A Comprehensive Study of Declarative Behavioural Modelling Languages

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

17-01-2022 / 24-01-2022

CITATION

Day, Nancy; Bandali, Amin (2022): A Comprehensive Study of Declarative Behavioural Modelling Languages. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.18551255.v1>

DOI

[10.36227/techrxiv.18551255.v1](https://doi.org/10.36227/techrxiv.18551255.v1)

A Comprehensive Study of Declarative Behavioural Modelling Languages

Amin Bandali, Nancy A. Day

Abstract—Declarative behavioural modelling is a powerful modelling paradigm that enables users to model system functionality abstractly and formally using constraints. There are several different formal declarative languages and they have greatly varying constructs for representing a transition system, and they sometimes differ in rather subtle ways. We qualitatively compare seven formal declarative modelling languages B, EVENT-B, ALLOY, DASH, TLA⁺, PLUSCAL, and ASMETAL with respect to three main categories: structuring transition systems (control modelling), data descriptions in transition systems (data modelling), and modularity aspects of modelling. We developed this comparison by completing a set of case studies across the data- vs. control-oriented spectrum of models in all of the above languages. We present recommendations based on our observations about the differentiating characteristics of each of these languages to help modellers answer the important question of which declarative modelling language may be most suitable for modelling their system.

Index Terms—Declarative behavioural modelling, transition systems, B, Event-B, Alloy, Dash, TLA+, PlusCal, Asmetal

1 INTRODUCTION

Formal modelling is useful for clarifying concepts in requirements and designs, and finding errors prior to downstream development efforts. Modelling languages allow a user to abstract away irrelevant details prior to coding and to analyze the model, ensuring that the problem is well understood, and that the proposed solution correctly serves the required purpose. In this article, we focus on formal declarative modelling languages for describing a behavioural model abstractly and concisely. Examples of such languages are Z [1], ALLOY [2], and TLA⁺ [3]. A **declarative formal model of a transition system** has the following characteristics:

- 1) Transitions are described in a declarative manner using constraints, rather than through imperative statements. This characteristic means the model is not necessarily executable (in the programmatic sense).
- 2) Data can be completely abstract, user-defined or user-axiomatized, which can represent rich and abstract datatypes in the model.
- 3) The language has a formal logical foundation, usually first-order logic (FOL) and/or set theory.
- 4) The model does not include the size (scope) of the sets. The scopes may need to be specified for analysis.

We are motivated to study the characteristics of declarative modelling languages by the many applications and demonstrated usefulness of these languages to help design systems or analyze and verify properties about the design of existing systems. Examples include Zave's use of ALLOY to find bugs in the specification of the Chord network protocol [4]; Newcombe's report on the use of TLA⁺ by engineers at Amazon which has helped find subtle bugs

in complex real-world systems and prevent the bugs from reaching production [5]; and the use of B by Huynh *et al.* for formalizing a new healthcare access control model with conflict resolution for managing patients' consent as to who can access their Electronic Health Records (EHR) in Québec [6].

Because there are many declarative languages to choose from, an important question is how does one make a choice of which language to use? The languages often vary in their capabilities in subtle ways. We address this question by presenting a qualitative language comparison criteria and comparing a number of popular declarative languages, highlighting the modelling of control, data and modularity aspects of models in each language. We arrived at this comparison by doing a number of case studies in multiple languages to expose the differences in paradigms, conventions, and structures among the languages. Our goal is to compare the language constructs that facilitate modelling (as abstractions of behaviour) rather than the theoretical expressivity of the languages.

We selected the seven popular, declarative modelling languages B [7], EVENT-B [8], ALLOY [2]¹, DASH [9], TLA⁺ [3], PLUSCAL [10], and ASMETAL [11] for comparison. We chose languages that have tool support for model checking. However, we are not comparing the languages on their tool support because the capabilities and performance of analyzers frequently change. This point of view is also shared by Lamport: TLA⁺ existed for several years before the TLA⁺ Toolbox was created for it.

In contrast to other work that compares declarative modelling languages (*e.g.*, [5], [12], [13], [14], [15]), we focus on models of transition systems. We develop a set of categorized comparison criteria and examine in depth

• Amin Bandali (bandali@uwaterloo.ca) was with the David R. Cheriton School of Computer Science, University of Waterloo.
 • Nancy A. Day (nday@uwaterloo.ca) is with the David R. Cheriton School of Computer Science, University of Waterloo.

Manuscript received April 19, 2005; revised August 26, 2015.

1. We use Alloy version 5. Version 6 includes Electrum and primed variables have special meaning so older Alloy models may not work in Version 6.

each language with respect to each criterion. Further, our case studies span a diverse range of examples on the data- vs. control-oriented spectrum of models. We model each of the six examples in all of the seven languages, producing a total of forty-two models. We use these models and our observations from carrying out the case studies to describe the capabilities of each language and to make recommendations as to which language(s) we think are the best fit for modelling various kinds of behaviour.

The contributions of our paper are:

- a set of criteria to compare declarative behavioural modelling languages;
- a diverse set of case studies on the data- vs. control-oriented spectrum of models, all rendered in seven languages;
- a comparison of the selected declarative modelling languages (B, EVENT-B, ALLOY, DASH, TLA⁺, PLUSCAL, and ASMETAL) based on the criteria; and
- our recommendations for the choice of modelling language based on the characteristics of the transition system under description, rooted in our observations of the differences and similarities observed in our comparison, along with recommendations for how declarative behavioural modelling languages can be improved.

We first present the methodology for our study (Section 2). Next, we describe some basic terminology (Section 3). In Section 4, we consider the *data modelling* aspects of each language and the structuring of the data descriptions of a transition system. In Section 5, we discuss the *control modelling* aspects in each language, meaning the structuring of a transition system in terms of individual transitions. In Section 6, we investigate the constructs relating to *modularity* of models in the languages and the structuring of a transition system on a larger scale in terms of files as well as subtransition systems. We present the constructs of the languages in the comparison as needed to illustrate the differences. Section 7 presents recommendations for the choice of declarative modelling language based on the characteristics of the transition system under description. We conclude with related work on declarative languages that we did not include in our study and other language comparisons. The goal of our work is to aid users in understanding how declarative languages assist in specify the appropriate level of description for a model and how to render these details in different languages.

2 METHODOLOGY

To qualitatively compare declarative modelling languages for writing a model of a transition system, we began by selecting six relatively small examples and modelled them in three declarative modelling languages (B, DASH, and TLA⁺), forming an initial set of comparison criteria for the languages and publishing our initial results [16].

We observed differences in the language’s abilities to express control-oriented aspects and data-oriented aspects. A model is more **control-oriented** if it has complex conditions for when a transition is relevant that are naturally expressed using modes and/or control states, or if its behaviour is naturally decomposed into concurrent (independent) parts.

A model is more **data-oriented** if it has abstract and complex specification of data decomposition and operations.

Thus, our next step was to expand both the set of languages we explored and the set of case studies. We dropped the smallest case studies from our original study, replacing them with larger, more realistic systems that span the data- vs control-oriented spectrum. We completed forty-two models (six models each in seven languages).

Table 1 shows the case studies and their sources considered in this work. For each case study (row), the number in each column indicates the order in which the model was completed in that language. We ordered our work such that every column includes either a 1 or an E, meaning that every language was used as the first language for at least one of the case studies or originated in that language. We then used that first version as the reference model for the other models of that case study in different languages. Our first three models of EHealth, digital watch, and musical chairs were completed concurrently (in our original study [16]), and thus are all labelled ‘1’ in the table. For the case studies where no reference model was available, we used the informal or semi-formal description of the system from the originating paper to write a first model, and used that model as our reference for the subsequent models of the case study. To avoid any bias, we assigned a random order to each model of each case study in each of the languages. We carried out the case studies top to bottom, starting with the EHealth system and ending with Railway, learning new languages as needed.

While doing the case studies, we observed and took notes about various characteristics of the modelling languages forming a comparison criteria. We later developed isolated exemplars, which are small examples meant to demonstrate the differences between languages with respect to a particular criterion. If a model needed updating (e.g. addition of new behaviour or properties *etc.*), we iterated through the versions of the model in all languages, updating those accordingly as well. Table 2 shows the sizes of the models in the common measuring unit Lines of Code². Fig. 1 shows the case studies with their control/data aspects relative to each other. All our models are publicly available³.

The equivalence of our models in different languages is accessed by observation, and by using a model checker and potentially other available tool support for each language to verify a similar set of properties about a model across all of the selected languages.

Our methodology was designed to limit bias towards one language or another when drawing our conclusions regarding language comparisons and recommendations based on them. For threats to validity, to minimize threats to internal validity, we randomized the order in which we carried out each case study across the languages. A possible threat to *external validity* is that we may be missing comparison criteria that could have arisen if we had done more case studies. To partially remedy this, we made sure to use a diverse set of examples across the data- vs. control-oriented characterization spectrum, and of different sizes. A possible threat to *construct validity* is that our recommendations may

2. We did not count headers and footers but we did count commonly used blank lines for separating sections of the model.

3. <https://github.com/WatForm/watform-models/tree/master/2020-bandali-thesis>

TABLE 1: Order of modelling case studies across languages

Language Case study	B	EVENT-B	ALLOY	DASH	TLA ⁺	PLUSCAL	ASMETAL
EHealth [17]	1	2	3	1	1	4	5
Library [13], [18]	E [13]	1	E [13]	3	5	2	4
Railway [19]	1	7	3	5	6	4	2
Digital Watch [20]	1	2	5	1	1	4	3
Musical Chairs [21]	1	4	E [22]	1	1	3	2
Landing Gear [23]	3	E [24]	4	2	5	1	E [25]

Legend: E indicates Existing models, *i.e.* those that we had no influence on. The numbers in each row indicate the order of the creation of models for the case study.

