Verifying HyperLTL properties in Event-B

Jean-Paul Bodeveix[1] [0000−0002−4179−6063], Thomas Carle[1] [0000−0002−1411−1030], Elie Fares[4,1] [0000−0002−9277−4024], Mamoun Filali[2] [0000−−0001−5387−6805], and Thai Son Hoang[3] [0000−−0003−−4095−−0732]

[1] Univ. Toulouse 3 – IRIT, Toulouse, France
[2] IRIT – CNRS, Toulouse, France
[3] University of Southampton, UK
[4] Higher Colleges of Technology, Ras Al Khaimah, UAE

Abstract. The study presented in this paper is motivated by the verification of properties related to hardware architectures, namely timing anomalies that qualify a counter-intuitive timing behaviour. They are avoided by a monotonicity property which is an Hyper-LTL property. We present how to prove some classes of Hyper-LTL properties with Event-B.

1 Introduction

The study presented in this paper is motivated by the verification of properties related to hardware architectures, especially multicore platforms. We are concerned about timing anomalies that qualify a counter-intuitive timing behaviour: a locally faster execution leads to an increase in the execution time of the whole program [7]. Monotonicity is a property that ensures such timing anomalies do not occur. The monotonicity property does not belong to the usual safety and liveness classes because it relates to two distinct execution traces. Such kind of properties have already been identified and coined as hyperproperties [2]. Here, we investigate the notion of hyperproperties and evaluate their verification methods [6] for our Event-B context[5]. The verification of hyperproperties has mainly been studied in the context of model-checking. Here, our models are parameterized and will be formalized and reasoned about with the help of proof assistants. For this purpose, in Section 3, we reuse a proof-based framework [3] on top of the Rodin Event-B framework.

2 Hyperproperties

Definitions. Hyperproperties introduce universal and existential trace quantifiers as in the general purpose TLA+ [5] language or in the specialized language HyperLTL [1] extending linear temporal logic with trace quantifiers:

$$\varphi ::= \exists \pi \cdot \varphi \ \mid \ \forall \pi \cdot \varphi \ \mid \ \psi \qquad \psi ::= a_\pi \ \mid \ \psi \vee \psi \ \mid \ \neg\psi \ \mid \ \mathbf{X}\ \psi \ \mid \ \psi \ \ \mathbf{U}\ \psi$$

[5] Hyperproperties have been mainly applied to the security domain.

where $\forall\pi$ and $\exists\pi$ are universal and exisitential quantification over some trace π, a_π is atomic proposition interpreted on a trace π, $\mathbf{X}$ and $\mathbf{U}$ are LTL next and until temporal operators. Other logical and temporal operators can be defined as syntactical sugar accordingly.

HyperLTL semantics. Let P be a set of propositions, $\mathcal{T} \subseteq \mathbb{N} \to 2^P$ a set of traces, Π a set of path identifiers and $T \in \Pi \twoheadrightarrow \mathcal{T}$. The judgment $T \models_i \varphi$ (φ is satisfied by the assignments T at time i) is defined as:

$$\begin{array}{ll}
T \models_i a_\pi & \text{if } a \in T[\pi][i] \\
T \models_i \neg\psi & \text{if } T \not\models_i \psi \\
T \models_i \psi_1 \vee \psi_2 & \text{if } T \models_i \psi_1 \text{ or } T \models_i \psi_2 \\
T \models_i \mathbf{X}\ \psi & \text{if } T \models_{i+1} \psi \\
T \models_i \psi_1\ \mathbf{U}\ \psi_2 & \text{if } \exists j \geq i \text{ s.t. } T \models_j \psi_2 \text{ and } \forall\, i \leq k < j,\ T \models_k \psi_1 \\
T \models_i \exists\pi \cdot \varphi & \text{if } \exists\, t \in \mathcal{T} \text{ s.t. } \pi \mapsto t, T \models_i \varphi \\
T \models_i \forall\pi \cdot \varphi & \text{if } \forall\, t \in \mathcal{T} :\ \pi \mapsto t, T \models_i \varphi
\end{array}$$

where $\pi \mapsto t, T$ denote the same function as T, except π is mapped to t. An important point to note is that time advances synchronously in every considered trace.

Examples of Hyperproperties. Hyperproperties have mainly been introduced for security. We illustrate the use of HyperLTL through the expression of two security properties. They use a predicate P encoding the behavior of the considered system.

