for m to be released in order to hold it. There are three events which control the holding and release of mutexes by processes. The “HoldOnMutex” event is triggered for a process p and a mutex m if p is not waiting for any mutex and it does not already hold m. Once it is triggered p holds m by adding the pair $\{p \mapsto m\}$ to the $Holds$ relation. The “WaitOnMutex” event is triggered when process p has to wait to hold m because there is already some other process holding it. The “ReleaseMutex” is the counterpart to “HoldOnMutex” and allows a process to release a mutex it no longer needs. This very simplistic model allows us to exercise the encoding presented in the paper. The carrier sets and the enumerated set for event types are immediately encoded into several signatures as seen

Table 1: Mutexes Event-B Specification

Carrier Sets	**Event** HoldOnMutex
$Process$	***Guards***
$Mutex$	$p \in Process$
Variables	$m \in Mutex$
$Holds$	$p \notin \mathrm{dom}(Waits)$
$Waits$	$m \notin \mathrm{ran}(Holds)$
Initialisation	***Actions***
$Holds := \varnothing$	$Holds := Holds \cup \{p \mapsto m\}$
$Waits := \varnothing$	**Event** WaitOnMutex
Event ReleaseMutex	***Guards***
Guards	$p \in Process$
$p \in Process$	$m \in Mutex$
$m \in Mutex$	$p \notin \mathrm{dom}(Waits)$
$p \notin \mathrm{dom}(Waits)$	$m \in \mathrm{ran}(\{p\} ⩤ Holds)$
$m \in \mathrm{ran}(\{p\} \lhd Holds)$	***Actions***
Actions	$Waits := Waits \cup \{p \mapsto m\}$
$Holds := Holds \setminus \{p \mapsto m\}$	**Invariants**
	$Holds \in Process \leftrightarrow Mutex$
	$Waits \in Process \leftrightarrow Mutex$
	$\mathrm{dom}(Waits) \neq Process$

Table 2: Carrier Set and Enumerated Set Encodings

sig $Process\{\}$
sig $Mutex\{\}$
abstract sig $Events\{\}$
one sig $Undef$, $HoldOnMutexE$, $WaitOnMutexE$, $ReleaseMutexE$ extends $Events\{\}$

Table 3: State Signature and Types

sig $HoldsRel\{$ $rel : Process \rightarrow Mutex$ $\}$	sig $WaitsRel\{$ $rel : Process \rightarrow Mutex$ $\}$	sig $State\{$ $Holds : HoldsRel,$ $Waits : WaitsRel,$ $Ev : Events$ $\}$

in table 2. The state signature along with the types of the $Holds$ and $Waits$ relation, which are extracted from the invariants is shown in table 3. Although it would be possible to define the relations $Holds$ as $Holds : Process \rightarrow Mutex$ and analogously for $Waits$ we didn't do that in this case because the encoding of Event-B expressions as $((A \leftrightarrow B) \twoheadrightarrow C)$ is done recursively on the structure of the expression generating $n + 1$ signatures for n operators.

The initialisation event is encoded as a fact that sets the initial state $s0$, and is shown on the left side of table 4. This encoding just defines the variable $s0$ as the first element of the state ordered set and then sets each field of the state with its initial value. On the right side the trigger is a fact that forces for each state transition one of the events to trigger.

Table 4: Event $Initalisation$

fact $Initial\{$ let $s0 = ord/first\{$ $s0.Holds.rel =$ none $\rightarrow$ none $s0.Waits.rel =$ none $\rightarrow$ none $s0.Ev =$ Undef $\}$ $\}$	fact $EventTrigger\{$ all $s : State - ord/last\{$ let $s' = ord/next[s]\{$ $HoldOnMutex[s, s']$ or $WaitOnMutex[s, s']$ or $ReleaseMutex[s, s']$ $\}$ $\}$ $\}$

Table 5: Event $WaitOnMutex$

pred $WaitOnMutex[s, s' : State]\{$
some $p : Process, m : Mutex\{$
// Guards
$!(p$ in dom $[s.Waits.rel])$
m in ran$[($dom$[s.Holds.rel] - \{p\}) <: s.Holds.rel]$
// Action
$s'.Waits.rel = s.Waits.rel + \{p \rightarrow m\}$
$s'.Holds = s.Holds$
$s'.Ev = WaitOnMutexE$
$\}$
$\}$

