Adding records to Alloy*

Julien Brunel[1,2], David Chemouil[1,2], Alcino Cunha[3,4][0000−0002−2714−8027], and Nuno Macedo[3,5][0000−0002−4817−948X]

[1] ONERA DTIS, Toulouse, France
[2] Université fédérale de Toulouse, Toulouse, France
[3] INESC TEC, Porto, Portugal
[4] University of Minho, Braga, Portugal
[5] Faculty of Engineering of the University of Porto, Porto, Portugal

Abstract. Records are a composite data type available in most programming and specification languages, but they are not natively supported by Alloy. As a consequence, users often find themselves having to simulate records in ad hoc ways, a strategy that is error prone and often encumbers the analysis procedures. This paper proposes a conservative extension to the Alloy language to support record signatures. Uniqueness and completeness is imposed on the atoms of such signatures, while still supporting Alloy's flexible signature hierarchy. The Analyzer has been extended to internally expand such record signatures as partial knowledge for the solving procedure. Evaluation shows that the proposed approach is more efficient than commonly used idioms.

Keywords: Alloy · Formal specification · Model checking

1 Introduction

Records (or *structs*) are a composite data type, available in most programming and specification languages, that represent n-ary Cartesian products together with named projections (a.k.a. fields). The Alloy language [3], however, does not support such composite types; only sets and flat n-ary relations can be modeled. Users often simulate a record type using a signature and associated fields, and enforcing two constraints: i) *completeness*[6]: there is a record atom for each possible combination of field values, so that every record is always available; and ii) *uniqueness*: each record is uniquely represented by a single atom, so that equality between similar records holds. This manual encoding is however cumbersome, error-prone and difficult to maintain. This paper proposes to extend Alloy with a new **`struct`** signature modifier to improve the support for records. Hierarchies of record signatures can also be defined. This extension

* This work is supported by the research project CONCORDE of the Defense Innovation Agency (AID) of the French Ministry of Defense (2019650090004707501), and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project EXPL/CCI-COM/1637/2021.

[6] A particular case of *generator axiom* [3].

```
1 sig ValueA, ValueB {}
2 var sig Id {}
3 sig Node { succ : set Node, var inbox : set Msg }
4 struct sig Msg { var id : one Id, pl : lone Payload }
5 abstract struct sig Payload { from : one Node }
6 struct sig PayloadA extends Payload { val : one ValueA }
7 struct sig PayloadB extends Payload { val : one ValueB }
8 ...
9 fact trace { always some m:Msg,n:Node | send[n,m] or process[n,m] }
10 check { safety } for 3 but 10 steps
```

Fig. 1. Message-passing protocol with the `struct` extension

is backed by a direct translation from the Alloy Analyzer to the underlying Pardinus model finder [8,5]. The Alloy visualizer is also adapted accordingly to ease the interpretation of instances with records.

2 Motivating example

2.1 Example with the proposed extension

Consider, for instance, a model of an abstract message-passing protocol where each message is comprised of an internal identifier and of an optional payload made of the identifier of the sender node and some value that can be of different types. During analysis, we expect the solvers to consider domains with different sets of identifiers, nodes and values, but be able to refer to *all possible messages*.

A possible encoding in Alloy using the proposed extension is shown in Fig. 2. A record signature `Msg` (l. 4) represents the available messages, composed of a mandatory `Id` and an optional `Payload` with additional information. `Payload` (l. 5) is also a record signature with the identifier of the sender node (here abstracted by referring directly to the `Node`), but is declared as **`abstract`**, so that it can be extended by messages containing values of different types, here just denoted by `PayloadA` (l. 6) and `PayloadB` (l. 7) pointing to `ValueA` and `ValueB` elements, respectively. Signatures marked with **`struct`**, and their fields, can then be used as any plain signature in the rest of the model, as in the inbox of nodes (l. 3) or in the **`fact`** `trace` (l. 9) that controls the evolution of the protocol in a typical Alloy style. During analysis, all plain signatures take arbitrary values within the specified scope, as in plain Alloy. Record signatures are considered to be *complete*, containing all possible combinations of values within the universe of discourse, and the user is not expected to control their scope. For instance, the **`check`** `safety` command (l. 10) imposes a maximum scope of 3 for the plain signatures. In a state that happens to have 3 atoms of each plain signature, this would result in 9 `PayloadA` atoms, 9 `PayloadB` atoms, and 57 `Msg` atoms. Note that the set of available identifiers is mutable during the execution of the protocol (l. 2): the content of record signatures may then change in each state.

```
1 // same signatures as in Fig.1, but without the struct keyword
2 ...
3 pred unique {
4   always {
5     all disj m1,m2:Msg | m1.id ≠ m2.id or m1.pl ≠ m2.pl }
6     all disj p1,p2:PayloadA | p1.from ≠ p2.from or p1.val ≠ p2.val
7     ... } }
8 pred complete {
9   always {
10    all n:Id,p:Payload | some m:Msg | m.id = n and m.pl = p
11    all n:Id           | some m:Msg | m.id = n and no m.pl
12    ... } }
13 check { (unique and complete) implies safety }
14  for 3 but 57 Msg, 10 steps
```