TABLE 2: Lines of code for each case study across the languages

Language Case study	B	EVENT-B	ALLOY	DASH	TLA ⁺	PLUSCAL	ASMETAL
EHealth	62	111	135	95	120	101	87
Library	180	164	317	120	146	151	207
Railway	82	86	387	280	79	84	78
Musical Chairs	68	84	130	65	101	106	97
Digital Watch	135	210	295	112	197	160	142
Landing Gear	300	302	483	265	395	200	298

Fig. 1: Our case studies, across the data- vs. control-oriented characterization spectrum

be biased by our previous knowledge of some language, as we are more familiar with some languages than others. To alleviate this bias, we have included multiple new languages and new case studies, that we were not previously familiar with. Another potential threat to construct validity is that not all of our models may follow all the idioms of the languages, due to the modeller being a novice user of some the languages. To partially remedy this, we use E models as references for our models of case studies whenever they were available to us. Further, our comparison criteria are very syntax-based, thus our comparison is not based on idioms and writing idiomatic models. Threats to *conclusion validity* are those concerning the validity of our conclusions and recommendations about the choice of modelling language. To address these, we devised an extensive set of comparison criteria to compare the languages against while carrying out the case studies, and used a diverse set of examples for our case studies based on which we make our recommendations.

3 TERMINOLOGY

As terminology varies from one language to the next, we first define some standard terminology that we will use throughout this article when comparing the seven languages. We say a construct is **explicit** in a language if the

language has a textual representation corresponding to that construct; and otherwise we say it is **implicit**, which means it is built/calculated by the tool support behind the scenes. The following are concepts in modelling a transition system:

- **Snapshot:** is a mapping of variables to values. These are called dynamic variables as their values can change in a trace of a transition system.
- **Snapshot space:** is the set of all possible snapshots of a transition system, *i.e.* the cross product of variable values.
- **Monitored variable/environmental variable:** is a variable that can be only observed by the model but not changed by it—they are only changed by the environment. Monitored variables are commonly used for modelling environmental phenomena such as temperature obtained from an input sensor.
- **Controlled variable/non-environmental variable:** is a variable that may be both observed and changed by the model.
- **Constant:** is an identifier with a value that retains the same value throughout all of the transitions of the transition system.
- **Transition System (TS):** is a tuple $TS = (S, TR, I)$ where S is a set of snapshots (a subset of the snapshot space), $TR \subseteq S \times S$ a transition relation, and $I \subseteq S$ a set of initial snapshots. A model in a declarative modelling

language defines a transition system that starts in an initial snapshot $s_0 \in I$ and progresses from a snapshot s to the next snapshot s' for $(s, s') \in TR$.

- **Trace:** is an execution path, consisting of a series of steps, where a **step** is a pair (s, s') of snapshots where every $(s, s') \in TR$ and for the initial (s, s') of the series $s \in I$. A transition system usually describes many traces.
- **Transition (T):** a set of pairs (s, s') each mapping from a source snapshot s to a destination snapshot s' . When a transition is defined using declarative constraints, it represents a set of pairs hence each $T \subseteq TR \subseteq S \times S$. An **enabled** transition is one that can be taken in a step.
- **Labelled control states:** are a distinguished finite set of identifiers that are used to control when a transition can be taken. Languages with labelled control states can have **control state hierarchy** (states/transitions within states) and **concurrency** (behaviour executing in parallel).
- **Frame problem:** refers to the issue of how snapshot variables that are not explicitly changed in the description of a transition may or may not change from one snapshot to the next. The frame problem is particularly an issue in declarative languages that rely on logical constraints on variables for describing the changed and unchanged variables in a transition. Since in all studied languages only one transition is taken per step, the frame problem can be discussed at the individual transition level (Section 5).
- **Invariant:** of a transition system is a formula that is true in all snapshots in all traces.
- **Fairness constraint:** is a condition on the traces of a transition system.
- **Stuttering step:** is often used to allow a change in the monitored variables but not the controlled variables so that transitions may become enabled. The semantics of stuttering steps—when they might occur, and which variables they allow to change and which ones they keep unchanged—can differ between two declarative modelling languages, and from one model to the next.

In most languages, a model describes a unique (although potentially non-deterministic) transition system. In Alloy, a set of formulas may describe a set of transition systems because the set of snapshots is explicit. However, in this article, we focus on methods of describing a unique transition system in these languages.

4 DATA MODELLING

In this section, we discuss in depth the description of the data aspects of models in each of the declarative modelling languages examined in this study. Data is constrained in the guards and actions of transitions and in the invariants of the model. Table 3 summarizes the most important differences in the data aspects of the selected declarative modelling languages. Fig. 2 helps compare the languages with respect to the description of data using an equivalent snapshot declaration across the languages.

Snapshot Declaration. A snapshot is a mapping from variables to values. In the examples of Fig. 2, the snapshot variables are $v1$, $v2$, $f2$, $R1$, and a set of events (evs)

and/or individual events $ev1$, $ev2$. In B, EVENT-B, TLA⁺, PLUSCAL, the section in the model for declaring variables is called **VARIABLES**. EVENT-B separates the transition relation (a **MACHINE**) from data specifications (a **CONTEXT**). In PLUSCAL variables can also be declared within processes. In ASMETAL, the variables are declared in the **signature** section, which may also include additional information. In ASMETAL, dynamic variables are classified into **monitored** and **controlled**. In ALLOY, a snapshot element can be declared anywhere, but typically they are explicitly declared as relations in a signature representing the snapshots of the model (in Fig. 2 this signature is called **Snapshot**). In DASH, snapshot elements are declared at the top of each **state** block. As DASH supports labelled control state hierarchies, the declaration of snapshot elements can be distributed through nested **state** blocks. Also, DASH has explicit syntax for marking a snapshot element as part of the environment, using the **env** keyword.

Primitives datatypes are the smallest unit of data in a language, and usually consist of scalars and sets (which include relations). In declarative modelling languages, similar to programming languages, scalars are objects like numbers, Booleans, strings, or user-declared objects. A set is an unordered collection of distinct objects. Sets are the fundamental building blocks for more complex units of data in abstract modelling. Fig. 2 shows the various ways sets are declared in the languages (e.g., **abstract domain** in ASMETAL and **SETS** in B and EVENT-B). In Alloy and Dash, sets are declared as signatures (e.g., line 2).

ALLOY and DASH are notable as languages that do not include scalars – everything in Alloy is a set, which reduces the set of operators needed (e.g., set membership becomes set inclusion). A scalar in ALLOY is represented using a singleton set. The lack of scalars means there are some unusual characteristics of the language. Formulas have Boolean values, but there is no Boolean set and no Boolean constants. The reason for the absence of Boolean-valued expressions is that their presence would render an expression like **not** p ambiguous when the set p contains zero or more than one Boolean. Instead, Boolean-like valuations can be mimicked using the **Boolean** set from the **util/Boolean** module of ALLOY's standard library. Alloy does support the use of integer scalars: 5 is syntactic sugar for the singleton set $\{5\}$.

Typically, the languages with scalars use the **POW** operator (for powerset) in the type signature to declare a snapshot variable as a set (e.g., line 9 of ASMETAL and line 15 of B). All of the languages with scalars provide the means to define a set by listing its members (thereby declaring these members as distinct constants, e.g., creating an uninterpreted/deferred set (B - line 4) or carrier set (EVENT-B - **partition** on line 17).

Constructors are operators that create composite units of data from primitives or other composite data units within a type signature. A **type signature** is syntax in a language to denote the kind of object contained in a snapshot variable, constant, or quantified variable. For example, functions, relations, and records are constructors. Multiplicities in a type signature impose constraints that limit the values in the composite data.

