Extensible Record Structures in Event-B

Asieh Salehi Fathabadi[0000−0002−0508−3066], Colin Snook[0000−0002−0210−0983], Thai Son Hoang[0000−0003−4095−0732], Dana Dghaym[0000−0002−2196−2749], and Michael Butler[0000−0003−4642−5373]

ECS, University of Southampton, Southampton, U.K.
{a.salehi-fathabadi, cfs, t.s.hoang, d.dghaym, m.j.butler}@soton.ac.uk

Abstract. Event-B is a state-based formal method for system development. The Event-B mathematical language does not support a syntax for the direct definition of structured types such as records. This paper proposes extending the Event-B language with direct record definitions. A key feature is the ability to extend records with new fields in refinement steps. The XEvent-B tool, which supports a textual representation of Event-B models, is extended to provide support for direct record definition and automatic transformation of record structures into standard Event-B elements. We demonstrate this work by modelling of the Tokeneer case study.

1 Introduction and Motivation

In Event-B [1], system state is modelled using data structures. However, the Event-B mathematical language does not support a syntax for the *direct* definition of record data structures. Record structures result in more readable models while retaining the ease of refinement and proof. We have extended the Event-B language to support direct definition of record structures. Our motivation is to allow modellers to use the familiar concepts of record structured datatypes in Event-B modelling. Moreover, we aim to have a smooth integration of records with the step-wise refinement paradigm in Event-B. Here, a record is a collection of fields of different data types, in fixed number and sequence [7]. In refinement, we may extend existing records with new fields. This allows us to introduce details to the structured data in an incremental fashion. Our work is inspired by [3] but offers an improved translation into Event-B (see discussion in Section 6).

This paper is structured as follows: Section 2 provides background on the Event-B. Section 3 presents tool support. Section 4 reports of the syntax and transformation of record structure, followed by application of it in the tokeneer case study presented in Section 5. Section 6 compares with other data structuring methods. Finally Section 7 concludes.

2 Background

Event-B [1] is a refinement-based formal method for system development. An Event-B model contains two parts: *contexts* for static data and *machines* for

This work is supported by the HiClass project (113213), which is part of the ATI Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture.

dynamic behaviour. Contexts contain carrier sets s, constants c, and axioms $A(c)$ that constrain the carrier sets and constants. Machines contain variables v, invariant predicates $I(v)$ that constrain the variables, and events. Event-B uses a mathematical language that is based on set theory and predicate logic. Event-B is supported by the Rodin Platform [2], an extensible open source toolkit which includes facilities for modelling and verification techniques.

3 Tool: CamilleX

The records feature is based on our EMF framework for Event-B [11] which uses the *Eclipse Modeling Framework* (EMF) [13] and provides extension and translation mechanisms to extend the Event-B language. CamilleX [5] provides an extensible text representation of Event-B models (as opposed to Rodin XML files). CamilleX supports two types of text files, *XMachine* and *XContext*, which are automatically translated to the corresponding Rodin machine or context. We have extended the CamilleX grammar to support the new records extension. CamilleX uses the XText [4] framework to implement an editor and translation tool. XText is an EMF-based open source framework for developing domain-specific languages with a human-readable text persistence. When CamilleX files are saved, the CamilleX translator calls any extension translators. In our case a records translator will generate the 'flattened' standard Event-B elements in the target machine and/or context. Records are translated to standard Event-B and hence direct support for records is not required in the existing Rodin tools.

4 Record Structure

Record Syntax: A record in an Event-B XMachine or XContext text file is specified using the following syntax:

record record_id [**extends** extended_record_id]
(field_id : [multiplicity] field_type)∗

Each record has an identifier, *record_id*, and can optionally extend another record, *extended_record_id*. A record contains zero or more field(s). A record field has an identifier, *field_id*, an optional multiplicity, *multiplicity*, and a data type, *field_type*.

Multiplicity: Multiplicity defines the minimum and maximum number of times the field element can appear in the record. There are three alternative multiplicity options for a field: - **one**: the field contains exactly one value. - **opt**: (optional) the field contains zero or one values. - **many**: the field contains zero or more values. If no multiplicity option is specified for a field, it is considered as **one**. While the **one** multiplicity is common, our **opt** and **many** multiplicities give modellers the flexibility in defining the cardinality associated with each field in a record.

Extension: A record can be extended via single inheritance, allowing record structures to model hierarchies that occur in refining a model. Instances of an extending record have the fields of the record that they extend as well as the new fields that they define. Static record fields are specified in a context, while dynamic record fields are specified in a machine. A record can extend another record in three ways as follows:

- A record specified in a context/machine extends a record specified in the same context/machine. This approach supports hierarchical definition of data structures. Where some records share some fields, the common fields can be specified as a parent record which is extended by child records.

- A record specified in a machine, extends a record specified in a context, seen by the machine. This is where a record contains both static and dynamic data, we extend fields of a static record by new dynamic fields.

