A Comparison of the Declarative Modelling Languages B, Dash, and TLA$^+$

Ali Abbassi, Amin Bandali, Nancy A. Day, and Jose Serna
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
Email: {aabbassi, abandali, nday, jserna}@uwaterloo.ca

***Abstract*—Declarative behavioural modelling is a powerful modelling paradigm that enables users to model system functionality abstractly and concisely. We compare two well-used formal declarative modelling languages, B and TLA$^+$, with a new modelling language called Dash. Dash is an extension of Alloy with explicit syntactic constructs for modelling transition systems, and it includes control state hierarchy and events. Particular topics that we cover in our comparison are: differences in the datatypes and type systems; how the transitions/operations can be described; how the transition relation is a combination of the transitions; and the default choice each language makes regarding permitted variable changes in a transition. Our goal is to discuss the interesting differentiating characteristics of each language to aid users in determining which language is the most suitable for their system.**

I. Introduction

Behavioural modelling languages that are declarative enable users to model system functionality abstractly and concisely, providing early feedback through analysis of a model. A behavioural model is a model of a transition relation, *i.e.,* pairs of variable-value mappings (which we call snapshots) that represent the possible steps of the system.

The declarative behavioural modelling paradigm, as represented by languages such as Z [1], VDM [2], B [3], Alloy [4], and TLA$^+$ [5], generally has the following characteristics:

1) Describes transitions through constraints (rather than calculations).
2) Supports user-introduced datatypes and rich datatypes, such as lists, trees, and rings, with an axiomatization of their behaviour.
3) Has a formal semantics, usually rooted in first-order logic (FOL) and/or set theory.
4) Is one integrated model.

Because of item (1) above, declarative models are not necessarily executable. Models can be used for any level of system description, however, declarative models are valued for their abstractness and conciseness, making them useful very early in the system development process during requirements engineering. Because of their formal roots in FOL, automated analysis of the behaviour via model checking has previously been limited. More recently, as analysis methods such as bounded model checking (BMC [6]) and finite model finding have been developed, model checking of declarative models has become a tractable problem, increasing the value of declarative modelling. In bounded model checking, there is a limit placed on the number of steps of a transition relation that are verified. In finite model finding, a finite limit is placed on the size of all sets making the problem decidable.

We compare two well-used declarative modelling languages, each with model checking support, B [3] and TLA$^+$ (Temporal Logic of Actions) [5], with a new modelling language we are developing called Dash [7], [8]. Dash is an extension of Alloy with explicit syntactic constructs for modelling transition systems, and it includes labelled control state hierarchy and events inspired by Statecharts [9]. Model checking support for Dash is accomplished via a translation to Alloy [8].

A general categorization of different kinds of systems can be made with respect to whether the system's behaviour is more control-oriented or more data-oriented. Control-oriented systems generally have simple datatypes and simple operations on these datatypes, but complex behaviour with respect to when a transition is taken. The Statecharts [9] family of languages with hierarchical and concurrent labelled control states were developed to model this control complexity. Data-oriented systems have rich datatypes (such as lists and structures), but limited complexity with respect to when transitions can be taken; typically all transitions can be taken at any moment (subject to the preconditions of a transition).

The method we followed for our comparison is to model a small number of reactive systems across the spectrum of data-oriented and control-oriented systems. From these efforts, we devised aspects to describe differences and similarities among the languages. Our contribution is to elucidate the differences in the syntactic modelling constructs provided by each language for describing a transition system and the semantic meaning of these top-level constructs. Because the semantics of a language are implicit, it is useful to have a clear statement of their subtle differences, such as how transitions are combined to form a transition relation, to ensure that a user's model is an accurate representation of the system. We leave for future work comparisons regarding robustness and performance of tool support.

II. Background

The purpose of all three languages we compare is to represent a unique transition system, *i.e.,* a description of dynamic system behaviour specifying how the model moves from one variable to value mapping, called a snapshot, to another. Each of these moves is one transition. All the languages we compared are intended to describe abstract behaviour. Thus, they are all built on first-order logic and set theory to describe rich datatypes (*e.g.,* uninterpreted types, relations/tables, functions). The behaviour of the functions and relations can be uninterpreted (*i.e.,* unconstrained), semi-interpreted (*i.e.,* partly axiomatized) or completely interpreted (*i.e.,* defined). All of the languages allow models to be written without describing the size of sets (the scopes); the scopes may need to be chosen for analysis. In each language, there is a method for describing snapshots and transitions. In all languages the package of behaviour in a transition is atomic (*i.e.,* transitions cannot interfere with each other). Next, we provide a brief overview of each of the languages. We show examples of the languages in Section IV.

A. B

The B language describes software systems formally as interacting abstract machines. It was first developed by Abrial, and has continued to be developed for commercial usage. The backbone of B is first-order logic and set theory. Sets can be specified using set comprehension, or set operations such as Cartesian product, intersection, power set, *etc.*, and can be used for updating a variable's value, and creating constraints. Predicates can be axiomatized or defined using propositional logic operations and set predicates such as membership, subset, *etc.* Relations over sets are elements of the Cartesian product of sets. Functions can be declared explicitly by using function-specific type syntax (for partial, total, surjective functions *etc.*), or implicitly by restricting relations.

Each abstract machine is comprised of multiple sections that together describe a model. Each abstract machine starts with the keyword **`MACHINE`**, which is followed by the name of the machine and ends with the **`END`** keyword. The keyword **`SETS`** begins a section that declares the sets used in the model. Sets can be used as types in B with the '`:`' syntax in the **`INVARIANT`** section, which is called a membership predicate. Subtyping is not supported explicitly but can be achieved using predicates. If the type of the sets in an operation does not match, B's typechecker will raise an error. The elements of a set can be enumerated where it is introduced, or later can be constrained in the operations or invariants.