B is notable for having various functions/arrows (partial, total, bijective, *etc.*) as constructors similar to Z. For

```

1 // Alloy
2 sig A {}
3 sig C extends A
4 one sig c1 extends Const { }
5 one sig c2 extends Int { } { c2 = 3 }
6 enum Event { ev1, ev2 }
7 sig Snapshot {
8   v1: set A,
9   v2: one Int,
10  evs: set Event
11  f2: A some -> one B,
12  R1: C -> B
13 }
14 fact R1_multiplicity {
15   all a: A | #(a.R1) = 2
16 }
17 // transitions and TR follow ...

1 // AsmetaL
2 signature:
3   abstract domain A
4   domain C subsetof A
5   enum domain Event = {EV1 | EV2}
6   static c1: C
7   static c2: Integer
8   monitored evs: Powerset(Event)
9   controlled v1: Powerset(A)
10  controlled v2: Integer
11  monitored f2: Powerset(Prod(A, B))
12  controlled r1: Powerset(Prod(C, B))
13  derived issurfun:
14    Powerset(Prod(D1, D2)) -> Boolean
15  derived islto2:
16    Powerset(Prod(D1, D2)) -> Boolean
17 definitions:
18   function c2 = 3
19   function issurfun($f in Powerset(Prod(D1, D2)
20     )) = ...
21   function islto2($r in Powerset(Prod(D1, D2)))
22     = ...
23 // transitions go here ...
24 invariant inv_f2_surfun over f2: issurfun(f2)
25 invariant inv_r1_lto2 over r1: islto2(r1)
26 // TR goes here ...

1 // Dash
2 sig A {}
3 sig C extends A
4 one sig c1 extends Const { }
5 one sig c2 extends Int { } { c2 = 3 }
6 state Example {
7   v1: set A,
8   v2: one Int,
9   f2: A some -> one B,
10  env R1: C -> B,
11  env ev1, ev2 : event
12  ...
13  invariant {
14    all a: A | #(a.R1) = 2
15  }
16  // transitions and TR follow ...
17 }

1 /* TLA+ -- Pluscal nearly identical
2 EXTENDS Integers, FiniteSets
3 CONSTANTS A, C, c1, c2
4 VARIABLES v1, v2, evs, f2, R1
5 Event == {"ev1", "ev2"}
6 c2 == 3
7 Singleton(S) == {{i} : i \in S}
8 Range(f) == {f[x] : x \in DOMAIN f}
9 SurFun(a, b) == {f \in [a -> b] : b = Range(f)}
10 RelDom(R) == {x[1] : x \in R}
11 RelRan(R) == {x[2] : x \in R}
12 RelDomRes(S, R) == {x \in R : x[1] \in S}
13 OneToN(R, n) == \A x \in RelDom(R) :
14   Cardinality(RelRan(RelDomRes(R, {x}))) = n
15 TypeOK ==
16   /\ v1 \in SUBSET A
17   /\ v2 \in Singleton(Int)
18   /\ evs \in SUBSET Event
19   /\ c1 \in C
20   /\ c2 \in Int
21   /\ f2 \in SurFun(A, B)
22   /\ R1 \in C \X B /* \X is cross product
23   /\ OneToN(R1, 2)
24 /* transitions and TR follow ...

1 // B
2 SETS
3   A;
4   Event = {ev1, ev2}
5 VARIABLES v1, v2, evs, f2, R1
6 CONSTANTS isC, c1, c2
7 PROPERTIES
8   // subtype predicate
9   isC : A --> BOOL &
10  c1 : A &
11  isC(c1) = TRUE &
12  c2 : INTEGER &
13  c2 = 3
14 INVARIANT
15  v1 : POW(A) &
16  v2 : INTEGER &
17  evs : POW(Event)
18  f2 : A --> B &
19  R1 : A * B &
20  !(v).(v: A & isC(v) <=>
21    card(R1(v)) = 2 &
22    not(isC(v))
23    => card(R1(v)) = 0)
24 // transitions follow ...

1 // Event-B (context file)
2 CONTEXT
3   c0
4 SETS
5   A
6   Event
7 CONSTANTS
8   C
9   nC
10  ev1
11  ev2
12  c1
13  c2
14 AXIOMS
15  partition(A, C, nC)
16  card(C) > 0
17  partition(Event, {ev1}, {ev2})
18  c1 : C
19  c2 : INT
20  c2 = 3

1 // Event-B (machine file)
2 MACHINE
3   m0
4 SEES
5   c0
6 VARIABLES
7   v1
8   v2
9   evs
10  f2
11  R1
12 INVARIANTS
13  v1 : POW(A)
14  v2 : INT
15  evs : POW(Event)
16  f2 : A --> B
17  R1 : C <-> B
18  !v . v : C =>
19    (card({v} <| R1) = 1 &
20    card(R1[ {v} ]) = 2)
21 // transitions follow ...

```

Fig. 2: Snapshot Declaration Examples

TABLE 3: Summary of data aspects of languages

Language Criteria	B	EVENT-B	ALLOY	DASH	TLA ⁺	PLUSCAL	ASMETAL
Primitives	scalars, sets	scalars, sets	sets	same as ALLOY	scalars, sets	same as TLA ⁺	scalars, sets
Constructors	fun, rel, rec, multiplicity	fun, rel, multiplicity	rel, multiplicity	same as ALLOY	fun, rel	same as TLA ⁺	rel
Built-ins	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$, $\mathbb{B}$, str, seq, tree	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$, $\mathbb{B}$	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{B}$, str, seq, ord, graph	same as ALLOY	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$, $\mathbb{B}$, str, rec, seq, bag	same as TLA ⁺	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{B}$, char, str, seq, bag, map, Undef
Type signatures & typechecking	✓	✓	✓	✓	—	—	✓
Subtypes	—	✓	✓	✓	✓	✓	✓
Scopes	✓	✓	✓	✓	✓	✓	—
Events	—	—	—	event	—	—	—
Constants	CONSTANTS	CONSTANTS	✓	✓	CONSTANTS	CONSTANTS	static

Legend:

$\mathbb{Z}$: Integers, $\mathbb{N}$: Naturals, $\mathbb{N}_1$: Naturals excluding zero, $\mathbb{R}$: Reals, $\mathbb{C}$: Complex numbers, $\mathbb{B}$: Booleans.
fun: function, **rel**: relation, **rec**: record, **seq**: sequence, **ord**: ordering, **str**: string, **char**: character.

example on line 18 of the B fragment in Fig. 2, the arrow used means f_2 is a surjective function. R_1 on line 19 is a binary relation, with the additional 1-to-2 constraint imposed in a formula on lines 20–23. B also has records with named fields. More complex built-in constructors in B include (zero-indexed) sequences and trees. EVENT-B is similar to B but has three additional arrows (for total and/or surjective relations) for constructing specific kinds of relations, but EVENT-B lacks a record constructor.

ALLOY use relations as the composite unit of data, created using the ‘ $\rightarrow$ ’ arrow constructor (e.g., line 12). As relations are ubiquitously used in ALLOY models, ALLOY provides convenient dot join ‘.’ and box join ‘[]’ operators, which resemble record field access and array indexing respectively. Somewhat hidden to users of ALLOY is that ALLOY does not actually have record objects. Fields of signatures look like record fields, but signatures are actually sets and a signature field is a relation mapping its parent signature to the field’s value. Distinct to ALLOY are a range of **multiplicity constraints** within its type signatures. A multiplicity constraint is a constraint limiting the number of elements in the range associated with elements in the domain. ALLOY has four multiplicity keywords, **set** (any number), **lone** (zero or one), **one** (exactly one), and **some** (one or more).

ASMETAL’s primary composite data unit is relations, created using the **Powerset** and **Prod** constructors for creating sets and tuples respectively (e.g., line 12). Although ASMETAL has a dedicated function constructor arrow that can be used in type signatures, the possible operations on a function are limited.

TLA⁺ and PLUSCAL do not have type signatures, and thus no constructors; but formulas can constrain a set to be a function or relation (e.g., lines 15–23), which we describe later as part of typechecking.

The **built-ins** of a language are the names of particular primitives or particular composite data units that are part of the syntax of the language or part of its standard libraries, and are available to the modeller without declaration in the model. Most declarative languages have built-in primitive and complex datatypes such as integers and sequences. Notably, ALLOY’s built-ins include a constructor that imposes

linear total order on other sets. TLA⁺’s model checker TLC overrides the definitions of some of TLA⁺’s built-ins with implementations in Java for improved efficiency.

Typechecking refers to checking that the constraints expressed in the type signatures are consistent with the use of the data in the formulas. B, EVENT-B, ALLOY, DASH, and ASMETAL have type signatures, and TLA⁺ and PLUSCAL do not. In TLA⁺, any typing constraints must be stated with other invariants and are properties checked to hold in every snapshot of the transition system during the main analysis (as opposed to constraints that limit the model). An example of such typing constraints is the **TypeOK** predicate defined on lines 15–23 of Fig. 2. In our experience, having to manually add typing invariants in the absence of a type-checker can be error-prone and can hinder the debugging experience, as an error arising from the violation of these typing constraints can result in a cryptic error message.

In B and EVENT-B, type signatures are declared in the **INVARIANT** section along with other invariants (e.g., lines 15–23). B and EVENT-B use typechecking to statically catch type errors like assigning a value from a set to a variable with a type signature declaring a different/incompatible set, and applying a function to an argument not matching its type signature (prior to model checking). These languages can distinguish the typing constraints in the invariants from non-typing constraints in the invariants, and check them during the typechecking phase.

The Alloy Analyzer does a very limited kind of typechecking to ensure no expressions are redundant based on type signatures and that relations are of fixed arity. It is an important distinction that in ALLOY, the type signatures constrain the model’s reachable snapshot space (as opposed to being invariants that are verified to hold of the model).

Type signatures in ASMETAL are included along with each variable declaration. ASMETAL relies on several syntactic markers to deduce the expected type of various kinds of expressions, such as “12n” to mean “12” is a natural number.

A **subtype** is the name of a subset of values of another set. All the languages support subtyping except for B, where subtyping can be accomplished using membership predicates but this requires extra preconditions whenever the subtype is used. In the B example, a parent set A is

declared on line 3, and a constant predicate `isC` is declared and constrained on lines 6 and 9 to model the subtyping. `isC` must be used as a precondition for any use of the `R1` relation. ALLOY has the most expressive constructs for subtyping. To declare disjoint subsets of a set, the `extends` keyword is used, as on line 3. Non-disjoint subset signatures can be declared using the `in` keyword. Furthermore, in ALLOY a set can be `abstract` meaning that all of its elements are part of subsets.

An **event** denotes an occurrence at a moment in time, such as a button being pressed or a card swiped. Events are helpful in describing the abstract behaviour of reactive systems. An event may be a precondition of a transition. Among the declarative modelling languages studied in this work, events are a feature unique to DASH. An event in DASH can be added to a snapshot using the `event` keyword, as on line 11. To model events in languages other than DASH that do not have a notion of events built into them, a regular snapshot Boolean (or enumerated type) variable can be used to indicate if the events has occurred or not. Fig. 2 shows exemplars using this approach across the languages.

A **constant** is a mapping from a name to a value, either a primitive or a composite data unit. In contrast to a snapshot variable, a constant retains its value throughout all of the transitions of the transition system. B, EVENT-B, and TLA⁺ support the declaration of constants in a distinct part of the model. In ASMETAL, constants are declared under the `signature` section after snapshot variables, using the keyword `static`, as shown on lines 6 and 7 of the ASMETAL code block in Fig. 2. ALLOY does not have an explicit keyword or construct for constants. A constant is declared as a regular variable as a field of a signature (usually not the snapshot signature). In DASH, a constant is declared outside the DASH state hierarchy.

