Noname manuscript No.
(will be inserted by the editor)

Dash: Declarative Behavioural Modelling in Alloy with Control State Hierarchy

Jose Serna · Nancy A. Day · Shahram Esmaeilsabzali

Received: date / Accepted: date

Abstract We present Dash, an extension to the Alloy language to model dynamic behaviour using the labelled control state hierarchy of Statecharts. From Statecharts, Dash borrows the concepts to specify hierarchy, concurrency, and communication for describing behaviour in a compositional manner. From Alloy, Dash uses the expressiveness of relational logic and set theory to abstractly and declaratively describe structures, data, and operations. We justify our semantic design decisions for Dash, which carefully mix the usual semantic understanding of control state hierarchy with the declarative perspective. We describe and implement the semantics of a Dash model by translating it to Alloy, taking advantage of Alloy language features. We evaluate our Dash translation and perform model checking analysis, enabled by our translation, in the Alloy Analyzer using several case studies. Dash provides modellers with a language that seamlessly combines the semantics of control-modelling paradigms with Alloy's existing strengths in modelling data and operations abstractly.

Jose Serna
David R. Cheriton School of Computer Sciences
University of Waterloo
Waterloo, ON, Canada, N2L 3G1
E-mail: jserna@uwaterloo.ca

Nancy A. Day
David R. Cheriton School of Computer Sciences
University of Waterloo
Waterloo, ON, Canada, N2L 3G1
E-mail: nday@uwaterloo.ca

Shahram Esmaeilsabzali
David R. Cheriton School of Computer Sciences
University of Waterloo
Waterloo, ON, Canada, N2L 3G1
E-mail: sesmaeil@uwaterloo.ca

1 Introduction

The goal of model-driven engineering (MDE) [43] is to reduce the complexity of software-based systems through the use of models that are more abstract than descriptions in design and code. Analysis engines applied to the models provide feedback on a model's correctness prior to downstream development. Alloy [23, 22] is a popular modelling language with a simple, yet powerful, syntax of relational logic with quantifiers and transitive closure; and automated tool support, which searches for instances and counterexamples within a finite scope. Alloy has been used in many applications (*e.g.,* program verification [41], network protocols [55], security [25], train control [48]) and has a strong user community [1].

Although Alloy is often used for the exploration and analysis of structures in software designs, it can also be used to model dynamic behaviour abstractly. Uninterpreted sets and uninterpreted or semi-interpreted relations are supported, and changes to these relations over time are described declaratively to form a transition system. Temporal logic model checking queries of these transition systems can be specified and evaluated in the Alloy Analyzer using bounded model checking [4, 10], linear temporal logic (LTL) model checking [28] or transitive-closure-based model checking [16].

However, the Alloy language in versions prior to version 6 did not include any explicit language constructs for modelling dynamic behaviour [23]. Two existing extensions of Alloy for modelling the flow of time and change are Electrum [28] (which has now been included in Alloy 6) and DynAlloy [17]. Electrum adds the declaration of time-varying values to Alloy, and LTL as operators to describe system behaviour. DynAlloy extends Alloy with actions that can be composed sequentially, non-deterministically, or iteratively to represent system changes arranged in a manner similar to an imperative program. Missing is an extension to Alloy that allows modellers to describe dynamic behaviour with labelled, hierarchical, concurrent states, which originated with Statecharts [21] and were made popular by UML Statemachines [50].

We present a new extension to Alloy, called Dash, which provides Statecharts-like language abstractions to model dynamic behaviour. Dash extends Alloy with transition declarations, a useful construct to represent change. A transition permits a modeller to describe *when*, and *what* change happens in a model. To describe *when* a transition occurs, Dash provides the hierarchical and concurrent control states of Statecharts and the guards of transitions. To describe *what* changes, Dash provides transition actions and destination control states. The guards and actions of transitions are described abstractly in Alloy, which fits nicely with declarative modelling. Labelled control states allow modellers to name a point in the model's execution with transitions exiting or entering that state. Hierarchy provides a means of concisely grouping behaviours (all states within a state share some common behaviours) and

[1] The community organizes spaces to discuss the future of Alloy, help modellers and developers of the language, and has a dedicated research track in an international conference (see `http://alloytools.org/community.html`).

expressing priority. Concurrent states provide modularity by allowing the description of behaviours that are (mostly) independent. A Dash model is a declarative description of a transition system at the level of first-order logic and set theory. Our work is aimed at Statecharts or UML modellers who seek a way to describe behaviour more abstractly. Modellers who use Alloy to model dynamic systems will also benefit from the abstractions that Dash provides to model complex control-oriented behaviour.