The section on **`CONSTANTS`** declares the constant (unchanging) elements of a model. The **`PROPERTIES`** section contains constraints on constants. Snapshot elements are declared under the keyword **`VARIABLES`**. The **`INVARIANT`** section contains constraints that must be proven to hold in every snapshot of a model. The type constraints for variables must be stated in this section as membership predicates. The **`ASSERTIONS`** section contains the properties to be checked. The initial snapshot of a model is described under the **`INITIALISATION`** keyword using assignments. And finally, transitions are specified using operations under the **`OPERATIONS`** keyword. The postconditions of an operation are *generalized substitutions* of new values of snapshot elements. Various statements are allowed in these substitutions, such as skip, assignment, non-deterministic assignment from sets, and precondition followed by postcondition. Any of these can be combined in parallel or sequential order.

B provides users with the ability to further refine their abstract machines, as they progress in a project, but we did not use this facility in our study.

We used the ProB [10] tool for model checking our B models. Another tool for B is AtelierB [11].

B. Dash

Dash is a new language that we are developing for writing declarative behavioural models; it combines the abstractions of first-order logic and set theory with common control-oriented modelling constructs such as labelled control state hierarchy and named events. Dash is an extension to the Alloy language for describing behavioural models.

In Dash, a transition has a labelled control state as its source and destination. The control state hierarchy is described in a nested manner where **`state`** means a basic state or OR-state and **`conc state`** means an AND-state. Some parts of Dash can be described graphically using standard Statecharts notation (as in 2), however, for generality, Dash is a textual language. In Dash, a transition has the following structure:

```
trans <name> {
  from <src_state>
  on <trigger_event>
  when <guard_condition>
  goto <dest_state>
  do <action>
  send <generated_event>
}
```

Every part of a transition is optional, and if not present, the meaning is given by the context. The **`from`** part indicates the source state of a transition while the **`goto`** part describes the destination state; if either of these parts is omitted, its value is the most immediate state that contains the transition definition. The **`on`** part describes the event that triggers a transition and the **`when`** part specifies the guard condition. The action of a transition is specified using the **`do`** keyword and any internal events generated are stated using **`send`**. Guard conditions and actions are described using Alloy syntax where unprimed variables names refer to the values of snapshot elements in the source snapshot and primed variable names are the values in the destination snapshot. Alloy formulas include set operators and make extensive use of the join (`.`) operator since everything in Alloy is a set (scalars are represented as singleton sets).

Inside a state, Dash supports the declaration of the snapshot elements using Alloy's types. In a declaration, the keyword **`env`** designates a snapshot element as a monitored variable. The value of a controlled variable can be modified by actions of transitions, whereas the value of monitored variables changes non-deterministically. The keyword **event** introduces named events, such as "button pushed", which are names for occurrences at a moment in time that can trigger a transition. In Dash, labelled control states are used as namespaces so different snapshot elements can have the same name as long as they are not declared in the same control state. Constants are declared outside of states using Alloy syntax.

Dash follows the usual semantics of Statecharts: transitions from states higher in the hierarchy have priority over those lower in the hierarchy, and concurrent states can each take one transition in response to an environmental input forming big steps (consisting of multiple transitions).

We used tools that we developed for Dash to translate Dash to Alloy and used the Alloy Analyzer for model checking [8]. Our tools are built using Xtext [12][1].

C. TLA$^+$

TLA$^+$ is a formal specification language developed by Leslie Lamport, based on the idea that using simple mathematics is the best way to write formal descriptions; and that a specification language should provide the bare minimum required for writing simple mathematics to describe systems precisely. TLA$^+$ has first-order logic with an untyped classical set theory as its modelling language.

TLA$^+$ specifications are organized into modules, each of which can *extend* (import) other modules. The most common sections in a TLA$^+$ module are **CONSTANTS**, **VARIABLES**, and **ASSUME** blocks, and two-snapshot predicates (operators) called actions, which are transitions. Users must create a top-level transition relation combining two-snapshot predicates. In addition to propositional and predicate logical connectives, TLA$^+$ has built-in operators for working with various datatypes: typical set operations from set theory (*e.g.,* $\in$, $\cup$, and $\subseteq$), functions, as well as records and tuples, both of which are syntactic sugar on top of functions.

We used the tools in the TLA Toolbox, which includes the SANY parser and semantic analyzer for TLA$^+$ [13], the TLC model checker [14], [13], the PlusCal algorithm language [15], the TLATEX pretty printer [13], and the TLAPS (TLA$^+$ Proof System) [16]. TLA$^+$ supports model refinement. Note that even though the TLATEX pretty printer is usually used for typesetting TLA$^+$ specifications, for the purposes of consistency with the other two languages discussed in this paper, TLA$^+$ excerpts are shown in ASCII notation, as input by a modeller.

III. Methodology

To compare the three languages, we chose a variety of small examples across the range of data-oriented to control-oriented systems. All the examples correspond to reactive systems in that they interact with their environment and either do not have a final snapshot or have a final snapshot with a loop. Strongly data-oriented models have rich datatypes (such as tables/relations that change over time), but limited need for specifying when a transition is relevant. Strongly control-oriented models have limited datatypes (*e.g.,* numbers, Booleans, arrays of fixed size), but are rich in their need to describe when a transition is relevant, typically dependent on modes and priorities. Some languages implicitly, through their semantics, control whether a transition is examined to see if it can be taken in a step (*e.g.,* the semantics of hierarchical control states of Statecharts determine whether a transition is relevant or not); other languages require users to explicitly encode any such relevancy requirements.