- A record specified in a refining context/machine, extends a record specified in the abstract context/machine. This approach supports data refinement where new fields are defined for an existing abstract record in a refinement level.

Record Transformation: By saving the XText file, a context/machine file including the translated Event-B elements for specified records are generated. The translation elements includes sets, constants and axioms in a context and variables and invariants in a machine. In a context:
- a non-extending record is translated to a set: **sets** record_id
- an extending record is translated to a constant and an axiom, specifying the record type as a sub-set relation:
constants record_id
axioms record_id $\subseteq$ extended_record_id
- each field is translated to a constant and an axiom, specifying the field type:
constants field_id
axioms field_id $\in$ record_id ($\leftrightarrow$/$\nrightarrow$/$\rightarrow$) field_type
There are three alternative relation types for a field depending on its multiplicity: - "many": is translated to a relation ($\leftrightarrow$), -"opt" (optional): is translated a partial function ($\nrightarrow$), -"one": is translated to a total function ($\rightarrow$).
In a machine:
- a record in a machine must extend another record. An extending record is translated to a variable and an invariant, specifying the record type as a sub-set relation:
variables record_id
invariants record_id $\subseteq$ extendedc_record_id
- each field is translated to a variable and an invariant, specifying the field type:
variables field_id
invariants field_id $\in$ record_id ($\leftrightarrow$/$\nrightarrow$/$\rightarrow$) field_type

5 Case study

The Tokeneer system [8] consists of a secure enclave and a set of system components including a card reader and a fingerprint reader. In this paper, we outline the application of record structures in specifying the system-level states. The primary objective of the tokeneer system is to prevent unauthorised access to the secure enclave. A successful scenario involves: arrival of a permitted user at the door who then presents a card on the card reader and a matching finger print at the fingerprint reader. The system will then unlock the door allowing the user to open it and enter the enclave. A card contains a token and a token includes certificates. Figure 1 presents the hierarchy of certificate types.

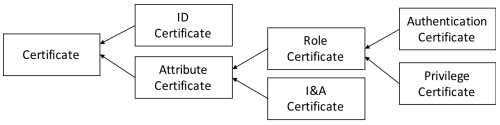

Fig. 1: Hierarchy of tokeneer certificate types

The Event-B model of tokeneer includes an abstract level and two levels of refinement where the door and card specifications are modelled respectively. The certificate hierarchy types, token and card structures are specified using record definitions in the context of the second refinement as below (left); and the machine *m2_card*, seeing context *c2_card*, is partly presented below (right):

```
context c2_card extends c1_door
sets KEYPART PRIVILEGE
     CLEARANCE TOKENID
     FINGERPRINT
records
record CERTIFICATE
  idIssuer : issuer
  validityPeriod : time
  signature : opt KEYPART
record IDCert extends CERTIFICATE
  subject : USER
  publicKey : KEYPART
record AttCert extends CERTIFICATE
  baseCertId : issuer
  tokenId : TOKENID
record RoleCert extends AttCert
  role : PRIVILEGE
  clearance : CLEARANCE
record PrivCert extends RoleCert
record AuthCert extends RoleCert
record IandACert extends AttCert
  fingerprintTemplate : FINGERPRINT
record TOKEN
  tokenID : TOKENID
  idCert : IDCert
  privCert : PrivCert
  iandACert : IandACert
record CARD
  token : TOKEN
record USER extends USER
  fingerprint : FINGERPRINT
end
```

```
machine m2_card refines m1_door sees
      c2_card
variables validToken
records
record USER extends USER
  holds : opt CARD
record TOKEN extends TOKEN
  authCert : opt AuthCert
invariants
@inv1 : validToken ⊆ TOKEN
@inv2 : holds∼ ∈ CARD ⇸ USER
@inv3 : ∀tkn . tkn ∈ validToken ⇒
  baseCertId(privCert(tkn)) = idIssuer(
    idCert(tkn)) ∧
  baseCertId(iandACert(tkn)) =
    idIssuer(idCert(tkn)) ∧
  tokenId(privCert(tkn)) = tokenID(tkn
    ) ∧
  tokenId(iandACert(tkn)) = tokenID(
    tkn)
events
event holdCard any user crd where
 @grd1 : user ∈ USER
 @grd2 : crd ∈ CARD
 @gdr3 : user ∉ dom(holds)
 @grd4 : crd ∉ ran(holds)
 @gdr5 : token(crd) ∈ validToken
 @gdr6 : fingerprint(user) =
     fingerprintTemplate(iandACert(
     token(crd))) then
 @act1 : holds(user) := crd end
end
```

Record *USER*, includes the *fingerprint* field, specified in the context, *c2_card*, and is extended in the refining machine, *m2_card*, to include the *CARD* field. We modelled the fingerprint as a static property of a user and holding a card as an optional dynamic property (i.e., defined in the machine). The *holds* field of the *USER* record is declared as optional, specifying that each user can hold at most one card. The invariant *inv2* specifies that each card can be held by at

most one user. This is as example of how we can specify extra properties of a defined record.