Fig. 2. Message-passing protocol in plain Alloy

Notice in passing that this semantics for record is also well-suited when using the popular trace exploration features[7] of Alloy: record signatures have a single possible valuation, so when exploring different configurations of the protocol, the user will not be encumbered by solutions that vary on available messages and actually represent the same configuration.

Evaluation, in Section 4, shows that despite increasing the size of the domain, our encoding is in fact more efficient than the typical ad hoc solutions employed at the Alloy level.

2.2 Example in plain Alloy

When modeling a system that handles record types, such as the example from Fig. 1, Alloy users would probably employ a similar structure but without the **`struct`** annotations, as depicted in Fig. 2. The first consequence of this is that records are no longer unique, and thus equality between atoms is not equivalent to equality of records. This can be forced by an additional constraint, such as `unique` (l. 3). The second consequence is that the user has to reason about scopes for records. To force every record to exist, one can define a constraint such as `complete` (l. 7) *and* set the scope of records to the maximum possible size, as in the **`check`** in l. 11. Notice how exact scopes on records *cannot* be enforced because the scope of the other signatures is also non-exact.

Remark that an alternative to a complete encoding is to carefully reason about the need for records during analysis, and perhaps end up with a tighter scope. For instance, if a protocol exchanges at most one message at each step, it will only ever require as many messages as steps, so the analysis could limit the scope of `Msg` to 10 (and remove the `complete` premise). Note, however, this leads

[7] Those allow to explore other static configurations, or initial states, or traces [1].

to cumbersome scenario exploration, since iterating over different configurations may just change the set of available messages.

Finally, a less flexible encoding than that of Fig. 2 is not to declare signatures standing for records but to use Alloy n-ary fields to represent them. For example, `Payload` would be replaced by `(Node → ValueA) + (Node → ValueB)`. However, the modeling of fields is cumbersome in this approach (especially when **`lone`** fields and hierarchies of records are allowed) and, more importantly, Kodkod relations corresponding to records are, again, not exact.

3 Introducing Records

3.1 Overview and Syntax

Records are specified using a new **`struct`** keyword applied as a signature modifier. The fields of a record type must be partial (resp. total) functions, *i.e.* they must be of arity 2 and have multiplicity **`lone`** (resp. **`one`**); they can be of any type excluding circular dependencies; and they may be declared mutable. Like plain signatures, records can be arranged in a tree-shaped record-type hierarchy, using the **`extends`** keyword, and they can be declared as **`abstract`**. A plain signature can also be declared as a subset of a record signature using the **`in`** keyword. Multiplicity constraints and bounds cannot be imposed on record signatures as their scope is automatically computed. Finally, a record signature can be referenced as any plain signature in the rest of the model.

3.2 Encoding and Semantics

Our extension relies on a specific encoding of records in Pardinus [5] (an extension of Kodkod [8]). Notice that in Kodkod, relations (incl. sets) are declared as taking any value between two sets: given a relation, the lower bound represents tuples that must exist in all valuations while the upper one represents those that may exist. When these are equal, the relation is said to be exact. The latter are important for performance because their value is computed before resolution. However, exactness of arbitrary relations cannot be specified in Alloy itself.

Our first key idea is then, for every concrete record signature, to translate it into an *exact* constant set of fresh atoms in bijection with the set of all combinations of *upper bounds* of its fields (*i.e.* some combinations may not exist in some states). Uniqueness and completeness are thus ensured by definition. The function rc computes the said set of records:

$$\begin{aligned}
&\mathtt{at(f{:}\ \textbf{one}\ R)} = \mathtt{rc(R)} \text{ if } \mathsf{R} \text{ is a } \mathtt{\textbf{struct},}\ \mathtt{up(R)} \text{ otherwise}\\
&\mathtt{at(f{:}\ \textbf{lone}\ R)} = \mathtt{at(f{:}\ \textbf{one}\ R)} \cup \{\mathtt{NOTHING}\}\\
&\mathtt{rc(\textbf{abstract struct sig}\ R\ \ldots\ \{\ \ldots\ \})} = \textstyle\bigcup \mathsf{rc(children(R))}\\
&\mathtt{rc(\textbf{struct sig}\ R\ \ldots\ \{\ \ldots\ \})} = \textstyle\bigcup \mathsf{rc(children(R))} \cup \pi_1(\mathsf{bij}(\prod \mathsf{at(fields(R))}))
\end{aligned}$$