– *Observational determinism* is a 2-safety property. It states that two traces agree at any time on their observable outputs if they agree at any time on their observable inputs:

$$\forall\pi_1 \cdot \forall\pi_2 \cdot P(\pi_1) \wedge P(\pi_2) \wedge \Box(inp_{\pi_1} \Leftrightarrow inp_{\pi_2}) \Rightarrow \Box(out_{\pi_1} \Leftrightarrow out_{\pi_2})$$

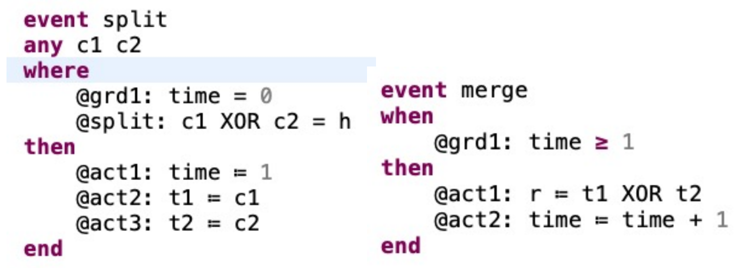
– *Generalized non-interference*: It states that observing public information reveals no private information: one can find a trace π that agrees with π_1 on its public inputs and with the private part of any other trace π_2, thus not revealing π_1 private information.

$$\begin{array}{l}
\forall\pi_1 \cdot \forall\pi_2 \cdot P(\pi_1) \wedge P(\pi_2) \Rightarrow \\
\quad \exists\pi \cdot P(\pi) \wedge \Box(\mathrm{pub}_\pi \Leftrightarrow \mathrm{pub}_{\pi_1} \wedge \mathrm{priv}_\pi \Leftrightarrow \mathrm{priv}_{\pi_2})
\end{array}$$

Example 1 (Secret Transfer). Consider a system where *Alice* wants to send a secret bit h to *Bob* by "spliting" h into two bits t_1 and t_2 such that $h = t_1\ XOR\ t_2$ and transfer t_1 and t_2 separately using different channel to *Bob*. Upon receiving both t_1 and t_2, *Bob* reconstructs the received bit $r = t_1\ XOR\ t_2$ accordingly. Assuming that any intruder can only have access to either t_1 or t_2 but not both, the protocol ensures that the value of the secret h is not revealed.

– *Observational determinism.* if two traces have the same input h, they have the same output r: $\forall\ \pi_1, \pi_2 \cdot h_{\pi_1} = h_{\pi_2} \Rightarrow r_{\pi_1} = r_{\pi_2}$.
– *Generalized non-interference.* For every two traces π_1 and π_2, there exists trace π such that $h_\pi = h_{\pi_1}$ and $t_{1\pi} = t_{1\pi_2}$, i.e., leaking information about t_1 alone does not reveal the secret h.

An Event-B model of *split* and *merge* can be seen below. Note that variable *time* is used to ensure event ordering. At time 0, `split` splits the Boolean h into t_1 and t_2. Afterwards, `merge` computes the result r.



```
event split
any c1 c2
where
    @grd1: time = 0
    @split: c1 XOR c2 = h
then
    @act1: time ≔ 1
    @act2: t1 ≔ c1
    @act3: t2 ≔ c2
end
```

```
event merge
when
    @grd1: time ≥ 1
then
    @act1: r ≔ t1 XOR t2
    @act2: time ≔ time + 1
end
```

3 Verification in Rodin/Event-B

Verifying HyperLTL properties needs comparing several executions and thus, in the context of Event-B, several copies of the same machine using copies of state variables. This comes to build composed machines, a feature present in the CamilleX [4] plugin. We thus present this plugin and how it can be used to produce proof obligations ensuring some HyperLTL properties.
The CamilleX plugin is a Rodin plugin which brings several syntactic extensions to Event-B among which are machine inclusion and event synchronization.

- machine inclusion: the command `includes` M `as` m_1 ... m_k inserts k copies of M variables and invariants to the current machine, variables of each copy being prefixed by the corresponding alias m_k.
- event synchronization: the command `synchronises` $m_i \cdot e$ called from an event ev where m_i is machine inclusion prefix and e an event of the included machine adds e parameters and guards to ev after prefixing these parameters and the variables referenced by the guards by m_i.

The CamilleX plugin has some limitations we have bypassed through some extensions. The first one concerns the set of copied parameters, guards and invariants. The second one concerns the copy of proofs.