A **scope** is the size of a set of objects, *i.e.* the total number of distinct elements it contains. None of the languages studied in this work require specifying the scopes as part of the model. However, some of the languages allow the definition of scopes to be part of the model. ASMETAL is the only language studied here without a default set size; user-defined domains and their elements must be explicitly stated in the model. B, EVENT-B, TLA⁺, and PLUSCAL have language constructs for setting the scope, sometimes using set cardinality operators. In ALLOY and DASH, scopes are upper bounds or exact sizes of sets, and are specified for each analysis query. All top-level signatures have a default scope of 3.

5 CONTROL MODELLING

In this section, we discuss in depth the features provided in each language for modelling the control aspects of the transition system. In Table 4, we show a summary of the values of our comparison criteria for each language for the criteria where there are the most interesting differences. At the end of this section, we discuss how control modelling features can affect the possibility of creating an inconsistent model (meaning no transition system exists for the model). Fig. 3 shows exemplars defining a transition relation and transition in each of these three languages for a member join operation in our library case study.

Initialization. All the studied languages distinguish a snapshot element's initialization from its declaration, except for PLUSCAL which not only allows but recommends specifying the initial value of each snapshot element together with its declaration. Further, all the languages except ASMETAL support some form of nondeterministic assignment for initialization of snapshot elements.

Transition relation. Next, we discuss how each language allows a modeller to create the transition relation, TR , of a transition system. We begin with languages wherein the representation of TR is the most explicit, and follow with those where it is the least explicit. For each language, we examine its constructs for defining a transition relation, and how stuttering steps can be included in the model. Later, we examine each language's constructs for defining individual transitions that are composed together to form TR .

ALLOY, TLA⁺, and ASMETAL require a more explicit representation of a transition relation, where TR is defined directly in the model text. Because TR is described via a set of constraints, and not imperative definitions, a transition relation may be inconsistent.

In ALLOY, the transition relation, TR , is defined completely explicitly, and its form can vary greatly depending on how the snapshot and transitions are defined. For instance, with a `Snapshot` signature as the snapshot representation and its fields as variables, TR can be decomposed into predicates that can be viewed as transitions with before and after snapshots as arguments. The transition relation description for our example is on lines 20–26 of Fig. 3. Stuttering in ALLOY must be modelled explicitly, by writing a transition that constrains some or all of the variables to remain unchanged.

In TLA⁺, the transition relation is defined mostly explicitly. By convention, the transition relation is a predicate Next defined as the disjunction of all of the model's transition predicates (lines 8–13 of Fig. 3). If a transition predicate takes an argument, the argument is existential quantified ($\exists$) outside the transition predicate in the transition relation (line 10). In TLA⁺, one could write a model that is not a disjunction of the transition predicates, however, the TLC model checker for TLA⁺ is not optimized to handle such models, because it rewrites the transition relation as a disjunction of as many simple subactions as possible [26]. If TR is not a disjunction, two possible issues arise: First, from a debugging perspective, the visualizer will not show TR as a composition of smaller transition predicates, thereby making it harder to reason about the model and its behaviour. Second, from a model checking performance perspective, TLC spawns a worker thread for each subaction to explore the snapshot space, but because TR was not broken down into smaller subactions, TLC may not spawn an optimum number of worker threads. TLA⁺ models include stuttering steps between any two transitions implicitly (even when the system has not reached a deadlock). This implicit addition of stuttering to TR does not change the meaning of a model, since all TLA⁺ property formulas are invariant under stuttering (*i.e.* adding or removing stuttering steps does not affect whether or not a behaviour satisfies a temporal formula).

In ASMETAL, the transition relation is defined mostly explicitly using a special `rule` named `r_Main` (line 7). By

TABLE 4: Summary of control aspects of languages

Language Criteria	B	EVENT-B	ALLOY	DASH	TLA ⁺	PLUSCAL	ASMETAL
<i>TR</i> representation	Implicit	Implicit	Explicit	Implicit	Mostly Explicit	Implicit	Mostly Explicit
Stuttering	E	E	E	I(c) + E	I(a) + E	I(a) + E	I(c) + E
Control state hierarchy	—	—	—	✓	—	—	—
Concurrency	—	—	—	conc state	—	process	—
Transition Preconditions	PRE	guard	stylistic	events, guards	stylistic	await, when	—
Frame problem	unchanged	unchanged	may change	env	UNCHANGED	unchanged	monitored
Invariants	Property	Property	Constraint	Constraint	Property	Property	Property
Fairness	—	—	—	—	WF, SF	fair, fair+	JUSTICE, COMPASSION

Legend: **E**: Explicit. **I(c)**: Implicit with controlled variables unchanged. **I(a)**: Implicit with all variables unchanged.

convention, the main rule is broken down into smaller rules, each a transition, such as the one for the join transition shown on lines 2–5. The main rule specifies when and how each transition will be called. ASMETAL is more of an imperative-style language (though it does include various declarative, constraint-like expressions and rules, which is why we chose to include it in our set of languages). To write a transition relation that takes a non-deterministically-chosen enabled transition in each step (as in the other languages), we have to declare an enumerated set with each element corresponding to one transition. Then, we use the **choose** rule in the definition of the transition relation to choose an element from that set, and use a **switch** with a **case** for each transition, executing the transition corresponding to the chosen element of the enumerated set. This method can be cumbersome and error-prone in a model with a large number of transitions, since the modeller may forget to update the **switch** cases when adding or removing transitions. Stuttering in ASMETAL can be added explicitly, using the **skip** rule. Stuttering also occurs implicitly in the case of a deadlock, which happens when the set of snapshot elements updated by a transition is empty (*i.e.*, no assignments in the transition) or inconsistent (*i.e.*, there are conflicting assignments). This implicit stuttering in ASMETAL allows changes to the monitored variables by the environment, so as to enable potential further progress of the transition system [27]. This stuttering behaviour added implicitly to *TR* also ensures that *TR* is total.

The remaining languages — B, EVENT-B, PLUSCAL, and DASH — each have an explicit construct for defining transitions, which are implicitly composed together to form a transition relation. In each language, only one transition is allowed to be taken in each step, and all different interleavings of the enabled transitions in each step are represented in the traces of the model.

The transition relation in B, EVENT-B, and PLUSCAL is implicitly formed as follows: at any step, any transition whose precondition is satisfied (*i.e.*, is enabled) may be chosen to be taken. There is no requirement on the preconditions of the transitions to be non-overlapping, and more than one transition may be enabled at the same time, resulting in non-determinism in the transition system. In the remainder of this section, we will refer to this formation of *TR* as *TR_{IMP}*.

In B, the transition relation is implicitly formed, per

TR_{IMP}, and is a composition of transitions, referred to as operations in B, which may have zero or more preconditions, and one or more assignments (called substitutions) of values to snapshot elements as postconditions. A stuttering step in B must be represented explicitly, using the **skip** (identity) generalized substitution, which takes no action.

In EVENT-B, the transition relation is formed implicitly, per *TR_{IMP}*, and consists of transitions, referred to as events in EVENT-B, which may have zero or more preconditions, and one or more actions as postconditions. In EVENT-B, a stuttering step must be denoted explicitly, using a **skip** event, which is a transition that is always enabled (its guard is **TRUE**) and does nothing (it has no actions).

In PLUSCAL, the model consists of a set of processes that run concurrently. The number of copies of each process type is chosen at analysis time. The transition relation is formed implicitly, per *TR_{IMP}*. One transition per process is taken in each step. A stuttering step in a PLUSCAL algorithm may be represented explicitly in a process, using the **skip** atomic instruction, or occur implicitly, between any two PLUSCAL steps (defined in the transitions subsection below).

In DASH, the transition relation is formed implicitly following the semantics of concurrent, hierarchical state machines (as in Statecharts [20]). Within Fig. 3 on line 2, there is the state declaration for the concurrent state **Library**⁴. The states can be arranged in a hierarchy of AND and OR states (represented textually using nesting). AND-states represent concurrent behaviour and OR states decompose behaviour. For a transition to be taken, the snapshot must include the source state of the transition, and transitions exiting states at a higher level in the hierarchy have priority over lower states. Because of the concurrency, DASH makes the distinction between big steps and small steps in the transition relation so that one big step can consist of the reaction of multiple concurrent states to an input. Big steps consist of multiple small steps, which are each one transition. In a big step, at most one transition per concurrent region can be taken. Monitored (**environmental**) events can change only at big step boundaries (called a stable snapshot), so the occurrence of a monitored event can trigger multiple transitions as long as the transitions are in different concurrent regions. Events generated by one transition can trigger other transitions within the same big step. The entire big

4. The root state is always a concurrent state even if there is only one component state.

```

1 // B
2 Join (newmember_) =
3   PRE
4     newmember_ : MEMBERID &
5     newmember_ /: member
6   THEN
7     member := member \/ { newmember_ }
8   END;
9 // ... (more transitions)

1 \* TLA+
2 Join(newmember) ==
3   /\ newmember \in MemberID
4   /\ newmember \notin members
5   /\ members' = members \union {newmember}
6   /\ UNCHANGED << books, loans, reservations >>
7 \* ... (more transitions)
8 Next ==
9   ...
10  /\ \E m \in MemberID:
11    \/ Join(m)
12    \/ Leave(m)
13    ...

1 // Alloy
2 // Lib is the snapshot sig
3 pred CanJoin[newmember:Member,L:LibSnapshot] {
4   // newmember does not exist in the Library.
5   no (newmember & L.members)
6 }
7 pred Join[newmember:Member, L,L':LibSnapshot]{
8   ----Precondition-----
9   CanJoin[newmember, L]
10  ----Postcondition-----
11  // add the m in the set of members
12  L'.members = L.members + newmember
13  ----Nochanges-----
14  NoChangebooks[L, L']
15  NoChangeLoan[L, L']
16  NoChangeSeqBook[L, L']
17  NoChangeRenew[L, L']
18 }
19 // ... (more transitions)
20 pred tr[] {
21   all l: Lib - LibOrd/last |
22     some m: Member |
23       Join[m, l, l.LibOrd/next]
24     or Leave[m, l, l.LibOrd/next]
25     or ...
26 }

1 // Dash
2 conc state Library {
3   trans join {
4     when !(newmember in members)
5     do members' = members + newmember
6   }
7   // ... (more transitions)
8 }

1 // Event-B
2 Join:
3   ANY
4   newmember
5   WHERE
6     newmember : MemberID not theorem
7     newmember /: members not theorem
8   THEN
9     members := members \/ {newmember}
10  END
11 // ... (more transitions)

1 \* PlusCal
2 process Join = "Join"
3 variable newmember \in MemberID
4 begin
5   join:
6     when newmember \notin members;
7     members := members \union {newmember};
8     goto join
9 \* ... (more transitions)
10 end process

1 // AsmetaL
2 rule r_Join($newmember in MemberID) =
3   if (notin(members, $newmember)) then
4     members := union(members, {$newmember})
5   endif
6 // ... (more transitions)
7 main rule r_Main =
8   choose $b in BookID, $m in MemberID, $rule in
9     RuleId with true do
10     switch($rule)
11     ...
12     case JOIN:
13       r_Join[$m]
14     ...
15   endswitch