[1]Our tool is available online at http://129.97.7.33:8080/dash/

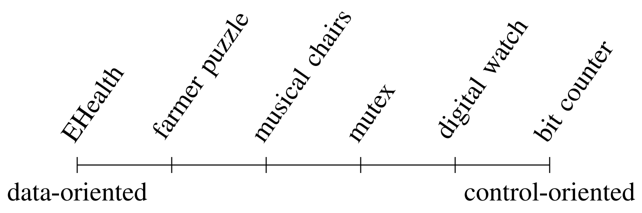

Fig. 1. Range of Models

Figure 1 shows our evaluation of where our six examples fit on the spectrum of data-oriented to control-oriented. For each of these examples, we created a model in each of the three languages. It is difficult to judge them for equivalence because they do not have exactly the same snapshot space, depending on what variables are chosen for the snapshot (*e.g.,* control state hierarchy) and how the behaviour was allocated to transitions. It was important to allow each modeller to describe the behaviour in the way that was the most natural in the language. We used two methods to check their equivalence: 1) inspection and discussion; and 2) verification of approximately the same set of temporal properties in each model, some of which were domain-independent properties. Since the tools varied in whether they supported LTL (Linear Temporal Logic) or CTL (Computation Tree Logic), our properties are not necessarily exactly the same in the different languages.

The EHealth example (originally in [17]) resides at the far left end of the characterization spectrum, as it uses rich datatypes like sets and tuples and has no need for controlling when a transition is relevant beyond preconditions. EHealth is an electronic health system that ensures that medications prescribed to patients are safe, by keeping track of the known dangerous interactions between the medications recorded in the system. The system allows a transition adding a medication to a patient's prescription if and only if it does not interact dangerously with any of the other medication previously prescribed to that patient. We formulated and checked that this safety property is an invariant of the system.

The farmer puzzle example is a famous puzzle in which a farmer wants to move his animals and a bag of seeds without any of them being eaten by any other. The transitions are the possible moves of the farmer. The puzzle example fits just after EHealth in our spectrum. This example does not have any events or concurrency. The only control required on when a transition is relevant is the location of the farmer.

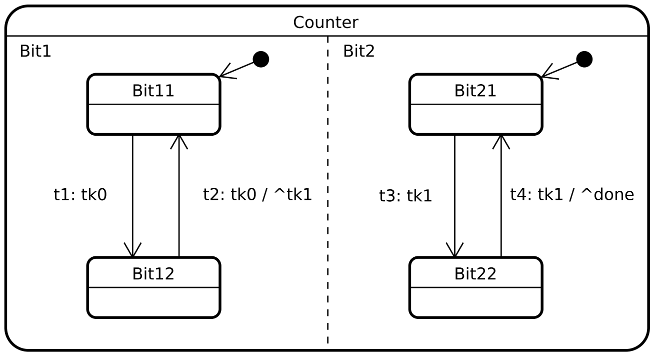

Fig. 2. Model of a two-bit counter

The musical chairs example (originally in [18] and modelled in Alloy in [19]) resides somewhere in the middle of the characterization spectrum. The modes of the game (players are sitting, walking, *etc.*) are represented using control states, and it has interesting datatypes to represent the mapping between the players of the game and the chairs they sit on. We checked some safety and liveness properties.

The mutex example is a well-used example for preventing two concurrent processes from entering their shared section at the same time. On our spectrum, the mutex example is located after the musical chairs example, because there is more complexity in when a transition is relevant than in musical chairs and its datatypes are fairly simple.

The digital watch example is an adaptation of Harel's model [9]. The model hierarchically decomposes a watch's behaviour, and makes extensive use of environmental events for the buttons of the watch to enable transitions. We verified some reachability properties.

The bit counter example (adopted from [20]) models a two bit counter with concurrent states for each bit. The control state hierarchy of the model is depicted in Figure 2 using standard Statecharts notation. The bit counter example is a good representative of a control-oriented model because it naturally utilizes state hierarchy, concurrency, and cascading effects between concurrent regions from an environmental input. We checked that the model always reacts to environmental events, and that operations are successfully completed.

Prior to our work, the musical chairs, digital watch and bit counter examples had previously been done in Dash by us; the mutex example existed in B and TLA$^+$ by other authors; and the EHealth example had previously been done by one of our authors in TLA$^+$. All the other models were created for this paper. Our models are available at: https://cs.uwaterloo.ca/~nday/models/2018-modre .

IV. Comparison

In the next sections, we describe the differences and similarities we observed as we modelled our examples in the various languages.

A. Datatypes, Typechecking, Snapshot Declaration

Each language supports datatypes that describe the possible values for either snapshot variables or constants. All the languages support the declaration of uninterpreted sets/types, which are critically important in the formal description of requirements at an abstract level. To compare the languages, Figure 3 shows an equivalent snapshot declaration in each language.

In B, sets are declared in the **SETS** section and the names of the snapshot elements are listed in the **VARIABLES** section. The types of variables are included under **INVARIANT** (which must be proven). Scalars are permitted as on line 17 as well as set types (using the power set as a type). The types of snapshot variables (the expressions following the '`:`') are called membership predicates and can be described using a rich syntax. There is no explicit means of defining multiplicity of relations; these can be added as constraints (lines 28–31).

Built-in types in B consist of numbers, sequences, records, strings, and trees. Any of these datatypes have their own pre-defined operations available to users. For example, type `tree(S)` is a set of trees over the domain of `S`. A typechecking pass is able to determine if the value of an argument is of the correct type before more expensive analysis. B does not support subtyping, but it can be accomplished using predicates. For example, a parent type is defined in the **SETS** section (line 3), and a constant predicate is defined, as in lines 11 and 34, to model the subtyping, and must be used as a precondition for any use of the `R1` relation.

Dash uses Alloy signatures for set declarations on line 4. A subtype is created on line 7 using the **`extends`** keyword. Alloy does not support multiple inheritance. Based on Alloy, in Dash every variable is a set or a relation and any set can be used as a type. There are no scalars, thus line 11 of the Dash model creates a set of A's and line 13 shows how to create a singleton set equivalent to a scalar. The type declaration of **`one Int`** is implicitly an invariant stating that set `v2` must always be a singleton set in all snapshots. These typing invariants limit the reachable snapshot space (as opposed to being properties to prove about the model). It is possible to create an inconsistent description if there is a transition that adds two elements to this set.