Extending our previous proposal for Dash syntax [46], this paper describes Dash syntax in full (Section 3), and adds the following novel contributions:

- a definition of the semantics of Dash, which seamlessly integrate hierarchical control constructs with declarative modelling (Section 4),
- a method (and tool) for translating Dash models to Alloy (Section 5),
- case studies that show the modelling capabilities of Dash across the spectrum of data- and control-oriented systems, and that verification of Dash models is feasible (Section 6).

In Section 4, we describe the semantic choices for Dash. We address the meaning of concurrency, events, and the frame problem, creating the usual big steps and small steps of Statecharts. The challenge in choosing these semantics is to merge the usual declarative perspective of allowed underspecification (*i.e.,* if not explicitly constrained, non-deterministic change is allowed) with the operational perspective of Statecharts of explicit specification (*i.e.,* if not constrained, implicitly no change is allowed). Guided by our semantic choices, we describe a translation from Dash to Alloy so that no new tools are required for analysis (Section 5). The challenge in developing this translation is to match our semantic choices for Dash without creating extra state space. We describe how we exploit Alloy language features to model the control state hierarchy of Dash. Our translator is implemented using Xtext [53] and is available on-line at the Dash website[2], so anyone can try writing a Dash model and use the publicly available Alloy Analyzer to check properties of their models.

We evaluate Dash by comparing examples of Dash models of dynamic systems with either hand-crafted Alloy models or the equivalent Alloy models resulting from our translation (Section 6). Our case studies include a large model from an avionics software development project. To evaluate the analysis enabled by our translator, we model checked several properties of the models. Our cases studies show that Dash can model systems across the data-oriented to control-oriented spectrum concisely; that useful results can be achieved by model checking Dash models in Alloy; and that the counterexamples produced by Alloy can be understood in terms of the Dash model. The case studies are available on-line at the Dash website.

With the creation of Dash, we enable the modelling and automatic analysis of models constructed using common and useful constructs from both declarative languages and control-oriented languages.

[2] http://dash.uwaterloo.ca:8080/

2 Background

2.1 Alloy

Alloy is a popular modelling language based on first-order logic and set theory. Using a finite model finder called Kodkod [49], given finite sizes (scopes) for each set, the Alloy Analyzer automatically searches for instances (values for the sets and relations) that satisfy a model's constraints. In an Alloy model, a set is described using a signature and elements of a signature are called atoms. Relations between this set and another set are specified as fields within the signature:

```
1  abstract sig A {}     // a set called A
2  sig C, D {}           // a set called C and a set called D
3  sig B extends A {     // a subset of A called B
4    R1: C,              // a relation from B to C
5    R2: C -> lone D     // a relation from B to C to D
6  }
```

Alloy uses keywords such as `no` (empty), `lone` (at most one element), `one` (exactly one element), `some` (one or more elements), and `set` (any number of elements, including zero) to constrain the multiplicity of relations and expressions. If no multiplicity is declared the default `one` applies. An Alloy signature that extends another signature is a subset and is called a subsignature. All the immediate subsignatures of a signature are disjoint[3]. A signature can be declared as `abstract` meaning that the set can only contain atoms that are in the subsignatures. Constraints in Alloy are described in facts such as:

```
1  fact {
2    // at least one c for every b in R1
3    all b: B | some c in b.R1
4  }
```

The expression `b.R1` conveniently looks like the `R1` field of `B`'s record/class, but is actually using the join operator (`.`) to take the range of the pairs in `R1` that have `b` as their first element[4]. The association of relations directly with signatures gives the Alloy language an object-oriented flavour, although there is no means of relating behavioural changes with the signature. Alloy provides common set operations on relations and functions (such as join, union, *etc.*), and goes beyond first-order logic by including the transitive closure operator (which can be computed for a finite set):

```
1  C + D    // union of C and D
2  C - D    // difference of C and D
3  C & D    // intersection of C and D
4  ^R1      // transitive closure of R1
```

Alloy facts can be decomposed into predicates and functions that take arguments. The Alloy Analyzer produces a visual representation of a satisfying instance when one can be found.

[3] Non-mutually disjoint subsets can also be modelled.

[4] Technically, Alloy has no scalars, so `b` is a singleton subset of `B`.

Modelling a transition system in Alloy is accomplished by creating a set of states and constraining a binary relation over these states to be the transition relation. The transition relation can be iterated to do bounded model checking (BMC) [4, 10]. Via Electrum (which is now part of the Alloy Analyzer as of version 6), dynamic models can be translated to nuXmv [28]. For the evaluation of model checking properties of Dash models, we use transitive-closure-based model checking (TCMC) [16], in which the meaning of all temporal operators in Computation Tree Logic with fairness constraints (CTLFC) [8] are described in terms of Alloy's transitive closure operator.