```

Fig. 3: Transitions (from the Library case study)

step is considered the model's response to an environmental input. Implicit stuttering in DASH happens only at the big step boundaries, when no more transitions may be taken. To enable stuttering when one or more transitions are enabled, an explicit stuttering transition must be used.

Transitions and the Frame Problem. Next, we describe how each language can represent an individual transition T of a transition system. As we are working with declarative modelling languages, each transition describes a set of pairs $(s, s') \in T$ (rather than one pair). To create a transition $T \subseteq TR \subseteq S \times S$ a language may use two-snapshot predicates, assignment operators, or both. A predicate corresponding to a transition T describing a set of pairs $(s, s') \in T$ can be broken down into pre_T on the source snapshot and $post_T$ on the source and destination snapshots, combined using

some logical connective $\underline{?}$:

$$\begin{aligned}
 T(s, s') &= [pre_T(s)] \underline{?} [post_T(s, s')] \\
 &= [guard_T(s) \wedge src_T(s) \wedge evt_T(s)] \underline{?} \\
 &\quad [act_T(s, s') \wedge dest_T(s') \wedge genevt_T(s')]
 \end{aligned}$$

The transition may only be taken when pre_T is true. Each of pre_T and $post_T$ may in turn consist of one or more explicit language constructs. The preconditions pre_T of the transition may consist of a guard $guard_T$, a source labelled control state src_T , and triggering event evt_T ; and the post-conditions $post_T$ may consist of actions act_T , destination control state $dest_T$, and generated events $genevt_T$, constraining the snapshot elements after the transition is taken. We will describe the languages in increasing order of explicit

language constructs for describing a transition. The **frame problem** [28] is the question of whether snapshot elements keep their values from before the transition if not explicitly changed in the transition.

ALLOY does not have a special construct for defining transitions. Commonly, a transition is modelled as a predicate over unprimed and primed state variables, constraining the value of the snapshot elements in the source and destination snapshots. The predicate used to describe the join transition is on lines 7-18 of Fig. 3. Unlike the other languages, primed variables do not carry any special meaning in ALLOY, and are used by convention⁵. Though a transition may be represented using a single predicate, Farheen *et al.*'s guidelines for Alloy models of transition systems [29] separate a transition definition into two separate predicates—one for preconditions over the variables in the source snapshot, and another for postconditions over the variables in the source and destination snapshots—to promote structure. Commonly, the pre and post predicates for each transition are conjoined together to form the main transition predicate (*i.e.*, $_?$ is $\wedge$), and the transition predicates are then disjoined together to form the transition relation. This is referred to as the *disjunctive modelling method* in Farheen *et al.*'s guidelines. The guidelines recommend this method for decomposing the transition relation because adding a transition does not change the behaviour of other existing transitions, and is thus more likely to produce a transition relation expected by the modeller. With respect to the frame problem, in ALLOY any variable not constrained in a transition predicate may change nondeterministically from the source to the destination snapshot, and there is no distinction between monitored and controlled variables. Thus, frequently Alloy models include lines such as lines 14–17, which call predicates that constrain variables to retain their values.

In TLA⁺, a transition (an *action* in TLA⁺ terminology) is a two-snapshot predicate implicitly over unprimed and primed versions of all snapshot elements. The join transition's description in TLA⁺ can be found on lines 2–6. Similarly to ALLOY, in TLA⁺ a transition may be further broken down into separate pre and post parts for clarity. The pre and postconditions are often conjoined together, *i.e.* $_?$ is $\wedge$. With regards to the frame problem, TLA⁺ requires that all transitions constrain the value of every snapshot element, either by constraints on the primed and unprimed variables or by marking them with the **UNCHANGED** keyword. TLA⁺ does not make a distinction between monitored and controlled variables.

ASMETAL does not have any construct for decomposing a transition into separate pre and post parts; and a transition definition consists only of an action—a *rule* in ASMETAL terminology. Lines 2-5 describe the join transition in our example. A transition in ASMETAL does not have preconditions on when it may be taken—all transitions of a model are always enabled. Conditional rules such as **if** and **case** are used to constrain the action(s) of the transition. An action may be one of several rules, producing a set

of assignments to controlled variables. The transition rule may be a simple rule such as the **skip** rule or the update rule (*i.e.*, variable assignment, using the $:=$ operator), or a more complex rule such as a **parallel** or **sequential** block, which enables composition and combinations of other rules. In ASMETAL, for $(s, s') \in T$ the transformation of s to s' is defined as the effect of applying as an atomic step the result of a consistent update set (one with no conflicting variable assignments) on s . With respect to the frame problem, any controlled snapshot element not assigned to in a transition is unchanged by that transition, and retains its value from the source snapshot.

In B, a transition is referred to as an *operation*, each consisting of one or more actions—called a *generalized substitution*—which may take several forms. Lines 2-8 show the B description of the join transition. An action may be a compound substitution—such as a block substitution (**BEGIN** S **END**) or a preconditioned substitution (**PRE** P **THEN** S **END**) with precondition P required to be true before calling the operation (otherwise the operation will not be enabled and cannot be executed) and S the body of the substitution. The modeller may refer to the value of the variable in the source snapshot using the $\$0$ suffix (in a non-assignment statement). For example, $x\$0$ would refer to the value of x in the source snapshot, and x to its value in the destination snapshot. With respect to the frame problem, in B any snapshot element not assigned to in a substitution retains its value and is unchanged by that transition. B does not distinguish between monitored and controlled snapshot elements.

In EVENT-B, a transition is referred to as an *event*, and is a simplification of a B *operation*. Lines 2-10 show the B description of the join transition. In EVENT-B a transition has only one general form, consisting of one or more of the following parts: parameters, guards, and actions. The parameters are found in the **ANY** clause. A *guard*, defined in the **WHERE** clause of the transition and the only form of precondition in EVENT-B, is the precondition required to be true for the transition to be enabled. An *action* in the **THEN** clause, consisting of a possibly non-deterministic assignment, describes how the source and destination snapshots relate, and is a simplification of B's generalized substitutions. Regarding the frame problem, similarly to B, in EVENT-B any variable not assigned to using an assignment operator is unchanged by the transition and retains its value. No distinction is made between monitored and controlled variables in EVENT-B.

In PLUSCAL, the language's constructs can express concurrency. A transition is a single step (atomic action) in one process consisting of the statements between one *label* and the next. Line 5 uses **join** to label the beginning of the join transition's description. Preconditions of a transition can be described using an **await** (or **when**) statement which acts like a guard, allowing the process body to be executed only when the guard expression evaluates to true. A label can be considered a non-hierarchical control state. PLUSCAL has a **goto** statement to jump to a label. In a process body, snapshot elements may be assigned a value using the $:=$ operator, and will otherwise retain their value from the source snapshot. PLUSCAL does not distinguish between monitored and controlled variables. Compared to TLA⁺,

5. In Alloy 6, priming of variables has a specific meaning so a different naming convention would have to be used. We continue to use the primed variable in this article to match existing literature on Alloy.

PLUSCAL has a number of safeguards in place that make certain classes of bugs syntactically invalid, thus allowing the parser to catch them when the modeller asks for the TLA⁺ translation of their PLUSCAL model [10]. Examples include mistakes such as multiple assignments to the same variable in a transition, and having unreachable statements.

A transition in DASH is defined using the **trans** keyword, optionally consisting of the parts described below. Line 3 is the beginning the description of the join transition in Dash. The precondition of a DASH transition is divided into three parts:

- 1) A guard condition of the transition, denoted using the **when** keyword, which is an ALLOY expression over the snapshot variables.
- 2) A source labelled control state of the transition, denoted using the **from** keyword. Since DASH has explicit control state representation using **state** blocks, labelled control state blocks may be nested to form a control state hierarchy, useful for grouping related states together. In the absence of an explicit **from** part, the most immediate state containing the transition definition will be used as the source labelled control state of that transition.
- 3) An event triggering the transition, denoted using the **on** keyword, for triggering the transition whenever a certain event occurs.

The postcondition of a DASH transition is divided into three parts:

- 1) The actions of the transition, denoted using the **do** keyword, are two-snapshot ALLOY predicates over unprimed and primed variables, modelling the effects of executing the transition.
- 2) A destination labelled control state of the transition, denoted using the **goto** keyword. In the absence of an explicit **goto** part, the most immediate state containing the transition definition will be used as the destination labelled control state of that transition.
- 3) The generated events, denoted using the **send** keyword, for firing one or more events upon the execution of the transition.