Here, `up` returns the upper bound of a plain signature as in regular Alloy, and `at` returns atoms corresponding to a field; `NOTHING` is a distinct, dummy atom representing the empty assignment; fields yields all fields of a **struct**, including inherited ones; children returns the immediate children of a **struct**; and bij returns a set of fresh record atoms in bijection with its argument, concatenated with the argument itself (then π_1 returns the set of record atoms itself). Recursion is forbidden in record hierarchies so rc is well defined. Finally, mutability of fields does not change this computation. We also generate an *exact* binary relation, for every field, *projecting* every computed record atom to the corresponding field atom (we can retrieve the projections as bij keeps track of record atoms *and* their originating field values). Applying the function on Fig. 1, we get the following Pardinus declarations:

```
// Plain signatures yield sets given with lower, upper bounds:
Node   : {}, {(N0),(N1),(N2)} // low = {}, up = {(N0),(N1),(N2)}
ValueA : {}, {(VA0),(VA1),(VA2)}
var Id : {}, {(I0),(I1),(I2)}
// ... while records yield exact, pre-computed sets:
PayloadA = π1({(PA0,N0,VA0),...,(PA8,N2,VA2)})
         = {(PA0),...,(PA8)} // similarly for PayloadB
Payload  = rc(PayloadA) ∪ rc(PayloadB)
         = {(PA0),...,(PA8),(PB0),...,(PB8)}
Msg      = π1(bij(up(Id) × (rc(Payload) ∪ {NOTHING})))
         = {(M0),...,(M56)}
// ... and exact, pre-computed projections (for fields):
val = {(PA0,VA0),(PA1,VA1),(PA2,VA2),(PA3,VA0),...,(PB8,VA2)}
id  = {(M0,I0),(M1,I1),(M2,I2),(M3,I0),...,(M56,I2)}
...
```

As explained above, these exact sets represent the upper-bound of the record signatures but not their actual values, since field types are not necessarily exact or may change in some states. Our second key idea is therefore that, whenever a call to a record signature or one of its fields is made in the rest of the model, it must be filtered to exclude records that do not exist in the universe. Moreover, NOTHING values must also be filtered out to obtain the empty assignment. For a record signature R, this is done by identifying which of its fields are defined at each state, *using the inverse image of the corresponding projection*, and intersecting them with R. Similarly, fields are filtered w.r.t. the existing records on their domain and codomain. For instance, here, some of the replacements are:

```
Msg      ⇝ Msg & id.Id & pl.(PayloadA+PayloadB+NOTHING)
PayloadA ⇝ PayloadA & from.Node & val.ValueA
pl       ⇝ pl & (Msg & id.Id & pl.(PayloadA+PayloadB+NOTHING))
                 → ((PayloadA & from.Node & val.ValueA) +
                    (PayloadB & from.Node & val.ValueB))
from     ⇝ from & ((PayloadA & from.Node & val.ValueA) +
                  (PayloadB & from.Node & val.ValueB)) → Node
```

Notice this also works for mutable fields (like id), since the filter is always evaluated in the current state. Finally, all these filter expressions are simplified

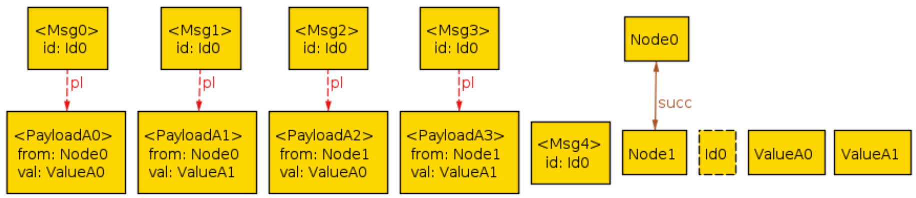

Fig. 3. Visualization of instances with records

if the binding expression can be shown to be exact. For instance, if the scope of `Node` is set **`exactly`**, we know that all possible node atoms are always present.

3.3 Visualization and Iteration

The Alloy visualizer has been adapted to identify record signatures: only filtered records are shown, they are represented with angle brackets, and plain fields are automatically shown as labels. Figure 3 shows an instance of the example from Fig. 1 in the visualizer. Also, all scenario exploration features keep their expected behavior. Note that since **`struct`** relations are exactly bound, scenario exploration is not hindered by alternative scenarios where only the set of available messages changes (although, of course, they will change if the signatures they depend on also change).

4 Evaluation

This section evaluates whether the performance of the **`struct`** encoding is feasible, particularly when compared with possible alternative approaches.