Functions can be created in Alloy using a small range of multiplicity keywords such as **`lone`** (partial function) or **`one`** (total function). Additionally, constraints can be added to require a relation to have a certain multiplicity, as in the two facts on lines 21–24 and lines 27–30, which require `f2` to be surjective and `R1` to have multiplicity 1:2. These are invariants of the model and constrain the reachable snapshot space. Also note that `C` is a subtype of `A` and on line 19 the domain of `R1` must be drawn from `C` only.

Somewhat hidden to users of Alloy is that there is no structure/packaging mechanism for data such as a record. The **`state`** syntax in Dash is translated to a snapshot signature in Alloy where each of the variables of the snapshot are actually relations between the snapshot and the variable value. It can

```
 1  // B
 2  SETS
 3    A; B;
 4    Event = {ev1, ev2}
 5
 6  VARIABLES
 7    v1, v2, f1, f2, R1,
 8    event
 9
10  CONSTANTS
11    isC
12
13  INVARIANT
14    // a set of A
15    v1: POW(A) &
16    // a singletone integer
17    v2: INTEGER &
18    // a total function
19    f1: A --> B &
20    // a surjective function
21    f2: A -->> B &
22    // binary relations
23    R1: A * B &
24    // Multiple events can
        happen simultaneously
25    event: POW(Event) &
26    // multiplicity constraint
27    // isC is predicate for
        subtype
28    !(var).(var: A & isC(var)
        <=>

29      card(R1(var)) = 2 &
30      not (isC(var)) =>
31      card(R1(var)) = 0)
32
33  PROPERTIES
34    isC: A --> BOOL
```

```
 1  // Dash
 2  open util/integer
 3
 4  sig A,B {}
 5
 6  // C is a subtype of A
 7  sig C extends A {}
 8
 9  state Example {
10    // a set of A's
11    v1: set A,
12    // a singleton set of Ints
13    v2: one Int,
14    // total function
15    f1: A one -> one B,
16    // function
17    env f2: A -> one B,
18    // This means C x B
19    R1: C -> B,
20
21    fact f2_is_Surjective {
22      all b:B |
23        some a:A | a->b in f2
24    }
25
26    // R1 multiplicity is 1:2
27    fact R1_multiplicity {
28      all a:A |
29        #a.R1 = 2
30    }
31
32    event ev1 {}
33
34  }
```

```
 1  \* TLA+
 2  EXTENDS Integers, FiniteSets
 3
 4  CONSTANTS
 5      A, B, C
 6
 7  VARIABLES
 8      v1, v2, f1, f2, R1, ev1
 9
10  \* helper operators
11  Range(f) ==
12      {f[x] : x \in DOMAIN f}
13  SurFun(a, b) ==
14      {f \in [a -> b] :
15       b = Range(f)}
16  Singleton(S) ==
17      {{i} : i \in S}
18  RelRange(R) ==
19      {x[2] : x \in R}
20  RelDomRes(S, R) ==
21      {x \in R : x[1] \in S}
22  OneToN(R, n) ==
23      \A x \in RelDom(R) :
24      Cardinality(RelRan(
        RelDomRes(R, {x}))) = n
25
26  TypeOK ==
27      /\ v1 \in A
28      /\ v2 \in Singleton(Int)
29      /\ f1 \in [A -> B]
30      /\ f2 \in SurFun(A, B)
31      \* \X means cross product
32      /\ R1 \in C \X B
33      /\ OneToN(R1, 2)
34      /\ ev1 \in BOOLEAN
```

Fig. 3. Snapshot Declarations in B, Dash, and TLA$^+$

be confusing to users to understand the output of the analysis (*e.g.,* counterexamples for model checking) when a variable name is actually a relation.

Alloy has no built-in datatypes, but it has library modules for the types Booleans, integers, lists, and graphs. The lack of built-in types in Alloy, in particular the integer type, comes from the finite model approach to analysis; since Alloy does not support the full behaviour of integers. Integers are a finite set of abstract objects just like any other set in Alloy. Typechecking is limited to checking if a set must be empty under the constraints of a specification [21].

In TLA$^+$, sets are often used as building blocks for more complex datatypes. For instance, the *Peano* module includes Peano's axioms used for the set Nat of natural numbers, with its zero element and the successor function [13]. In recent TLA$^+$ versions, naturals along with other standard modules such as bags and sequences are built into TLA$^+$ and are implemented in the host language (Java) to improve efficiency.

In TLA$^+$, a snapshot declaration is created by declaring the variables in the `VARIABLES` section. Because TLA$^+$ does not have a type system, typing constraints are explicitly stated as regular constraints. An example of this is demonstrated in the right-most column of Figure 3, where `TypeOK` is a typing invariant. These invariants must be proven. In our experience, having to manually add typechecking invariants in the absence of a typechecker can be error-prone, and can hinder the debugging experience: errors arising from violating typing constraints can result in cryptic error messages.

Lastly, TLA$^+$ distinguishes between relations and functions: a function is an object mapping a domain to a codomain, described using the '==' syntax, but a relation is a set of tuples. Functions are used behind the scenes for defining many of the built-in datatypes like sequences (tuples) and records.

Dash provides explicit syntax for stating that a snapshot variable is part of the environment (using the keyword **`env`**), meaning that it is monitored and not controlled. This distinction affects the semantics of transitions in that an environmental variable is allowed to change non-deterministically whereas a non-environmental variable is either constrained in the transition or keeps its previous value. B and TLA$^+$ do not provide this distinction.

Dash has explicit syntax for creating events (illustrated on line 32), which are occurrences at a moment in time (*e.g.,* "button pressed", "card swiped"). Events are often helpful in describing the abstract behaviour of reactive systems. The semantics of events is that they persist as long as transitions are enabled allowing multiple transitions to be taken in response to an environmental input. Neither B nor TLA$^+$ support named events so some representation, such as making them Booleans, must be chosen by a modeller.

None of the languages explicitly support operators for dynamic allocation of parts of the state (*i.e.,* "new" as in Spin [22]).