2.2 Statecharts

Statecharts [21] is a graphical formalism for modelling the transition systems of reactive (meaning interactive with their environment), non-terminating systems, with control-oriented behaviour. Control states have labels and are represented as nodes in a graph, and transitions are represented as arrows entering or leaving a state. Figure 1 is the Statecharts representation of a two bit counter (the two bit counter example is described in more detail in Section 3). Statecharts introduces three useful abstractions as constructs for modelling.

- A **control state** represents an equivalence class of executions with the same possible future behaviours in the system. Control states with no decomposition are called **basic states**. For example, in Figure 1 the states `Off` and `On` inside the state `Bit1` are both basic states[5].
- An **OR-state** is a control state that is a parent of one or more other states. For a modeller, this abstraction allows them to group concepts together with related behaviours. The hierarchy is also a means of expressing priority as the transitions exiting states higher in the hierarchy have priority over transitions exiting child states.
- An **AND-state** is a control state that is a parent to a group of states that together operate independently from sister states to their parent state[6]. AND-states (also called concurrent states/components) can communicate with each other but can take transitions independently. For example, in Figure 1, `Bit1` and `Bit2` are concurrent states that operate independently, but they communicate through the event `tk1`, which is generated when transition `t2` is taken and triggers transition `t3`. Two transitions are **orthogonal** if they are contained in different concurrent components.

In Statecharts, default states define the states the system is in upon initialization. Default states are represented by a transition to a state from a black dot. In Figure 1, the default states are `Bit1_Off` and `Bit2_Off`.

[5] In the rest of this article, we concatenate the names of states in the hierarchy to refer to a particular state. For example, to refer to the state `On` inside the state `Bit1` we use `Bit1_On`.

[6] Our terminology is slightly non-standard: in Statecharts variants, there is often a parent AND-state with OR-state components that execute concurrently. In Dash, we have eliminated the need for the parent state and label components that run concurrently with each other as concurrent states.

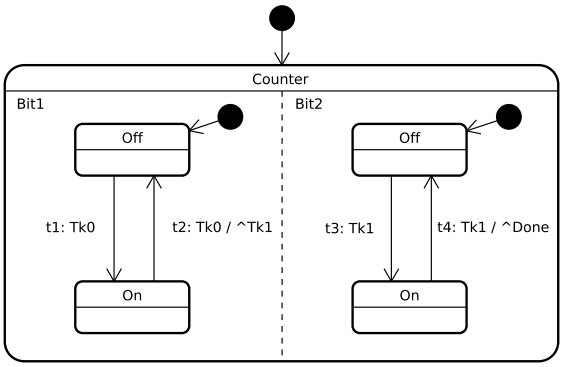

Fig. 1: Statecharts representation of a two bit counter.

Statecharts are a very popular modelling notation with many variations in its semantics [13, 2]. Because of the concurrent states, the semantics are described in terms of big and small steps. A big step consists of multiple small steps (individual transitions) that are the combined reaction of the model to an environmental input (taking a transition in one concurrent component can trigger transitions in another concurrent component). A version of Statecharts has been incorporated in UML statemachines [50]. Modellers find the labelling and hierarchy of states to be very useful abstractions to decompose the control-oriented behaviour of a system.

3 Dash

Dash extends Alloy with features for creating behavioural models. The fundamental constructs in Dash are the description of a state hierarchy, a set of transitions, and the initial constraints. In the next few paragraphs, we describe these constructs in Dash. Some additional syntactic constructs that facilitate the concise description of behavioural models in Dash are briefly described at the end of this section.

We explain Dash syntax and its features using two running examples. Each example highlights different features of Dash. The first example is the game musical chairs (adapted from [35]) and we use it to help in the presentation of state hierarchy, snapshot variables, events, and transitions. Musical chairs is a classical children's game where players dance to music around a set of chairs, eliminating players and chairs in each round until there is only one player sitting on a chair at the end of the game. Figure 2 shows part of a Dash model for the game musical chairs. On line 2 a concurrent state called `Game` is created, which is the root state of the hierarchy. A Dash model may have multiple root states, but all root states must be declared as concurrent components. Concurrent components are described with the `conc` keyword. Nested within `Game` there are control states `Start` (the default state on line

```
1  sig Chair, Player {}
2  conc state Game {
3    players: set Player          // snapshot variables
4    chairs: set Chair
5    occupied: Chair set -> set Player
6    env event MusicStarts {}     // events
7    env event MusicStops {}
8    init {                       // initial constraints
9      #players > 1
10     #players = (#chairs).plus[1]
11     occupied = none -> none    // empty relation
12   }
13   default state Start {...}    // default state
14   state Walking {
15     trans Sit {                // transition
16       on MusicStops            // event trigger
17       goto Sitting             // dest state
18       do {                     // action
19         occupied' in chairs -> players
20         chairs' = chairs
21         players' = players
22         // occupied is a total function
23         all c : chairs' | one c.(occupied')
24         // occupied is an injective function
25         all p : Chair.(occupied') | one occupied'.p
26       }
27     }
28   }
29   state Sitting {...}
30 }
```

Fig. 2: Dash model for musical chairs.