Table 6: Event $HoldOnMutex$ and $ReleaseMutex$

pred $HoldOnMutex[s, s' : State]\{$	pred $ReleaseMutex[s, s' : State]\{$
some $p : Process, m : Mutex\{$	some $p : Process, m : Mutex\{$
// Guards	// Guards
$!(p$ in dom$[s.Waits.rel])$	$!(p$ in dom$[s.Waits.rel])$
$!(m$ in ran$[s.Holds.rel])$	m in ran$[\{p\} <: s.Holds.rel]$
// Action	// Action
$s'.Holds.rel = s.Holds.rel + \{p \rightarrow m\}$	$s'.Holds.rel = s.Holds.rel - \{p \rightarrow m\}$
$s'.Waits = s.Waits$	$s'.Waits = s.Waits$
$s'.Ev = HoldOnMutexE$	$s'.Ev = ReleaseMutexE$
$\}$	$\}$
$\}$	$\}$

The events are encoded by translating each of the guards and actions into ALLOY expressions. Table 5 presents the encoding for the $WaitOnMutex$ event. Most of the encoding is straightforward, except for the case of the domain subtraction operator that is encoded into ALLOY using its definition based on the domain restriction operator. Note that in ALLOY, whenever the actions do now span all of the state variables, it is necessary to state that they are equal to the previous state.

The $HoldOnMutex$ events and its counterpart $ReleaseMutex$ are shown in table 6 on the left and right side respectively. The events are similar and their encoding just transforms the EVENT-B operators into those recognized by ALLOY. Given a syntactically correct ALLOY module header, it is possible to verify the ALLOY model by encoding the invariant as an assertion:

$$\text{assert } NoDeadlock\{\text{all } s : State \mid !(\text{dom}[s.Waits.rel] = Process)\}$$

and using the check command to verify it. The following check finds a deadlock within 6 states, 2 processes and 2 mutexes. This translates analogously into Event-B. There is an invariant failure within 6 event transitions for 2 processes and 2 mutexes.

$$\text{check } NoDeadlock \text{ for exactly 6 } State, \text{ exactly 2 } Process, \text{ exactly 2 } Mutex,$$
$$\text{exactly 6 } HoldsRel, \text{ exactly 6 } WaitsRel$$

The second line of the check command specifies necessary bounds based on the number of states Alloy should check. The number of such auxiliary signatures is always equal to the number of states.

4 Conclusion and Future Work

The focus of our work is to allow the users of the Event-B language to use the years of work and expertise in the development of the Alloy tools. This paper summarizes an encoding of Event-B into Alloy. The resulting Alloy model can serve to find counterexamples to false invariants and translate them back to Event-B. Future work entails the automatic generation of the encoding and its integration with the RODIN tool. The tool to be developed can then be extended to use other backends besides Alloy. Although this work focuses on model checking, there are cases where it is desirable, and often the preferred alternative, to do constraint based checking. We will investigate line of work as another possible extension to the tool.

References

J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

RODIN Consortium. Rodin deliverable D7 - Event B language. Technical report, RODIN Consortium, 2005. Available at `http://rodin.cs.ncl.ac.uk/deliverables/rodinD7.pdf`.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002.

Michael Leuschel and Michael J. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003. ISBN 3-540-40828-2.

Olivier Ligot, Jens Bendisposto, and Michael Leuschel. Debugging Event-B Models using the ProB Disprover Plug-in. *Proceedings of AFADL'07*, June 2007.

Paulo J. Matos and Joao Marques-Silva. Model checking event-b by encoding into alloy. In *Proceedings of ABZ Conference*, September 2008.

Leonid Mikhailov and Michael Butler. An approach to combining b and alloy. In *ZB'2002 – Formal Specification and Development in Z and B*, volume LNCS, pages 140–161, January 2002.

Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007. ISBN 978-3-540-71208-4.