B. Expressions

In this section, we discuss what operators are available to write expressions in each language.

Since all the languages are based on first-order logic and set theory, their expressions are reasonably similar in succinctness. They all provide operations to modify relations and functions. Alloy/Dash makes extensive use of the join (`.`) operator; in particular variables that are part of a snapshot, `s`, can be accessed using `s.v`, which actually means the join operation of the snapshot set with the relation, but looks very much like a record access.

B has the most imperative style in that changes to variables in a postcondition of an operation and initial values of variables are both described using the assignment syntax. Assignment can be to a non-deterministically chosen value. These assignments can be combined using operators such as '`||`' and '`;`' for parallel and sequential assignments, respectively.

In addition to the common operators, TLA$^+$ has the following constructs: `CHOOSE` operator, `IF..THEN..ELSE` expressions, `CASE` expressions, and `LET..IN` expressions.

Each language has packaging mechanisms for expressions to support modularity in specifications. In Dash, the **`pred`** construct of Alloy is a named, parameterized constraint that can be used in other constraints. Similarly, the **`fun`** construct of Alloy is a named, parameterized function that returns a value and can be used in other constraints. Alloy functions and predicates are unfolded prior to analysis. In B, functions and relations must be declared in the **`CONSTANTS`** section and axiomatized in the **`PROPERTIES`** section. There is no mechanism to define blocks with return values that do not form part of the model space, such as Dash/Alloy predicates and functions. In TLA$^+$, one can define new operators using '`==`'. For instance, on lines 11-12 of the right-most column in Figure 3 we define the `Range` operator, which is absent in TLA$^+$. TLA$^+$ also supports recursive and higher-order operators, which take other operators as arguments.

C. Initialization, Transitions, and Transition Relation

In this section, we discuss how a transition system is created in each language. Each transition is an atomic set of constraints that relate a pair of snapshot elements. To illustrate this discussion, Figures 4 and 5 show equivalent parts of the musical chairs example in the three languages. Type constraints in B and TLA$^+$ have been omitted for brevity. Table I summarizes some of the terminology used in the three languages.

Initialization: B and Dash have explicit sections devoted to expressing the initial constraints, whereas in TLA$^+$ any definition name can be used; by convention, it is `Init` as on line 12 and later it is referred to directly in the statement of the transition system on line 51. In Dash and TLA$^+$, any constraints can be used to limit the initial variable values so their initial constraints are similar (subject to differences in the syntax). In TLA$^+$, every variable must be initialized, which means that it is mentioned in an initialization constraint. For initialization in B, assignments ('`:=`') and non-deterministic choice from a set ('`::`') must be used for all variables, and combined by parallel composition ('`||`'). Because of this limitation, we were unable to specify that initially the number of players is one greater than the number of chairs. We could have made this a snapshot constraint (*i.e.,* always true) but that would be stronger than an initial constraint and we wanted to make it a property that is proven to be invariant about the model, so we chose to leave it out and make it true when we chose the scopes for analysis. Our B model of musical chairs requires the extra variables `PickedPlayer` and `PickedChair` (for reasons that we will explain later) so these must also be initialized.

Transitions: Figures 4 and 5 include one musical chairs transition in each language – the one for when the music stops and the players find seats, populating the occupied function. In B, transitions are called **`OPERATIONS`**; in Dash, the keyword **`trans`** is used; in TLA$^+$ no keyword is necessary because users must model explicitly how the transitions will be combined. Because Dash has labelled control states, it can make the modes of musical chairs (standing, sitting, *etc.*) the source state of transitions, whereas in B and TLA$^+$, these modes are variables and referred to in the preconditions of the transitions. It might have been possible to eliminate the state variable completely based on the status of the occupied variable, but this simplification would not be possible in all models. The textual description of the state hierarchy in Dash allows a transition to recognize its context and implicitly determine its source state, which makes the description more concise.

Dash has explicit events (such as `MusicStops`) to describe the occurrence of the music ending. In B, we encoded an event set (called `event`). In TLA$^+$, we used a Boolean variable to represent when the music is on or off. There are multiple ways to encode events, but depending on a user's understanding of the problem, explicit events may feel more natural to use.

In TLA$^+$ any expression relating snapshots can describe a transition. Furthermore, in TLA$^+$ there is no syntactic distinction between preconditions and postconditions. In B, the precondition and postcondition are explicitly labelled (keywords **`PRE`** and **`THEN`**), as in Dash (keywords **`when`** and **`do`**). In Dash and TLA$^+$, unprimed variable values are the values of variables before a transition and primed versions of the variables refer to their value after the transition has been taken. In Dash any expression in the language can go in the precondition and postcondition of a transition. For example, the constraint that `occupied` is a total function is specified on lines 36–37. The equivalent constraint in TLA$^+$ is on lines 30–31, which uses a type-like syntax. In B, postconditions are more structured and must be stated as assignments. To mimic the

```
// extracts of B model of musical chairs
SETS
  State = {standing, sitting, walking, won};
  Event = {MusicStarts, MusicStops};
  Chairs;
  Players

VARIABLES
  activePlayers, state, PickedPlayer, PickedChair
    , occupied, activeChairs, event

INVARIANT
    // type constraints
    activePlayers : POW1(Players)
  & state : State
  & activeChairs : POW(Chairs)
  & PickedPlayer : Players
  & PickedChair : Chairs
  & occupied : Chairs --> Players & event : Event

INITIALISATION
  activePlayers, state, activeChairs, occupied,
    event :=
    Players, standing, Chairs, {}, MusicStops ||
  PickedPlayer :: Players ||
  PickedChair :: Chairs

OPERATIONS
  ...

  MusicStops =
  PRE
    event = MusicStops &
    state = walking
  THEN
    state := sitting
  END;

  Assignment =
  PRE
      (not (dom(occupied) = activeChairs))
    & state = sitting
  THEN
    PickedChair::(activeChairs - dom(occupied));
    PickedPlayer::(activePlayers-ran(occupied));
    occupied := (occupied \/
        {PickedChair |-> PickedPlayer})
  END;

  ...
END
```

```
// extracts of Dash model of musical chairs
open util/integer

sig Chairs, Players {}

conc state Game {
  // Game variables
  activePlayers: set Players
  activeChairs: set Chairs
  occupied: Chairs set -> set Players

  event MusicStarts {}
  event MusicStops {}

  init {
      #activePlayers > 1
      #activePlayers = (#activeChairs ).plus[1]
      // all Chairs and Players included
      activePlayers = Players
      activeChairs = Chairs
      occupied = none -> none
  }

  default state Start { ... }

  state Walking {
    trans Sit {
      on MusicStops
      goto Sitting
      do  {
        occupied' in
           activeChairs -> activePlayers
        activeChairs' = activeChairs
        activePlayers' = activePlayers
        // occupied' must be total function
        all c : activeChairs' |
           one c .(occupied')
        // each occupying player is
        // sitting on one chair
        all p : Chairs.(occupied') |
           one occupied'.p
      }
    }
  }

  state Sitting { ... }
  state End { ... }
}
```