13), `Walking` (line 14), and `Sitting` (line 29), which represent the phases of the game. States can be arbitrarily nested in the state hierarchy.

The second running example, described in Dash in Figure 3 and a Statecharts representation in Figure 1, is a two bit counter (adapted from [13]) that we use to describe features such as concurrency, namespaces, and events. In the bit counter example, the concurrent control states `Bit1` and `Bit2` represent the least and most significant bits of the counter, respectively. Clock ticks are modelled using the environmental event `Tk0` and every time this event is received the internal count of `Bit1` is incremented. After an even number of ticks, `Bit1` sends the event `Tk1` and then `Bit2` increments its internal count. Finally, after four ticks `Bit2` sends the event `Done` to signal the end of the counting process. An environmental event represents input that is generated by the environment and can trigger transitions. Variables can also be declared as environmental, which means the model does not control their values and their values can change non-deterministically at the end of a big step. An

event or variable is declared to be environmental using the keyword `env`[7]. An internal event is generated by the model and an internal variable is controlled by the model.

```
1  conc state Counter {
2    env event Tk0 {}
3    conc state Bit1 {   // concurrent state declaration
4      event Tk1 {}      // local event
5      default state Bit1_Off {}
6      state Bit1_On {}
7      trans T1 {
8        from Bit1_Off
9        on Tk0
10       goto Bit1_On
11     }
12     trans T2 {
13       from Bit1_On
14       on Tk0
15       goto Bit1_Off
16       send Tk1
17     }
18   }
19   conc state Bit2 {
20     event Done {}
21     default state Bit2_Off {}
22     state Bit2_On {}
23     trans T3 {
24       from Bit2_Off
25       on Bit1/Tk1   // element qualified name
26       goto Bit2_On
27     }
28     trans T4 {
29       from Bit2_On
30       on Bit1/Tk1
31       goto Bit2_Off
32       send Done
33     }
34   }
35 }
```

Fig. 3: Dash model of a two-bit counter.

A snapshot is a mapping from variables to values, that changes as the model takes transitions (steps). A variable of a snapshot can consist of any type of value representable in Alloy. `Chair` and `Player` are uninterpreted sets introduced in Figure 2, line 1. Declarations within a state are variables that are part of the snapshot (*i.e.,* they can change value during the execution of a model). In musical chairs, the set of active players, active chairs, and the relationship between chairs and players (*i.e.,* who is sitting where) are the

[7] Our translator raises an error if a model tries to constrain the next value of an environmental variable or generate an environmental event.

snapshot variables (lines 3–5). The events of the music starting and stopping are declared on lines 6 and 7; these are declared to be environmental. Initial constraints on the variables are shown on lines 8–12 of the musical chairs model (Figure 2) and are described in pure Alloy syntax. These constraints ensure that when the game starts there is more than one player, there is one less chair than players and no one is sitting down.

Transitions are described within a `trans` block following the template:

```
1  trans <label> {
2    from <src_state>
3    on <trigger_event>
4    when <guard_condition in Alloy>
5    goto <dest_state>
6    do <action in Alloy>
7    send <generated_event>
8  }
```

These keywords are chosen to match the way a transition is described in English. An example transition is on lines 15-27 of the musical chairs model in Figure 2. Each component of a transition is optional and understood within its context. Transition `Sit` omits the `from` part of the transition and its source state is understood to be `Walking`. The action of the transition (`do`) is any formula in Alloy. Following the common Z style [47], unprimed variables are the current values of snapshot elements and primed variables are the variable values in the next snapshot[8]. For example, the formula on line 23 of Figure 2, means that every chair must have someone sitting on it in the next snapshot. There is no need to state all the possible combinations of which player could sit on which chair. This is an example of the conciseness and abstraction of declarative modelling, in contrast to typical control-oriented languages where the action is limited to being a sequence of assignments. The guard condition (`when`) is any formula in Alloy but may only refer to unprimed snapshot variables. We refer to the source state, guard condition, and the event trigger together as the pre-condition of a transition; and the action, generated event, and destination state as the post-condition.