An extension previously proposed by Montaghami and Rayside [7] tried to address some of these issues. A signature modifier **`uniq`** is used to internally introduce generator axioms. **`uniq`** signatures are however restricted to have field types that are exactly bound, which is limiting since in Alloy we expect to explore alternative configurations. A staged approach is then used to first solve **`uniq`** signatures, which are passed as partial instances for the remaining problem. Two strategies are proposed to find the configuration: one cannot be applied when there are multiple configurations, the other requires solving the model for all possible configurations. Such a technique has been proposed in [6], where problems are decomposed between the static and mutable parts, and configurations analysed in parallel.

Table 1 summarizes the results of our evaluation for two message-passing protocols — the Paxos [4] consensus protocol and an Echo [2] protocol to form a spanning tree in a network[8] — where messages are seen as records. We considered two different unsatisfiable check commands for each model. Each entry shows

[8] The extended version of the Analyzer and all the models are available https://github.com/haslab/Electrum2/releases/tag/records-beta.

Table 1. Evaluation of Paxos and Echo, in seconds, best time in bold

model	*cmd*	*scp*	*msg*	*stp*	A_U	A_C	D_U	D_C	R	G
Paxos	`ChosenValue`	3	183	10	**124**	TO	TO	TO	357	0.3
	`ChosenValue`	3	183	11	633	TO	TO	TO	**527**	1.3
	`ChosenValue`	3	183	12	TO	TO	TO	TO	**1054**	–
	`OneVote`	3	183	7	**34**	TO	TO	TO	266	0.1
	`OneVote`	3	183	8	321	TO	TO	TO	**224**	1.4
	`OneVote`	3	183	9	2345	TO	TO	TO	**231**	10.1
Echo	`SpanningTree`	5	10	10	1172	322	TO	262	**4**	13.4
	`SpanningTree`	5	10	11	2523	945	TO	508	**11**	5.5
	`SpanningTree`	5	10	12	TO	1651	TO	1023	**33**	2.8
	`Finish`	5	10	9	745	219	TO	1798	**29**	7.5
	`Finish`	5	10	10	2679	314	TO	2815	**38**	8.3
	`Finish`	5	10	11	TO	405	TO	TO	**59**	6.8

the command executed (*cmd*), the default scope (*scp*), the maximum number of distinct messages (*msg*), and the steps scope (*stp*). Commands were run in a 2.3 GHz Intel 8th-gen Core i5 with 16 GB RAM with Glucose as the selected SAT solver, and time-out was set to 1 hour. The results **`struct`** extension are reported as R, with G being the relative gain to the best other approach. We also developed equivalent plain Alloy versions, enforcing uniqueness and completeness of records (A_C), and with as many messages as steps (A_U). To compare with a stage approach, we also analyzed those same models with the decomposed parallel strategy from [6] (D_U and D_C).

Evaluation showed that for R, although the solving stage is faster, there is an overhead during translation of the Alloy model to SAT (not shown in the table). Nonetheless, the approach still pays off, outperforming the plain Alloy analyzes as the number of steps increases. Compared with A_C, the approach with fine-tuned scope A_U performs better in Paxos than in Echo, which has a smaller number of messages. Regarding the decomposed strategy [6] with complete scopes D_C, it occasionally outperforms the regular Alloy analyses but is still worse than our approach; the decomposed strategy with incomplete records D_U always performs worse than the others for these commands.

5 Conclusion

We have implemented an extension of Alloy with records that enables a natural specification and has better performance than usual approaches in our experiments. In the future, we plan to evaluate bigger case studies and to assess the performance of an extension to more complex field types (sets or sequences).

References

1. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: Simulation under arbitrary temporal logic constraints. In: 5th Workshop on Formal Integrated Development Envi-

ronment. Porto, Portugal (Oct 2019)
2. Chang, E.J.H.: Echo algorithms: Depth parallel operations on general graphs. IEEE Trans. Software Eng. **8**(4), 391–401 (1982)
3. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edn. (2016)
4. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
5. Macedo, N., Brunel, J., Chemouil, D., Cunha, A.: Pardinus: A temporal relational model finder. J. Autom. Reason. **66**(4), 861–904 (2022)
6. Macedo, N., Cunha, A., Pessoa, E.: Exploiting partial knowledge for efficient model analysis. In: ATVA. LNCS, vol. 10482, pp. 344–362. Springer (2017)
7. Montaghami, V., Rayside, D.: Staged evaluation of partial instances in a relational model finder. In: ABZ. LNCS, vol. 8477, pp. 318–323. Springer (2014)
8. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS. LNCS, vol. 4424, pp. 632–647. Springer (2007)