The tool support for DASH performs several well-formedness checks such as checking environmental events cannot be generated by a transition and all child states at the same level of the hierarchy are either all concurrent or not concurrent. With respect to the frame problem, monitored variables (marked with **env**) in DASH are allowed to change from one snapshot to the next when the next snapshot is stable (*i.e.* at the big step boundaries), and will otherwise retain their values from the source snapshot. For controlled variables, if the primed version is mentioned in the action of a transition, it is assumed that the action will constrain it; otherwise, they are forced to retain their value from the source snapshot.

With respect to **expressions/formulas** used to describe transitions, all of the languages are based on first-order logic and set theory and their expressions are reasonably similar in succinctness. They all provide operations to create and modify primitives and composite data units. All of the languages except ASMETAL support nondeterministic initialization and assignment of variable values.

The expressions of B and EVENT-B have a more imperative style than declarative, in that initialization and changing of variables are usually done using the **:=** assignment operator. Additionally, both languages have a nondeterministic assignment operator **⋈** for setting the value of a variable to a nondeterministically-chosen element from a set. When a regular assignment is not flexible enough, the value of the left-hand side variable can be constrained declaratively with a two-snapshot predicate on the right-hand side using the **⋈** operator in B and the **⋈** operator in EVENT-B. In B, assignment statements are combined using the **||** and **;** operators, for parallel and sequential assignments respectively.

ALLOY and DASH have expression languages that include set operators mixed with first-order logic. Models typically make extensive use of the join operator because every unit of data is a relation.

TLA⁺ has a small number of essential operators for conveniently working with sets and functions, TLA⁺ also has the **CHOOSE** operator, **IF . . THEN . . ELSE** expressions, **CASE** expressions, **LET . . IN** expressions, and **LAMBDA**. Even though TLA⁺ is a very small language, one can easily define new custom operators. TLA⁺ uniquely supports recursive and higher-order operators, which take other operators as arguments.

The ASMETAL language rarely uses symbols and most operators have long-form textual names. ASMETAL's built-in units of data are opaque, which means that the operations on the built-in composite data units are limited by what is currently implemented in the tool support for the language. As of now, in ASMETAL sets are the most flexible/featureful construct in the language, hence, use of the special arrows may be limiting.

With respect to formalizing **decision logic**, all languages except EVENT-B support at least one form of conditional expressions. B has **IF** and **CASE** conditional substitutions. ALLOY and DASH have **implies-else**. TLA⁺ and PLUSCAL have **IF-THEN-ELSE** and **CASE** expressions and PLUSCAL additionally has **if** statements as well. ASMETAL has **if-then-else** terms and rules. In EVENT-B, in the absence of conditional expressions, one may use event guards for making the execution of certain actions conditional, though for modelling decision logic for complex behaviours this would certainly not be as flexible and convenient as using if-then-else or case expressions for branching.

None of the languages explicitly support operators for **dynamic allocation** of parts of the state (*i.e.*, “new” in Spin [30]). Further, none of the languages have a message-passing **channel** feature like Spin does, and such a construct must be modelled using snapshot variables and techniques ensuring correct synchronization. Also, none of the languages have a built-in construct for **time**; but it can be modelled using techniques such as the “explicit-time” approach explained in [31], [32], wherein the current time is represented as a snapshot variable, and the passage of time is modelled using a transition that increments the value of that variable. We have not discussed some macro-like features of the languages. For example, ALLOY has predicates and functions and DASH has transition parametrization and transition comprehension to describe many transitions in one line (such as every state has a transition to an error state

on a failure event).

Invariants. When modelling a transition system, it can be useful to include a formula that is always valid, called an invariant. In B, EVENT-B, TLA⁺, PLUSCAL, and ASMETAL, invariants are written in a particular section of the model. The invariants are checked as properties that must hold of the model. In ALLOY and DASH, invariants are written in **fact** blocks. DASH additionally has **invariant** blocks. In both languages, invariants are constraints on snapshots and limit the reachable snapshot space of the model.

Fairness Constraint. A fairness constraint [33], [34], [35] is a condition on the traces of a model. A trace is said to be fair if and only if it satisfies the fairness formula. In a declarative modelling language, fairness constraints can either be used to limit the model to include only fair traces, or as part of a property to limit the traces on which a property is evaluated. In B, EVENT-B, ALLOY, and DASH there is no way to express fairness properties that limit the traces of the model⁶.

In TLA⁺, weak or strong fairness for transitions can be added to the model. If a transition with weak fairness (justice) is always enabled, it will eventually be taken. If a transition with strong fairness (compassion) is enabled infinitely often, it will eventually be taken. In PLUSCAL, a process can be marked as weakly fair or strongly fair. Further, for a PLUSCAL label A : in an unfair process, $A: +$ marks the label as weakly fair; and for a label A : in an weakly fair process, $A: +$ makes it strongly fair. In both cases, $A: -$ marks the label as unfair.

In ASMETAL, for a Boolean formula $\mathbb{f}$, **JUSTICE** $\mathbb{f}$ imposes weak fairness and requires that $\mathbb{f}$ be true infinitely often in the model. For two Boolean formulas $\mathbb{f}$ and $\mathbb{g}$, **COMPASSION** ($\mathbb{f}$, $\mathbb{g}$) requires that if $\mathbb{f}$ is true infinitely often then $\mathbb{g}$ be true infinitely often as well⁷.

Inconsistency. In declarative models, inconsistencies stem from contradictions in logical formulas. All languages allow inconsistencies. We consider three kinds of inconsistency that may occur in declarative models:

- **Contradictory transition postcondition:** when the logical formula corresponding to the postcondition of a transition is a contradiction. One possible form of contradictory transition postconditions is simultaneous assignment to the same variable in a transition. Having a contradictory transition postcondition is possible in all languages.
- **Contradictory transition relation:** when the logical formula corresponding to the transition relation, TR , is a contradiction (*i.e.*, is never true), and therefore there does not exist a snapshot trace satisfying the constraints of TR 's definition. In B, EVENT-B, PLUSCAL, and DASH, one cannot create a contradictory TR , since the transition relation of the model is created implicitly from the elements described by the modeller.
- **Deadlock:** refers to a snapshot that has no explicitly modelled outgoing transitions. In the absence of stuttering steps in a declarative modelling language, this

means a non-total transition relation. DASH, TLA⁺, PLUSCAL, and ASMETAL perform implicit stuttering steps when a transition system reaches a deadlock, thereby ensuring the totality of the transition relation and enabling potential further progress of the transition system. Another method for ensuring the totality of a transition relation is making the transition preconditions complete; *i.e.*, the disjunction of the preconditions of all transitions is true.

The interpretation of a deadlock in ASMETAL depends on the tooling used: for instance, the interactive runner for Asmeta considers the deadlock snapshot the end of execution and will terminate the run when that snapshot is reached, whereas the Asmeta animator will **stutter** infinitely when the deadlock state is reached. In B and EVENT-B, a transition with a contradictory postcondition (*e.g.*, an action/substitution like $x := \emptyset$ that is always false) is effectively never enabled (same as a transition with a false precondition), and the transition system would thus reach a deadlock if no other transition is enabled. Both B and EVENT-B employ static checks as part of their type system to catch various modelling errors like type mismatches (*e.g.* $a = \emptyset$ in the above exemplar) or simultaneous assignment to the same variable (*e.g.*, $a := a1$; $a := a2$) in the same transition. PLUSCAL has safeguards to make certain kinds of inconsistency unrepresentable in valid models. For instance, each labelled section may only contain at most one assignment statement for each variable.

6 MODULARITY

To create large models, some form of modularity is needed. In this section, we discuss how each language provides support for modularity in a description of a transition system. Modularity can be evaluated from two points of view:

- (1) How can a description of a single transition relation be decomposed into multiple parts? This decomposition may take the form of
 - (a) subtransition systems that are composed to create the single top-level transition relation implicitly or explicitly, where a subtransition system is a full description of a transition system; or
 - (b) subformulas relevant to other aspects of the model—such as a grouping information (axioms, *etc.*) regarding a unit of data—which we refer to as a data decomposition.
- (2) How can a description of a single transition system model be decomposed into multiple files? Typically, a decomposition into multiple files involves interfaces that can be parameterized and have private and public parts.

It may be possible for the decompositions of (1) to be realized in multiple files. In the following, we describe the different means within a language of decomposing the transition relation (*i.e.*, (1) above) and within this discussion address when/how different parts of the transition relation can be realized in multiple files. Both (1) and (2) may involve differing namespaces. A summary of our comparison on modularity can be found in Table 5.

6. In Alloy 6, linear temporal logic (LTL) operators are introduced which would allow the expression of fairness properties.

7. The missing closing parenthesis is intentional and part of the ASMETAL grammar.

TABLE 5: Summary of modularity aspects of languages

Language Criteria	B	EVENT-B	ALLOY	DASH	TLA ⁺	PLUSCAL	ASMETAL
sub <i>TR</i> s	✓	—	✓	—	✓	—	✓
sub <i>TR</i> namespaces	—	—	—	✓*	—	—	—
Data decomposition into multiple files	✓	✓	✓	✓	✓	✓	✓
File import	SEES , ...	SEES	open	open	EXTENDS	EXTENDS	import
File export	varies	entire context	non- private	—	non- LOCAL	non- LOCAL	export
File parameterization	✓	—	✓	✓	✓	✓	—
File namespaces	—	—	✓	—	✓	✓	—

*: Due to DASH's per-**state** namespaces.

In B, the definition of the transition relation is implicit, and it is possible to decompose a transition system into subtransition systems. Each B machine resides in one file, under one namespace. The B language has extensive constructs for creating relationships between machines, using the **SEES**, **INCLUDES**, **PROMOTES**, **EXTENDS**, **USES**, and **IMPORTS** keywords [36].