Fig. 4. Musical Chairs in B and Dash

constraints on occupied in the postcondition of the transition, in B we had to decompose it into two operations (`MusicStops` and `Assignment`) where Assignment incrementally matches a player with a chair. An operation will be completed only if the precondition is satisfied. In any of these operations, the generalized substitutions or assignments can be taken in parallel or sequentially, by using '`||`' for parallel and '`;`' for sequential.

Dash includes some syntactic sugar for writing transitions concisely, which we have not illustrated. Transition comprehension allows a set of similar transitions (*e.g.,* every state has a transition to an error state on an error event) to be expressed using a pattern. Add-ons allow an action (*e.g.,* increase a counter) to be added to multiple transitions. And transition templates provide a parameterized way to create transitions.

Transition Relation: In TLA$^+$, the transition relation must be stated explicitly (line 51), thus modellers have complete flexibility in how the transitions are combined. Contradictions within a postcondition can cause deadlock as no transition can be taken.

In B, the transition relation is implicitly $(pre_1 \wedge post_1) \vee (pre_2 \wedge post_2) \vee \ldots$; which means if more than one precondition is satisfied, only one of the postconditions needs to be satisfied.

In Dash, the transition relation is formed implicitly following the semantics of concurrent, hierarchical state machines.

TABLE I
COMPARING TERMINOLOGY OF B, DASH, AND TLA$^+$

Language	Initial State (s)	Transitions (s,s')	Snapshot Constraint (s)	What is Verified
B	**INITIALISATION**	**OPERATIONS**	**PROPERTIES**	**INVARIANT** (may include type constraints on snapshot elements), assertions, LTL
Dash	**init**	**trans**	**invariant**, type constraints	CTL with fairness constraints
TLA$^+$	Init	implicit	**ASSUME**, state and action constraints in model settings	invariants (may include type constraints), extended LTL

```
\* extracts of TLA+ model of musical chairs
EXTENDS Integers, FiniteSets

CONSTANTS CHAIRS, PLAYERS

VARIABLES
 activePlayers, activeChairs,
 occupied, music_playing, state

STATE_ == {"Start","Walking","Sitting","Won"}

Init ==
 /\ Cardinality(activePlayers) > 1
 /\ Cardinality(activePlayers)
     = Cardinality(activeChairs) + 1
 \* all Chairs and Players included
 /\ activePlayers = PLAYERS
 /\ activeChairs = CHAIRS
 \* initially the empty function
 /\ occupied = <<>>
 /\ music_playing \in BOOLEAN
 /\ state = "Start"

...

Sit ==
 /\ state = "Walking"
 /\ ~music_playing
  \* force occupied to be total
 /\ occupied' \in
     [activeChairs -> activePlayers]
  \* each chair maps to only one player
  \* \A is forall
 /\ \A c \in activeChairs,
      p1, p2 \in activePlayers:
      occupied'[c] = p1 /\ occupied'[c] = p2
      => p1 = p2
  \* each occupying player sits on one chair
 /\ \A p \in Range(occupied'),
     c1, c2 \in DOMAIN occupied':
     occupied'[c1] = p /\ occupied'[c2] = p
     => c1 = c2
 /\ state' = "Sitting"
 /\ UNCHANGED <<activeChairs, activePlayers,
    music_playing>>

...

Next == ... \/ Walk \/ Sit \/ ...

vars == << activePlayers, activeChairs,
    occupied, music_playing, state >>
Spec == Init /\ [][Next]_vars
```

Fig. 5. Musical Chairs in TLA$^+$

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. Particularly distinct from the other languages is the concurrent states found in a Dash model. Because of this concurrency, Dash makes the distinction between big steps and small steps in the transition relation. 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. Environmental events can change only at big step boundaries (called a stable snapshot) so the occurrence of an environmental event can trigger multiple transitions as long as the transitions are in different concurrent regions. Events generated by one transition can trigger other transitions (in different concurrent regions) within the same big step as happens in the bit counter example.

The difference between these methods of creating a transition relation from the transitions was most noticeable in the bit counter example, which has concurrency. In the TLA$^+$ model, since the sequence of transitions to be taken following a tick was clear from the model, the transitions of state Bit1 and state Bit2 (see Figure 2) were collapsed into one transition where all state changes happened in the same step. In another concurrent model, it may not be so obvious how to create a transition relation combining transitions in concurrent regions.

Frame Problem: All languages face the issue of how snapshot variables are allowed to change in a step. In TLA$^+$, every variable must be constrained by the transition, either by being declared **UNCHANGED** or by a predicate describing its value in the next state. In B, any variable not mentioned in the assignments retains its value from the previous step. This choice of semantics follows from B's imperative nature. In Dash, with its syntactic designation of environmental variables, if the primed version of a non-environmental variable is not mentioned in the postcondition, it keeps its value from the previous snapshot; environmental variables are unconstrained in the next stable snapshot and may not be constrained in postconditions. In Dash, there is the potential for an invariant (**fact**) to contradict the postcondition of a transition, which means that no transition system exists that satisfies the description.