Figure 3 shows how the nesting of control states in Dash matches the Statecharts hierarchy of Figure 1 with `Bit1` and `Bit2` being concurrent components. State blocks define namespaces in Dash. A reference to a variable from another state must be prefixed by its home state as on lines 25 and 30 in Figure 3. While the semantics use global communication (as in most Statecharts languages), supporting namespaces means that duplicate names are not an issue and modellers are aware of locality. A transition can generate an event using the keyword `send` as on line 32.

Many times it is useful to describe invariants[9] about a snapshot of the system as part of the behavioural specification. In Dash, we can describe these

[8] In the latest release of Alloy, which incorporates Electrum, prime has a special meaning, thus we would have to change our use of prime. For example, instead of `s'` we would rewrite it to `s_next` to be compatible with the newest version of Alloy.

[9] We mean invariants that are part of the model's behaviour not properties to check of the model.

constraints within an `invariant` block, and they hold whenever the state that contains the invariant declaration is active. For example, in the musical chairs game an invariant of the `Walking` state that there are no occupied chairs while the players are walking could be declared as:

```
1  state Walking {
2    invariant NoOccupiedChair {
3      no occupied
4    }
5  }
```

Dash offers some syntactic sugar to ease the description of the behaviour of a system. These syntactic constructs are meant to help in code reuse, systematic organisation, and facilitate the decomposition of a model based on different factors. A set of transitions can be described in a single statement using **transition comprehension**. For example,

```
1  trans to_error {
2    from * on error goto ErrorState
3  }
```

describes a set of transitions, one from every child state of the state that contains the transition comprehension declaration, that each go to the `ErrorState` on an `error` event. Using **addons**, part of the definition of a transition can be described in a different part of the model, similar to aspect-oriented modelling [12]. These addons are layered together to get the full description of a transition. For example:

```
1  addon (do incErrorCounter) to (from * to ErrorState)
```

adds the action `incErrorCounter` to every transition whose destination is the `ErrorState`. Another feature is **transition templates**, which capture similarities in transitions to avoid duplication in a model. A template is a parameterized definition of a transition that can be instantiated. Also, after recognizing the role that control states play in factoring snapshots into groups that have the same possible future behaviours, we realized that transitions can also be **factored by events and conditions**. There are models where control states are not a natural way to describe the behaviour and for these models labelled states can be omitted (except for the root state) and events and conditions can be used to structure the set of transitions. In these cases, the transitions are described within an `event` or `condition` block. These factoring blocks can be nested within each other any number of times to represent complex behaviours. Factoring offers a mechanism to systematically organise the transitions in a model and accommodates different modelling paradigms (*e.g.,* event-based modelling).

4 Semantic Decisions

Choosing a semantics for Dash must combine the meaning of the Alloy formulas used in the guards and actions of transitions with a semantics for Stat-

Table 1: Semantics Decisions for Dash.

Semantic option	**Choice in Dash**
Concurrency	Single
Big step maximality	Take one
Event lifeline	Present in remainder of big step
Variable lifeline	Immediate change in small step
Priority	Source state outer hierarchical

echarts that has understandable properties for users. In choosing our semantics, we seamlessly integrate the declarative perspective for data operations, which allows underspecification, with the operational perspective of Statecharts, which prescribes exact specification and has concurrency.

There are different semantic definitions for languages within the Statecharts family each with its own characteristics [2, 13, 29, 36]. Because of concurrency, semantics for hierarchical control states are usually given in terms of a **big step**, which is a representation of how a system takes multiple transitions in reaction to environmental input. A big step consists of one or more **small steps**, each of which can be one or more transitions. A big step continues until the model is **stable**, *i.e.,* no more transitions are enabled. At a stable snapshot, more environmental input (events and changes to variables) is needed to enable transitions. A transition is enabled if the system is in its source state, its trigger event is in the set of current events and its guard condition is satisfied. Transitions in multiple concurrent states may respond to the same environmental input (*i.e.,* occur in the same big or small step), thus the semantics of Dash must address the question of which transitions can be taken together.

We rely on the semantic framework of Esmaeilsabzali *et al.* [13], which describes a space of semantic aspects and options for this family of languages, to guide our semantic decisions for Dash. Our choices for each of the semantic options are described in Table 1, and are based on two reasons:

R1: As a declarative model, a transition action can describe a "large" change (*i.e.,* a sequence of operations in an action is rarely needed); and
R2: Users from both the declarative and control-oriented modelling approaches should find the semantic choices for Dash intuitive.

The semantic aspect Concurrency determines how many transitions can be taken in a small step. The option Single for this aspect means that only one transition can be taken in a small step to ensure transition atomicity. This choice is because of reason R2 since race conditions, which can occur when two transitions modify the same variable, can make a model inconsistent, and are difficult to debug.