Record *TOKEN* including a token ID and three static certificates, is extended to include the dynamic optional authorisation certificate in the machine, *m2_card*. During the first attempt to enter the enclave, a valid authorisation certificate is issued and added to the card. A token is valid if all of the certificates on it are well-formed: each certificate correctly cross-references to the ID certificate, and each certificate correctly cross-references to the token ID. This requirement is specified as an invariant, *inv3*; this is an example of the use of records in an invariant.

As an example of modification of the field values, we present the event $holdCard$ here. A set of guards check the validity of the card token, $grd5$, and matching fingerprint, $grd6$. If the guards hold then the record field *holds* is updated to include the new pair of $user$ and crd.

6 Comparison with other data structuring methods

Alloy [6] is a lightweight state-based formal modelling language which influenced our approach. Records are similar to Alloy *signatures*, with the same notion of extends and fields. We based our syntax for field multiplicity on Alloy multiplicities (lone=opt, one=one, some=many). Sibling signatures are disjoint by default and signatures can be marked as *abstract* indicating that their *subsignatures* form a partition. We did not include disjoint/partition features in records because it requires all siblings to be declared simultaneously in the same refinement. *Facts* can be added to constrain signatures with implicit quantification over the instances of the signature. For records, disjointness, partition and constraints can be specified using axioms or invariants.

Declarative Records A previous attempt [10] to support records (based on [3]) extended the Rodin Event-B notation. Records were converted to 'plain' Event-B by the static checker for proof verification. Records were only available as constants. Hence fields could not be varied individually. In order to change the value of a field, a new instance of the record was selected with the desired new field value and other fields unchanged. However, difficulties were experienced using the ProB model checker since it has to instantiate possible values of records. Our new approach supports variable record structures which alleviates the tooling challenges and our implementation is based on the CamilleX framework to provide human-usable text persistence whereas the previous plug-in persisted models as XML.

UML-B Class diagrams [9] structure data in a similar way to records. Classes, attributes and associations are linked to Event-B data elements (carrier sets, constants, or variables) and generate constraints on those elements that reflect the data relationships. A class represents a set of instances as does a record and attributes or associations are similar to fields. However, class diagram models provide more options (e.g. multiplicity of fields, partitioning of subsets) than are required in records. Initially, we considered whether a clean human-usable text persistence for class diagrams might provide an efficient route to textual record

structures. However, the abstract syntax (meta-model) for class diagrams has been designed to support diagram editors. Since the additional options and structural differences of class diagrams must be accommodated in the persistence, they impact the concrete syntax for records. Therefore, while class diagrams remain an alternative option for data structuring, it is beneficial to provide a separate textual syntax and tooling for records in Event-B.

7 Conclusion and Future Works

In the Event-B formal language, record structures can be defined using standard Event-B elements. For example, a dynamic record field can be specified as a variable and an invariant specifying the type of field. However there is no support for direct definition of record structures. As illustrated by the Tokeeneer case study, direct definition of records using our approach results in improved readability in modelling compared with indirect definition in Event-B. In particular, we have designed a new notation for extensible records, so that records can be smoothly extended during the refinement process. However when the Event-B model is refined, records can be refined either by extension (super-position of new fields), presented in this paper, or data refinement (replacement of fields), which is a future research direction. Furthermore, record structures will help identifying program data structures when generating code from Event-B model [12].

References

1. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. J-R Abrial and et al. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, 2010.
3. N. Evans and M. Butler. A Proposal for Records in Event-B. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2006.
4. M. Eysholdt and H. Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *OOPSLA*, pages 307–309. ACM, 2010.
5. Thai Son Hoang, Colin Snook, Dana Dghaym, Asieh Salehi Fathabadi, and Michael Butler. The CamilleX Framework for the RodinPlatform. 2021. accepted in ABZ.
6. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.
7. M. Flatt M. Felleisen, R. Bruce Findler and Sh. Krishnamurthi. *How To Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.
8. Praxis. Tokeneer. `https://www.adacore.com/tokeneer`. Accessed March 2021.
9. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, 2006.
10. Colin Snook. Event-B and Rodin Wiki: Records Extension. `http://wiki.event-b.org/index.php/Records_Extension`, 2010. Accessed March 2021.
11. Colin Snook and et al. Event-B and Rodin Wiki:. `http://wiki.event-b.org/index.php/EMF_framework_for_Event-B`, 2009. Accessed March 2021.
12. Sanjeevan Sritharan and Thai Son Hoang. Towards generating SPARK from Event-B models. In *IFM 2020*, Lecture Notes in Computer Science. Springer, 2020.
13. D. Steinberg, F. Budinsky, M. Paternostro, and Ed Merks. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley Professional, 2nd edition, 2008.