In EVENT-B, the definition of the transition relation cannot be further decomposed into subtransition systems. In EVENT-B, a model consists of contexts (potentially in multiple files) and a machine (in one file). Contexts describe non-changing parts of the system, namely the sets, constants, and axioms; and a machine contains the snapshot and transitions (the transitions are implicitly combined to build the transition relation). A machine can *see* any number of contexts by adding their names under its **SEES** section (e.g., line 5 of Fig. 2).

In ALLOY, the definition of the transition relation is explicit as a formula. Therefore any means in ALLOY of decomposing formulas can be used to decompose the description of a transition system. Formula decomposition can be split across multiple files and imported into the main file. This decomposition could be for subformulas of the transition system or for a grouping of axioms regarding a unit of data. ALLOY refers to each file as a module, which has its own namespace and may have private elements. In ALLOY a module can be parameterized.

In DASH, the definition of the state hierarchy for the transition relation must all reside together in one file because the transition relation is defined implicitly. Separate namespaces in the state hierarchy create interfaces between states, while still allowing global communication. A reference to an identifier in another state must be given by its fully qualified name. In DASH, data modelling can be described separately from the state hierarchy and therefore can be separated into multiple files as in ALLOY.

In TLA⁺, the definition of the transition relation is explicit as a formula. Therefore any of the means of decomposing formulas in TLA⁺ can be used to decompose the description of a transition system. Within one TLA⁺ file, the namespace includes all of the identifiers declared/defined in the current file. TLA⁺ refers to each file as a module. It is possible to import the definitions of a module into a dedicated namespace, as well as marking a definition or an instantiation as local to the current module.

In PLUSCAL, a transition system is defined in an

algorithm block. The definitions of the transitions must all reside together in the algorithm in one file because the transition relation is defined implicitly. Each process may have local variables only available to that process. In PLUSCAL, the data modelling aspects of a transition system can be described separately from the transition definitions, and therefore can be separated into multiple files as in TLA⁺.

In ASMETAL, the definition of the transition relation is explicit as an ASMETAL **rule**. Therefore any of the means of decomposing rules in ASMETAL can be used to decompose the description of a transition system. In ASMETAL, a model can be split across multiple files for subtransition system and data decomposition. ASMETAL refers to each file as a module, and there are two kinds of modules: modules containing a transition system, and modules that do not. Identifiers of another module can be imported selectively and a module can hide some of its identifiers. ASMETAL does not support prefixed/namespaced imports, and there is no support for parametrized modules.

In summary, the B, ALLOY, TLA⁺, and ASMETAL languages allow decomposition of the model into subtransition systems across multiple files, while EVENT-B, DASH, and PLUSCAL only allow data decomposition across multiple files.

An orthogonal perspective on the notion of modularity in B and EVENT-B is the concept of *refinement* [7], [8], [36], [37], and both B and EVENT-B have tool support dedicated to assisting the modeller with performing correct refinement steps.

7 RECOMMENDATIONS

When choosing a modelling language there is often a trade-off between choosing a language whose available structures nicely match the characteristics of the problem versus choosing a language with great flexibility to express any behaviour. For example, the Dash language provides a language construct for concurrent states, which can be mimicked in the very flexible TLA⁺. However, using a language construct to succinctly indicate concurrency helps us to decompose the problem and without it the model may be much harder to understand. In this section, we make recommendations regarding which language structures are most easily represented in other languages. Our goal in making these recommendations is to 1) help a user decide which language to choose because the problem includes

behaviours that are difficult to represent in some languages; and 2) help language designers to think about which behaviours it is worth supporting with explicit structures in the language.

Transition Relation: The languages with implicit transition relations and explicit constructs for transitions save text and improve understandability in a model compared to the more general languages of ALLOY, TLA⁺, and ASMETAL. In our case studies, we found the implicit definitions of the transition relations used in the languages to all be suitable for the wide range of problems we modelled. However, a key concern is how the modeller wants to include stuttering (or not). With an explicit transition relation, the user can model any kind of stuttering.

Subtransition Systems: Few of the languages had good support for decomposing a model into multiple sub transition systems in different files. While data decomposition is supported in all languages, more support for describing larger transition systems with different namespaces is needed for larger models.

Concurrent, hierarchical, and replicated states: DASH has concurrent and hierarchical states to explicitly capture concurrency and hierarchical decomposition of control states. These states can be arbitrarily nested. PLUSCAL has support for replicated, concurrent components but its processes must all be at the top-level of the model. In both DASH and PLUSCAL, the communication between components is global rather than directed. None of the languages support buffered communication, which is commonly used with replicated components for protocol modelling (*e.g.*, Spin [30]). Mimicking control states in the other languages can be accomplished through the use of snapshot elements whose values are the names of the control states. Mimicking state hierarchy is done with explicitly constraints to relate snapshot elements for the control states. Mimicking concurrency is much more complicated to model in a language that does not support it depending on the semantics of the concurrency. To model replicated concurrent components involves a combination of mimicking concurrency and parametrizing many snapshot elements by a process identifier as well as a snapshot. The DASH to ALLOY translator described in Serna [9] shows how much DASH abbreviates the ALLOY constraints needed to describe this kind of behaviour generally. Thus, if state hierarchy and concurrency are an important abstraction for a system, DASH and PLUSCAL are better choices than the other modelling languages we studied.

Complex decision logic: All the languages except EVENT-B support some kind of conditional statement for describing decision logic. Modelling languages for reactive systems such as SCR [38] and RSML [39] use tables to describe decision logic in a structured format (but cannot represent data operations as abstractly as in declarative languages). We particularly noticed the lack of support for decision logic in modelling the railway system where tables of data must be represented in the model. Support for modelling complex decision logic is an area that needs more support in declarative languages.

Frame Problem: One of the main differentiating factors between the languages studied in this work is their behaviour with respect to the frame problem. In our case stud-

ies, we found that the languages that required the modeller to explicitly declare whether (and how) or not a variable's value changes from one snapshot to the next to be helpful in producing a correct model of the desired behaviour. For instance, we found TLA⁺ and its requirement that every variable be constrained between the current and the next snapshot to help us more easily write models that behave as intended. While writing ALLOY models we noticed several times that the formulas we had written left one or more variable under-constrained, allowing them to change arbitrarily, resulting in an incorrect model. We found producing more predictable model behaviour in exchange for more verbosity to be an acceptable trade-off to make, especially in the landing gear case study with its more complex and nested decision branches.

Events: Events are explicitly supported only in DASH. These can be mimicked in other languages using Boolean snapshot elements. However, mimicking DASH's semantics for events in big-steps and small-steps is cumbersome in other languages. For reactive systems with concurrency, events are a common and useful abstraction so support for them in any language that has explicit constructs for modelling concurrency is very valuable.

Constrained relations/functions: TLA⁺, PLUSCAL, EVENT-B, and B provide explicit language syntax for limiting a relation to be a certain kind of function. While nice language features, each of these language constructs usually only abstract away about one formula. ALLOY and DASH go further with multiplicity constraints – these are a concise and general way of describing limitations on relations.

Subtypes: Most of the languages support subtyping and explicit type signatures. TLA⁺ and PLUSCAL have subtyping but not type signatures. Only for our B models did we have to introduce predicates to represent the subtypes.

Fairness: TLA⁺, PLUSCAL, and ASMETAL allow the model's traces to be limited by fairness constraints. Fairness constraints can be mimicked in the other languages through additions to the model and the property to check. For example, to mimic the process fairness constraints of PLUSCAL, we can add an "active" snapshot element for each process and place fairness constraints on this snapshot element in a property.

8 RELATED WORK

We did not include other formal declarative specification languages in our study for various reasons. For instance, we omitted the Z [1] and VDM [40] specification languages, due to the lack of model checking tool support for them, and we did not include formalisms based on process algebras and Petri Nets, as they are at a different (often lower) level of abstraction compared to declarative modelling languages. We also did not include Electrum [41], [42], [43] or DynAlloy [44], which are other extensions of ALLOY to model transition systems, as we decided to include at most one extension for each language, and we found DASH and its features to be a bigger change from ALLOY than Electrum and DynAlloy. Early in 2021, Electrum was incorporated into the Alloy release. Electrum adds the keyword "var" to designate variables as dynamic and primed variable used in a formula mean the value of the variable in the next

step. Electrum uses linear temporal logic (LTL) formulas to describe the traces of the transition system and therefore is quite different from the languages surveyed in this article. DynAlloy is based on program verification concepts of modelling pre and postconditions of actions and combining actions sequentially or in parallel to form a transition system.

Statecharts-based languages and UML state machines [45] provide a graphical manner to describe system behaviour, but do not fall into the category of declarative specification languages: their semantics are often not fully formal and they lack support for declaring datatypes. OCL [46] is a declarative language that can be used in combination with UML to constrain the pre and postconditions of transitions.

The languages of model checkers such as SMV [47] and Spin [30] are lower-level descriptions than what is often convenient for a user. nuXmv [48] is a re-implementation and extension of SMV that adds support for verification of infinite datatypes, such as integers and reals, and incorporates a verification engine with state-of-the-art SAT-based algorithms. However, they all have limited support for user-declared datatypes. Furthermore, in SMV/nuXmv the scope of datatypes and relations must be set at modelling time, which forces users to modify their models every time they want to analyze and check properties in at larger scope.

Recently, Ferrari *et al.* [15] report on an extensive qualitative and quantitative study of formal verification tools applied to railway system designs. Thirteen tools (including non-declarative languages) were compared and a case study completed in seven of those tools. Our overlap with this study is in that they include B and TLA+ in the set of thirteen tools, and evaluate model verification in ProB (one of the seven tools). In terms of the comparison performed, our study focuses only on language features, whereas theirs compares the capabilities, usability, maturity, and flexibility of the supporting tools. Within their language expressiveness category, they consider whether the languages have features such as the ability to express non-determinism, concurrency (asynchronous or synchronous), modularity, and complex data structures (such as sets). However, their observations on these questions are described at a very high level (*i.e.*, language has the feature or does not). Thus, our comparison is very complementary to theirs in studying a different set of languages, providing a detailed comparison of the language constructs, and using a broader set of case studies.