The big step maximality aspect specifies the termination criteria for a sequence of small steps, *i.e.,* when a system is stable. One concurrent component can generate events that cause transitions to be enabled in another concurrent component, which are taken later in a big step. We choose the op-

tion Take One, meaning that at most one transition per concurrent state can be taken in a big step (*i.e.,* all transitions in a big step must be orthogonal to each other). For reason R2, this choice provides a guarantee of termination of big steps. In the bit counter example (Figure 1), without the Take One semantic choice, when the environment generates the event `tk0` the model could engage in the following non terminating big-step `{t1, t2, t1, t2, t1, t2, ...}`. However, the Take One semantic choice forces the model to take either `t1` or `t2` but not both in a single big step. Because of reason R1, taking at most one transition in each concurrent state in a big step is unlikely to be a limitation.

For the EVENT LIFELINE aspect, we choose the option Present in Remainder of big step where a generated event can trigger transitions in the small steps after its generation. For reason R2, we want the small steps to be causal, meaning an event is generated before it triggers another transition within the big step.

For the VARIABLE LIFELINE, we choose to make the effects of the actions of a transition immediately available in the next small step to enable transitions, permitting a cascading flow of variable changes. Because of Take One for big step maximality, this choice for variable lifeline cannot cause a non-terminating big step where two transitions keep enabling each other (reason R2).

For PRIORITY, we prioritize the transitions based on the hierarchy of the source states. Transitions leaving a parent state have priority over those leaving a child state. This local choice is easier to understand than priority based on the lowest parent that includes both the source and destination state (reason R2).

Finally, we have to address the **frame problem** [31] where there is a mismatch between the usual semantics of declarative and control-oriented languages. In declarative languages, if a variable is not constrained in an action, it can change non-deterministically. In control-oriented languages (where actions are typically a sequence of assignments), an unchanged variable retains its value from the previous snapshot. We choose a middle ground between these two perspectives favouring conciseness of description. In Dash, a variable declared as `env` is allowed to change when the system is stable, but otherwise it retains its value. For a non-environmental variable, if its primed version is mentioned in the action of a transition, we assume the action constrains it; if its primed version is not mentioned in the action then we require that the variable retains its value from the previous snapshot. If the user does not like this default semantic choice, it can be overridden by toggling an option in our translator, thus allowing variables not mentioned in the transition's action to change non-deterministically in a transition.

Based on Esmaeilsabzali *et al.* [13], the set of semantic values we choose for Dash results in the semantics of Dash being *cancelling*, *non-deterministic*, and *priority consistent*. The semantics of Dash are cancelling because it is possible for a transition to be enabled during a big step and then become disabled by the effects of other transitions taken during the same big step. For example, the trigger condition of a transition `t` may evaluate to true at the beginning

of a big step, but after one or more small steps are taken, the cumulative effects of the actions of the taken transitions may make the guard condition of `t` false, disabling the transition. The non-determinism in the semantics of Dash means that if the same environmental input is given to two big steps of a model that have the same starting snapshot, the ending snapshots may be different, even if both big steps execute the same set of small steps and no transition is cancelled. The difference arises because of the *order* of execution of the small steps affects the cumulative effects of the actions of the transitions taken. Finally, the semantics of Dash are priority consistent, meaning that transitions that have higher priority are always taken before other transitions with lower priority. We believe these properties make Dash models easy to understand for users.

5 Translating Dash to Alloy

We use the semantic decisions of the previous section to define a translation of a Dash model to an Alloy model for formal analysis. Because Dash uses the Alloy language for describing transition guards and actions, a Dash modeller is expected to have knowledge of the Alloy language and Analyzer. Our goals are: 1) to utilize features of the Alloy language as much as possible to produce a concise representation of a Dash model's behaviour; 2) to avoid introducing extra state space in the translation; and 3) to create a mapping that will make it as easy as possible for a user to understand counterexamples from Alloy in terms of the original Dash model (which is evaluated in Section 6). In general, it is easier to map control states into a first-order language than it is to map first-order constructs into a mostly propositional language (see [16] for a comparison of modelling in Alloy vs NuSMV [7]) therefore we choose to map Dash to Alloy rather than map Dash to a model checking language such as SMV [32]. Our translator is fully automatic and implemented in Xtext [53], which provides robust editing tools.

Dash snapshots are translated into Alloy as a set of snapshots with relations that link each snapshot to its variable values. The snapshot[10] for the musical chairs model (Figure 2) is:

```
1  sig Snapshot  {
2    Game_players : set Player,
3    Game_chairs : set Chair,
4    Game_occupied : Chair set -> set Player,
5    conf: set StateLabel,       // active control states
6    events: set EventLabel,     // events
7    taken: set TransitionLabel, // transitions taken
8    stable: one Bool            // indicates big step boundaries
9  }
```

[10] The actual translated model differs slightly because an Alloy module is used that contains reusable definitions, and we perform some optimizations when generating the Alloy code (described at the end of this section).