Stuttering: The semantics of each language define what happens if there is no transition to be taken. If none of the preconditions are satisfied, B takes the generous approach and can make a transition to a state that satisfies the constraints defined in the **INVARIANT** section. In Dash, the non-environmental variables remain unchanged but the environmental variables can change. In TLA$^+$, a transition relation is accompanied by a tuple of variables such that every transition either satisfies the transition relation or leaves those variables unchanged. This admits stuttering transitions that do not affect those variables. In TLA$^+$ this is specified using the `[][Next]_vars` (line 51) notation, where `Next` is the transition relation of the specification and `vars` is a tuple of variables. Lastly, if the system reaches a snapshot where the precondition of no action holds, the system is said to have reached deadlock.

Snapshot Constraints: A snapshot constraint is a constraint on all snapshots that implicitly limits the reachable snapshot space beyond how transitions have already limited it. In B, these are called **PROPERTIES** and are used to limit the behaviour of constants rather than snapshot variables. It is possible to create an inconsistent model through contradicting constraints. In Dash, these are part of an **invariant** block and they can limit any part of the model. In TLA$^+$, there are assumptions, written in the **ASSUME** block. Snapshot and action constraints, written in TLC model settings, can restrict possible snapshots and transitions respectively. In Dash and TLA$^+$, the combination of transitions and snapshot constraints may result in an inconsistent/deadlocked model.

The final column of Table I states the term each language uses as the name of the section describing the properties-to-be-verified (if these go in a particular section) and the kind of properties allowed. In B and TLA$^+$, invariants, which may include type constraints, are part of the properties-to-be-verified; whereas in Dash, invariants and type constraints limit the reachable snapshot space.

D. Scalability

In this section, we consider how well the modelling languages support larger models, which require decomposition. In B, modules can be linked via one of **INCLUDES**, **SEES**, **USES**, **EXTENDS**, **PROMOTES**, and **IMPORTS** sections. By mentioning the name of a machine within these sections of another machine, the machines can interact.

In Dash, the state hierarchy provides a means of decomposing a model. Models parts can be written in multiple files and file concatenation is used to put the parts together. The snapshot declaration itself can be decomposed into parts that appear at different places in the state hierarchy. To reference a snapshot element declared in a different state, its name is prefixed by the state where it is declared. Separate namespaces create interfaces while still using global communication.

TLA$^+$ has a module system, which allows modellers to group related pieces of a specification together. Typically, each module resides in its own file. Modules can extend other modules, using the **EXTENDS** keyword, which brings definitions and declarations into scope. Submodules can be created by nesting one module inside another. TLA$^+$ also has a mechanism for instantiating modules, using the **INSTANCE** and **WITH** keywords, which not only allows for creating multiple instances of a module with different values substituted for the declared variables, but can also be used as a mechanism for variable hiding [13].

V. Related Work

Comparisons between modelling languages are useful to provide users with a means of discerning which modelling language is most suitable for their systems. Some textbooks (*e.g.,* [23], [24], [25], [26]) introduce various formal and informal modelling languages each with their own examples (or occasionally the same example) to illustrate the language. Ardis *et al.* [27] compared a number of formal modelling languages on the same example, however, none of the languages they compared are the three languages we include. The criteria these authors created to compare the languages are much higher level concerns, such as testability and language maturity, than our comparison which focuses on how a model is written in a language. Lamport and Paulson [28] compare specification with and without a type system. Newcombe *et al.* [29] compare TLA$^+$ and Alloy, and conclude that TLA$^+$'s data operations, together with its higher order and recursive operators, make descriptions simpler than Alloy's for engineers particularly with respect to nested record structures. Our comparison focuses more on the semantics of the transition system constructs whereas Newcombe *et al.* focus on usage of the language and its tool support. Sullivan *et al.* [30] compared three methods for modelling transition systems in Alloy (without extensions) for performance.

We chose to compare our new language, Dash, to B and TLA$^+$ because of the similarities in the target audience of the three languages, namely, abstract, formal specification early in the development process. Other similar languages that we could have compared to are: Z [1], VDM [2], Abstract State Machines (ASMs) [31], and Event-B [32], which all describe transition systems abstractly. In particular, we plan to compare to Event-B, which uses "events" (which are not the same as Statecharts' events) as packages of behaviour. Dash is the only language that has native support for events and other common control-oriented abstractions, and at the same time provides rich datatypes for specifying data-oriented models. Even though control state hierarchy can be encoded into these languages (*e.g.,* [33]), the task is not trivial.

Statecharts-based languages and UML state machines [34] provide a graphical manner to describe system behaviour, but do not completely fall into the category of declarative specification languages: their semantics are often not fully formal and they lack support for declaring datatypes. OCL [35] 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 a SMV [36] and Spin [22] are lower-level descriptions than what is often convenient for a user. nuXmv [37] 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. In contrast, in B, Dash, and TLA$^+$ scopes are set at analysis time. Chang and Jackson [38] explored creating a BDD-based model checker for transition systems specified declaratively. One of the major differentiators of Spin is the support for dynamic changes to the number of processes, whereas in most other tools the scope of elements is fixed during the analysis. Many authors have explored translations from descriptions in higher-level modelling languages to the input languages of model checkers.