Parameters, guards and invariants. The original CamilleX plugin only copies information from the directly included machine.We have modified this behavior as follows: (1) the machine inclusion command copies invariants of (indirectly) refined machines that do not use hidden machine variables. (2) the event synchronization command copies event parameters and guards of the whole chain of event extensions.
It has to be noted that copied invariants are not guaranteed to be preserved both in the original and the modified CamilleX plugins. Their preservation proof in the included machine may depend on the invariants established in the whole chain of refinements. We only copy invariants referring visible variables.

Proofs We have added to the CamilleX plugin the generation of proofs of the invariants copied from included machines. Consider a machine `M` as the one given in the next section. The preservation of invariant I by the event *ev* leads to a proof obligation named `ev/I/INV`. Consider a machine `M1` including two copies of `M` prefixed `m1` and `m2` and thus two invariants named `m1.I` and `m2.I`. The introduction of the event *ev* in `M1` leads to the generation of two proof obligations named `ev/m1.I/INV` and `ev/m2.I/INV`. Their proofs have in fact already been done in machine `M` up to the renaming of variables `V` and of `M` and parameters `P` of the event `ev` of `M`. The copied proofs may be incorrect because they use invariants visible by `M` and not copied in `M1`. They are thus replayed.

4 Verification of hyperproperties with CamilleX

We now use the (extended) CamilleX plugin to build machines in charge of producing proof obligations matching the assertion of the fact that a machine satisfies some HyperLTL properties. The class of properties supported by this methodology should have the shape $M \models \forall^+\exists^?\Box P$ (any non negative number of universal quantifiers followed by at most one existential quantifier).
Consider the machine M having a set of variables V together with invariants I and an indexed family of events ev_i:

```
machine M  variables V  invariants  I
  event evi  when Gi then Ai end
end
```

The universal prefix is obtained through composition: we create the synchronous product of several copies of the machine `M` to be checked. The optional existential quantifier is managed by refinement, where the machine corresponding to the existential quantification is refined by the synchronous product of several copies of machine `M`. In order to make the schema simpler, we separate $\forall^+$ and $\forall^+\exists$ managements.

4.1 $M \models \forall^+\Box P$ verification

In order to check the $\forall^+\Box P$ property over our given machine M, we build a `check` machine containing two or more independent copies of M state variables together with their invariants. The box property P is added as an invariant over the product state space. To simulate the universal quantifications over the behaviors, a free product of machine events is build: the behavior of the product machine is obtained by independently choosing at each step an event in each machine.

```
machine check includes M as m1 m2
  // two copies of M state variables and invariants
 invariants  P(m1_V,m2_V) // property to be proved as invariant
events // synchronous product (one step in each copy)
  event ev_ij synchronises m1·evi synchronises m2·evj end
end
```

Additional invariants might be required for the proof of the HyperLTL property.

Example 2 (Linear Pipeline). To illustrate the verification of a $\forall\forall$ property, we consider a strongly simplified specification of a linear pipeline and its monotony property. A processor cycle processes a stream of instructions through a sequence of stages. The `pos` variable maps instructions to their stage. Instructions may skip some stages. This feature is described by the `jumps` state variable. It must be seen as an input fixed during machine initialization and left unchanged during execution. A cycle moves instructions to the right while performing jumps as required.

```
machine mRISC sees cGEN // defines State as ℕ₁
 variables  pos jumps
 invariants
   @pos pos ∈ 1..LEN ↣ State
   @jumps jumps ∈ 1..LEN → ℙ(State) // skipped states (hits),
   @pos_jumps ∀i·i ∈ 1..LEN ⇒ pos(i) ∉ jumps(i)
events ...
   event  cycle  any P where
      @P_ty P ∈ 1..LEN ↣ State
      @P_gt ∀i·i ∈ 1..LEN ⇒ pos(i) < P(i)
      @H_to_P ∀i·i∈1..LEN ⇒ pos(i)+1..P(i)−1 ⊆ jumps(i)
      @at_P ∀i·i∈1..LEN ⇒ P(i) ∉ jumps(i)
   then @npos pos := P end
end
```