9 CONCLUSIONS

We have presented a qualitative analysis of seven declarative modelling languages (B, EVENT-B, ALLOY, DASH, TLA+, PLUSCAL, and ASMETAL) with respect to the three main categories of data modelling, control modelling, and modularity. Our figures and tables are intended to provide a summary reference for those wishing to compare the languages. Our analysis is based on the completion of six case studies in all languages. We described recommendations for the choice of declarative modelling language for transition systems and possible improvements based on our experience carrying out the case studies.

We believe that an *ideal* declarative behavioural modelling language would include the following features and characteristics:

- implicit representation for the **transition relation** by default for convenience, along with options to supplement or tweak the transition relation by more knowledgeable modellers particularly with respect to stuttering;
- a flexible module system for data decomposition of models across multiple files, with each file hosting one or more **parametrized** modules that support selective **import** and **export** of identifiers;
- a notion of **control state hierarchy** or similar constructs with replicated components to decompose (across multiple files) a transition system from a control modelling point of view into **subtransition systems**, each having their own (uniquely addressable) **namespace** and buffered communication as well as global communication;
- a more declarative rather than imperative **expression** style for transition actions, so as to stay true to the declarative behavioural modelling paradigm;
- dealing with the **frame problem** in a way that is both convenient and avoids the issue of under-specification as seen in ALLOY models, by either adding implicit constraints that keep unconstrained variables in a transition unchanged (as DASH does), or by requiring that every variable either be constrained or marked as unchanged using a special keyword (as TLA+ does);
- a simple but expressive means of **constructing** composite units of data such as **multiplicities**, as well as support for declaring **subtypes**;
- **typechecking** done as a separate pass, with typing constraints restricting the reachable snapshot space for improving performance, as opposed to treating typing constraints as properties checked with other invariants during the main analysis; and
- improved tool support for helping the modeller find the sources of **inconsistency** in their model.

A more in-depth discussion of the concepts in this paper and more exemplars of the differences between the languages can be found in Bandali [49].

ACKNOWLEDGMENTS

We thank Ali Abbassi, Jose Serna, Elias Eid, Tamjid Hosain, Joseph Poremba, and Khadija Tariq for discussions regarding this study. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

- [1] J. M. Spivey, *Z Notation - a reference manual*, 2nd ed. Prentice Hall, 1992.
- [2] D. Jackson, *Software abstractions: logic, language, and analysis*, 2nd ed. Cambridge, Mass: MIT Press, 2012.
- [3] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [4] P. Zave, "Using Lightweight Modeling to Understand Chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.
- [5] C. Newcombe, "Why Amazon chose TLA+," in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, 2014, pp. 25–39.

- [6] N. Huynh, M. Frappier, H. Pooda, A. Mammar, and R. Laleau, "SGAC: A multi-layered access control model with conflict resolution strategy," *Computer Journal*, vol. 62, no. 12, pp. 1707–1733, 2019.
- [7] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.
- [8] —, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
- [9] J. Serna, "Dash: Declarative behavioural modelling in Alloy," MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2019.
- [10] L. Lamport, "The PlusCal algorithm language," in *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, 2009, pp. 36–60.
- [11] A. Gargantini, E. Riccobene, and P. Scandurra, "A metamodel-based language and a simulation engine for abstract state machines," *Journal of Universal Computer Science*, vol. 14, no. 12, pp. 1949–1983, 2008.
- [12] P. Zave, "A practical comparison of Alloy and Spin," *Formal Aspects in Computing*, vol. 27, no. 2, pp. 239–253, 2015.
- [13] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," in *International Conference on Formal Engineering Methods (ICFEM)*, 2010, pp. 581–596.
- [14] M. A. Ardis, J. A. Chaves, L. J. Jagadeesan, P. Mataga, C. Puchol, M. G. Staskauskas, and J. V. Olnhausen, "A framework for evaluating specification methods for reactive systems experience report," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 378–389, 1996.
- [15] A. Ferrari, F. Mazzanti, D. Basile, and M. H. ter Beek, "Systematic evaluation and usability analysis of formal tools for railway system design," *CoRR*, vol. abs/2101.11303, 2021. [Online]. Available: <https://arxiv.org/abs/2101.11303>
- [16] A. Abbassi, A. Bandali, N. A. Day, and J. Serna, "A comparison of the declarative modelling languages B, Dash, and TLA+," in *International Workshop on Model-Driven Requirements Engineering (MoDRE) @ IEEE International Requirements Engineering Conference (RE)*. IEEE Computer Society Press, 2018, pp. 11–20.
- [17] J. S. Ostroff, "Validating software via abstract state specifications," York University, Tech. Rep. EECS-2017-02, 2017.
- [18] B. Fraikin, M. Frappier, and R. Laleau, "State-based versus event-based specifications for information systems: a comparison of B and EB³," *Journal of Software and Systems Modelling*, vol. 4, no. 3, pp. 236–257, 2005.
- [19] F. Mazzanti, A. Ferrari, and G. O. Spagnolo, "Towards formal methods diversity in railways: an experience report with seven frameworks," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 3, pp. 263–288, 2018.
- [20] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [21] N. Nisanke, *Formal Specification: Techniques and Applications*. Springer, 1999.
- [22] S. Farheen, "Improvements to transitive-closure-based model checking in Alloy," MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2018.
- [23] F. Boniol and V. Wiels, "The landing gear system case study," in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, 2014, pp. 1–18.
- [24] D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel, "Validation of the ABZ landing gear system using ProB," in *ABZ (Case Study)*, ser. Communications in Computer and Information Science, vol. 433. Springer, 2014, pp. 66–79.
- [25] P. Arcaini, A. Gargantini, and E. Riccobene, "Modeling and analyzing using ASMs: The landing gear system case study," in *ABZ (Case Study)*, ser. Communications in Computer and Information Science, vol. 433. Springer, 2014, pp. 36–51.
- [26] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA⁺ specifications," in *Correct Hardware Design and Verification Methods (CHARME)*, Mar. 1999, pp. 54–66.
- [27] E. Sherratt, "Relativity and abstract state machines," in *International Workshop on System Analysis and Modeling: Theory and Practice (SAM) Revised Selected Papers*, 2012, pp. 105–120.
- [28] J. McCarthy and P. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, no. 4, pp. 463–502, 1969.
- [29] S. Farheen, N. A. Day, A. Vakili, and A. Abbassi, "Transitive-closure-based model checking in Alloy," *Journal of Software and Systems Modelling*, vol. 19, pp. 721–740, 2020.
- [30] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
- [31] L. Lamport, "Real-time model checking is really simple," in *Correct Hardware Design and Verification Methods (CHARME)*, 2005, pp. 162–175.
- [32] H. Zhang, M. Gu, and X. Song, "Specifying time-sensitive systems with TLA+," in *IEEE International Computer Software and Applications Conference (COMPSAC)*, 2010, pp. 425–430.
- [33] Leslie Lamport, "Safety, Liveness, and Fairness," May 2019, <https://lamport.azurewebsites.net/tla/safety-liveness.pdf>.
- [34] K. R. Apt and E. Olderog, "Proof rules and transformations dealing with fairness," *Science of Computer Programming*, vol. 3, no. 1, pp. 65–100, 1983.
- [35] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, 1994.
- [36] ClearSy, *B Language Reference Manual, version 1.8.6*, 2007. [Online]. Available: <https://www.atelierb.eu/wp-content/uploads/sites/3/ressources/manref1.8.6.fr.zip>
- [37] M. Jastram and P. M. Butler, *Rodin User's Handbook: Covers Rodin v2.8*. USA: CreateSpace Independent Publishing Platform, 2014. [Online]. Available: <https://www3.hhu.de/stups/handbook/rodin/>
- [38] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Transactions on Software Engineering*, vol. SE-6, pp. 2–23, Jan. 1980.
- [39] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, Sep. 1994.
- [40] C. B. Jones, *Systematic software development using VDM*, 2nd ed. Prentice Hall, 1991.
- [41] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Foundations of Software Engineering (FSE)*, 2016, pp. 373–383.
- [42] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The Electrum analyzer: Model checking relational first-order temporal specifications," in *Automated Software Engineering*, 2018, pp. 884–887.
- [43] A. Cunha and N. Macedo, "Validating the hybrid ERTMS/ETCS level 3 concept with Electrum," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 3, pp. 281–296, 2020.
- [44] G. Regis, C. Cornejo, S. Gutiérrez Brida, M. Politano, F. Raverta, P. Ponzio, N. Aguirre, J. P. Galeotti, and M. Frias, "DynAlloy analyzer: A tool for the specification and analysis of Alloy models with dynamic behaviour," in *Foundations of Software Engineering (FSE)*. ACM, 2017, pp. 969–973.
- [45] "OMG unified modeling language," <http://www.omg.org/spec/UML/2.5/PDF/>, 2015, [Online; accessed Jul 4, 2021].
- [46] "OMG object constraint specification (OCL) specification," <http://www.omg.org/spec/OCL/2.4/PDF>, 2014, [Online; accessed Jul 4, 2021].
- [47] K. L. McMillan, *Symbolic Model Checking*. Springer, 1993.
- [48] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer-Aided Verification*, 2014, pp. 334–342.
- [49] Amin Bandali, "A comprehensive study of declarative modelling languages," MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2020.

PLACE
PHOTO
HERE

Amin Bandali Biography text here.

PLACE
PHOTO
HERE

Nancy A. Day Biography text here.