VI. Conclusion

Each modeller will have their own preferences and each system will have its own characteristics so we can draw few definitive conclusions regarding the three languages we compared. Overall, we found that B's typechecking would catch the most type errors prior to more expensive analysis, however, Dash/Alloy's subtyping is often convenient and natural for those with an object-oriented background. Alloy makes everything a set for simplicity in semantics and tool building, but it is a barrier for a novice user, and the lack of built-in types is unconventional. In B and TLA$^+$, the explicit typing constraints are properties to be checked whereas in Dash, the type constraints limit the reachable snapshot space, which often matches the user's intention for type constraints. TLA$^+$ provides the most flexibility to define the transition relation because users describe it explicitly, but this flexibility adds extra work for them. B and Dash force users to separate pre and postconditions of a transition, which provides clear structure to a model of a transition system. Dash stands out for control-oriented systems where transitions can naturally be factored based on modes and divided into concurrent regions. Modelling independent concurrency (where the synchronization between concurrent states is not obvious) in B and TLA$^+$ would be difficult and less clear. We hypothesize that the ability to model concurrent regions separately from each other is extremely valuable when modelling a distributed system. The importance of each of these factors must be considered relative to the domain of the requirements engineering problem.

In future work, we would like to extend our comparison to include other similar languages and also extend our comparison criteria to include an examination of tool support, which would address issues such as performance, robustness, and ease of understanding counterexamples.

References

[1] J. M. Spivey, *The Z Notation: A reference manual*, ser. International Series in Computer Science (2nd ed.). Prentice Hall, 1992.

[2] C. B. Jones, *Systematic Software Development Using VDM (2nd Ed.)*. Prentice-Hall, Inc., 1990.

[3] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*. Cambridge University Press, Aug. 1996.

[4] D. Jackson, *Software Abstractions*, 2nd ed. MIT Press, 2012.

[5] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *CHARME*, 1999, pp. 54–66.

[6] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[7] J. Serna, N. A. Day, and S. Farheen, "Dash: A new language for declarative behavioural requirements with control state hierarchy," in *International Workshop on Model-Driven Requirements Engineering @ RE*. IEEE Computer Society, 2017, pp. 64–68.

[8] J. Serna, N. A. Day, , and S. Esmaeilsabzali, "Dash: Declarative modelling with control state hierarchy (preliminary version)," Univ. of Waterloo, Cheriton School of Comp. Sci., Tech. Rep. CS-2018-04, 2018.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. of Comp. Prog.*, vol. 8, no. 3, pp. 231–274, 1987.

[10] M. Leuschel and M. Butler, "ProB: an automated analysis toolset for the B method," *Soft. Tools for Tech. Transfer*, vol. 10, no. 2, pp. 185–203, 2008.

[11] A. ClearSy, "AtelierB," *Training Manual*, vol. 2, 2002.

[12] "Xtext," https://eclipse.org/Xtext/, 2017, [Online; accessed 15-June-2018]. [Online]. Available: https://eclipse.org/Xtext/

[13] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002.

[14] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *CHARME*. Springer, 1999, pp. 54–66.

[15] L. Lamport, "The PlusCal algorithm language," in *Theoretical Aspects of Computing*, ser. LNCS, no. 5684. Springer, 2009, pp. 36–60.

[16] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying Safety Properties With the TLA+ Proof System," in *International Joint Conference on Automated Reasoning*, ser. Lecture Notes in Artificial Intelligence, vol. 6173. Springer, Jul. 2010, pp. 142–148.

[17] J. S. Ostroff, "Validating software via abstract state specifications," York University, Tech. Rep. EECS-2017-02, 2017.

[18] N. Nissanke, *Formal Specification: Techniques and Applications*. Springer, 1999.

[19] S. Farheen, "Improvements to transitive-closure-based model checking in Alloy," MMath thesis, Univ. of Waterloo, Cheriton School of Comp. Sci., 2018.

[20] S. Esmaeilsabzali, "Prescriptive semantics for big-step modelling languages," Ph.D. dissertation, Univ. of Waterloo, Cheriton School of Comp. Sci., 2011.

[21] J. Edwards, D. Jackson, and E. Torlak, "A type system for object models," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 189–199.

[22] G. Holzmann, *The Spin model checker: primer and reference manual*. Addison-Wesley Professional, 2003.

[23] N. Nissanke, *Formal Specification: Techniques and Applications*. Springer, 1999.

[24] V. Alagar and K. Periyasamy, *Specification of Software Systems*. Springer, 1998.

[25] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*. Pearson, 2006.

[26] D. Bjoner and M. C. Henson, *Logics of Specification Languages*. Springer, 2008.

[27] M. A. Ardis, J. A. Chaves, L. J. Jagadeesan, P. Mataga, C. Puchol, M. G. Staskauskas, and J. Von Olnhausen, "A framework for evaluating specification methods for reactive systems experience report," *IEEE Trans. on Soft. Eng.*, vol. 22, no. 6, pp. 378–389, Jun. 1996.

[28] L. Lamport and L. C. Paulson, "Should your specification language be typed," *ACM Trans. on Prog. Lang. and Systems*, vol. 21, no. 3, pp. 502–526, 1999.

[29] C. Newcombe, "Why Amazon chose TLA+," *ABZ*, vol. 8477, pp. 25–39, 2014.

[30] A. Sullivan, K. Wang, and S. Khurshid, "Evaluating State Modeling Techniques in Alloy," in *Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*, 2017, pp. 11–13.

[31] E. Börger and R. Stärk, *Abstract state machines: a method for high-level system design and analysis*. Springer, 2012.

[32] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.

[33] E. Sekerinski, "Graphical design of reactive systems," in *International B Conference*. Springer, 1998, pp. 182–197.

[34] "OMG unified modeling language," http://www.omg.org/spec/UML/2.5/PDF/, 2015, [Online; accessed 15-June-2018].

[35] "OMG object constraint specification (OCL) specification," http://www.omg.org/spec/OCL/2.4/PDF, 2014, [Online; accessed 15-June-2018].

[36] K. L. McMillan, "The SMV system," in *Symbolic Model Checking*. Springer, 1993.

[37] 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 *CAV*, ser. LNCS, vol. 8559. Springer, 2014, pp. 334–342.

[38] F. S.-H. Chang and D. Jackson, "Symbolic Model Checking of Declarative Relational Models," in *ICSE*, May 2006, pp. 312–320.