The property to be checked is that if a behavior makes fewer jumps, all the instructions are less advanced in the pipe.

```
machine check sees cGEN includes mRISC as m1 m2
 invariants
   @jumps ∀i·i∈1..LEN ⇒ m1_jumps(i) ⊆ m2_jumps(i)
   @isLate ∀i·  i∈1..LEN ⇒ m1_pos(i) ≤ m2_pos(i)
events
   event INITIALISATION ... end
   event product  synchronises  m1·cycle  synchronises  m2·cycle  end
end
```

4.2 $M \models \forall^{+}\exists\Box P$ verification

In order to check that M satisfies the hyperproperty $\forall\pi^{+}\exists\pi'\Box P(V_{\pi}^{+}, V_{\pi'})$, we first introduce a `check` machine which represents the product composition of the $\forall+$ portion (similar to the previous section) and prove that this machine refines the original machine M so that the property P is satisfied at any instant. To find the matching abstract trace, we must provide an event-to-event mapping. The `refines` clause allows specifying such a mapping.

```
machine check refines M includes M as m1 m2
 invariants  P(V+, V) // state-only body of hyper property
events
  event ev_ij refines ev_k // selected to get P
  synchronises m1·ev_i // Syncrhonise with the ev_i of m1
  synchronises m2·ev_j // Syncrhonise with the ev_j of m2
  end
end
```

Discharging the proof obligations of the machine `check` guarantees the correctness of the HyperLTL property as refinement proof obligations the existence of an abstract trace linked to the concrete trace through the provided gluing invariant. However, a failure in these proof attempts is inconclusive.

Example 3 (Secret Transfer non-interference). Consider Example 1 and focus on the generalised non-interference property, which can be formalized as $\forall\pi_1\forall\pi_2\exists\pi.h_\pi = h_{\pi_1} \wedge t1_\pi = t1_{\pi_2}$. We construct a composition machine for π_1 and π_2 and state that it is a refinement of M with the appropriate gluing invariants.

```
machine check refines M includes M as m1 m2
  @glue−h h = m1_h     @glue−t1 t1 = m2_t1
events
  event m1_split_m2_split refines split
  synchronises m1· split synchronises m2· split
  with
    @c1 c1 = m2_c1   @c2 c2 = m1_h XOR m2_c1
  end

  event m1_merge_m2_merge refines merge
  synchronises m1·merge synchronises m2·merge end
  ...
end
```

The refinement proof relies on the notion of "witnesses" in Event-B when the hidden bit is split. The witness for $c1$ (which will eventually be the value for $t1$) is chosen the same as $m2\_c1$ (so that the observed bit $t1$ will be the same for π_2 and π). The witness for (hidden) $c2$ is then chosen to ensure that $c1\ XOR\ c2 = m1\_h$ (since the hidden input h is the same for π_1 and π). In general, to prove refinement, it can be necessary to add invariants or revise the abstract model by event splitting.

5 Conclusion

This paper has presented a way to verify HyperLTL properties in the Event-B framework thanks to the use of the CamilleX plugin for building products of Event-B machines. As we have said, our example is strongly simplified. We envision to enrich this work first with respect the instrumentation of the verification of HyperLTL properties and with respect to our initial case study.

References

1. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. CoRR **abs/1401.4492** (2014), http://arxiv.org/abs/1401.4492
2. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393, https://doi.org/10.3233/JCS-2009-0393
3. Hoang, T.S., Snook, C., Dghaym, D., Fathabadi, A.S., Butler, M.: The CamilleX framework for the rodin platform. In: ABZ 2021- 8th International Conference on Rigorous State Based Methods: ABZ 2021 (07/06/21 - 11/06/21). pp. 124–129 (June 2021), https://eprints.soton.ac.uk/448174/
4. Hoang, T.S., Snook, C., Dghaym, D., Fathabadi, A.S., Butler, M.: Building an extensible textual framework for the rodin platform. In: Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops: AI4EA, F-IDE, CoSim-CPS, CIFMA, Berlin, Germany, September 26–30, 2022, Revised Selected Papers. p. 132–147. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-26236-4_11, https://doi.org/10.1007/978-3-031-26236-4_11
5. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., USA (2002)
6. Lamport, L., Schneider, F.B.: Verifying hyperproperties with TLA. In: 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021. pp. 1–16. IEEE (2021). https://doi.org/10.1109/CSF51468.2021.00012, https://doi.org/10.1109/CSF51468.2021.00012
7. Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A definition and classification of timing anomalies. In: Mueller, F. (ed.) 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany. OASIcs, vol. 4. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006), http://drops.dagstuhl.de/opus/volltexte/2006/671