In addition to relations for the model's variables, a snapshot also includes values for the set of control states called its configuration (`conf`), the set of events (`events`), a history variable of the transitions that have been taken in a big step (`taken`), and a boolean flag to indicate whether a snapshot is stable (`stable`). The set of taken transitions is needed to determine which transitions can be take in the rest of the big step (see later explanation).

We utilize Alloy's subtyping to define the control state hierarchy. The Alloy representation of the control state hierarchy of the bit counter model (Figure 3) is:

```
1  abstract sig StateLabel {}  // base type of all control states
2  abstract sig Counter extends StateLabel {}
3  abstract sig Bit1, Bit2 extends Counter {}
4  one sig Bit1_Off, Bit1_On extends Bit1 {}
5  one sig Bit2_Off, Bit2_On extends Bit2 {}
```

An abstract signature `StateLabel` is the base type for all control states. The relation `conf` contains elements of type `StateLabel` to determine the control states of the snapshot. On line 2, the control state `Counter` is declared to extend `StateLabel`. All non-basic control states are declared as abstract. The concurrent components `Bit1` and `Bit2` are declared as abstract subsignatures of `Counter`. Concrete (*i.e.,* non-abstract) signatures are used for basic control states. The keyword `one` means that a signature is a *singleton* set, meaning it contains only one atom and this atom is distinct from the atoms in other singleton signatures. These subsignatures directly match the meaning of the control state hierarchy. For example, if the system is in state `Bit1_Off`, it is also in state `Bit1` because of the subtype hierarchy. Thus, we can check if a state is in the current snapshot without searching through its ancestors or descendants resulting in a very succinct method of encoding the control state hierarchy in Alloy.

Events that are declared environmental (*i.e.,* using the `env` Dash keyword) are made subsignatures of an `EnvironmentEvent` signature. All other events are declared as part of an `InternalEvent` signature. The event declarations for the bit counter model are:

```
1  // base type of all events
2  abstract sig EventLabel {}
3  // base type of all env events
4  abstract sig EnvironmentEvent extends EventLabel {}
5  // base type of all internal events
6  abstract sig InternalEvent extends EventLabel {}
7  one sig Tk0 extends EnvironmentEvent {}
8  one sig Tk1, Done extends InternalEvent {}
```

The identifiers of transitions are modelled as signatures. They all extend the base signature `TransitionLabel`, as in the following fragment of the bit counter model:

```
1  abstract sig TransitionLabel {}
2  one sig T1, T2, T3, T4 extends TransitionLabel {}
```

The initial generic constraint on snapshots is that the system is in its default states, no transitions have been taken, and there are no internal events

(which is checked by taking the intersection (`&`) of the events of the snapshot and the set of internal events). Environmental events can be present in the initial snapshot in order to enable transitions. Additional initial constraints defined by a user are added to the model. For example, the initial constraint for the musical chairs model is:

```
1  pred init[s: Snapshot] {
2    s.conf = { Game_Start }
3    no s.taken
4    no s.events & InternalEvent
5    // model specific constraints
6    #s.Game_players > 1
7    #s.Game_players = (#s.Game_chairs).plus[1]
8    s.Game_occupied = none -> none
9  }
```

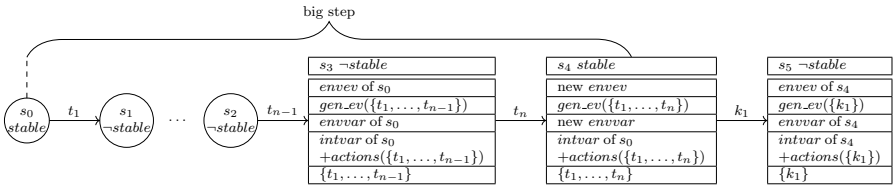

Fig. 4: Snapshots in a big step.

The semantics of a Dash model is a next snapshot relation containing pairs that are the possible **small steps** of a system. A modeller can check temporal properties at the big step boundaries by checking the property only when the system is stable. Some representations of Statecharts semantics use a **reset** step (not involving a transition being taken) between the end of one big step and the beginning of the next big step to clear the outputs generated in a big step, and to prepare the system for the next environmental input (*e.g.,* [27, 36]). Having a reset step in SMV has little penalty in the performance of model checking, however, in Alloy it increases the snapshot space with the extra reset snapshots, which significantly degrades the performance of model checking. Thus, in our translation of Dash models to Alloy, we avoid having a reset step through careful declarative specification of the semantics of a small step. Figure 4 shows a sequence of snapshots that are representative of the next step relation defined by a Dash model. The nodes represent snapshots, and details on the information contained in the relations[11] linked to the snapshot is given in tabular form. An arrow connecting two snapshots represent a small step. Each small step takes one transition (due to the choice of Single for CONCURRENCY). Snapshots that are **stable** are characterised by having:

[11] Environmental and internal events/variables are stored in the same relation, but separated for clarity in the diagram.

- an unconstrained set of environmental events (*envev* in Figure 4) that can trigger transitions in the next big step;
- internal events that were generated (*gen_ev* in Figure 4) by all the transitions in the last big step;
- unconstrained environmental variables values (*envvar* in Figure 4) that can trigger transitions in the next big step;
- internal variable values (*intvar* in Figure 4) that have the accumulated effects of all actions (*actions* in Figure 4 where + is used informally) of the transitions taken so far; and
- the set of transitions taken in the last big step.

The values of environmental events and variables do not change during the small steps of a big step. However, these values are allowed to change non-deterministically in stable snapshots ("new" in Figure 4). We do not show the changes in the `conf`, which include exiting the source control states of the taken transition and entering its destination states.

Next, we describe how the small step relation for a Dash model is defined in Alloy as the disjunction of predicates that describe each transition of the model. For example, the small step relation for musical chairs is:

```
pred small_step[s, s': Snapshot] {
  // naming convention:
  // RootState_EnclosingState_TransitionName
  Game_Start_Walk[s, s'] or
  Game_Start_DeclareWinner[s, s'] or
  Game_Walking_Sit[s, s'] or
  Game_Sitting_EliminateLoser[s, s']
}
```

Only one of these transition predicates will be true in a small step because each predicate contains a semantic constraint to enforce the Single semantic option. We outline the definition of a transition predicate abstractly in the following paragraphs.

For each transition, a predicate with the same name of the transition is created that combines the pre, post, and extra semantics predicates for the transition. For example, the predicate `t1` combines the pre, post, and semantics predicates for transition t_1, meaning that for t_1 to be taken, its pre-condition (`pre_t1`), post-condition (`post_t1`), and semantics predicates (`semantics_t1`) must hold[12]:

```
pred t1[s, s': Snapshot] {
  pre_t1[s]
  post_t1[s, s']
  semantics_t1[s, s']
}
```

The predicate for the pre-condition of a transition t_1 is evaluated relative to the current snapshot, s:

[12] These constraints are in addition to any other constraint declared elsewhere in the model. In Dash and in Alloy, it is possible to write conflicting constraints either implicitly (*e.g.,* in the declaration of multiplicity of relations), or explicitly (*e.g.,* using a fact) that make the model unsatisfiable.

```
1  pred pre_t1[s: Snapshot] {
2    src_state_t1 in s.conf
3    guard_cond_t1[s]
4    s.stable = True => {
5      // beginning of a big step
6      // transition can be triggered only by env events
7      trig_events_t1 in (s.events & EnvironmentEvent)
8    } else {
9      // intermediate snapshot
10     // transition can be triggered by any type of event
11     trig_events_t1 in s.events
12   }
13 }
```

This precondition is true if the source state of t_1 is in the snapshot's configuration (line 2) and the guard of t_1 evaluates to true for the snapshot's variable values (line 3). The evaluation of the presence of t_1's trigger event (lines 4–12) depends on if the snapshot is at the beginning of a big step or not (*i.e.,* stable or not). When the snapshot is stable, t_1's trigger event must be one of the new events from the environment (line 7); otherwise its event must be in the snapshot's set of events (line 11), which include the environmental events generated at the beginning of the big step and the internal events generated so far in this big step. Similarly, in the first step of a big step, the guard (`guard_cond_t1`) is evaluated with respect to potentially new environmental variable values because these are already in the snapshot.

The predicate for the post-condition of the transition t_1 is evaluated relative to the current snapshot, s, and the next snapshot, s':

```
1  pred post_t1[s, s': Snapshot] {
2    s'.conf = s.conf - exit_src_state_t1 + enter_dest_state_t1
3    action_t1[s, s']
4    testIfNextStable[s, s', t1, gen_events_t1] =>
5      s'.stable = True
6      s.stable = True => {
7        // big step = one small step
8        // only internal events are the ones generated by t1
9        // allow env events to change
10       s'.events & InternalEvent = gen_events_t1
11     } else {
12       // last small step of the big step
13       // add t1's gen events to the internal events
14       // allow env events to change
15       s'.events & InternalEvent =
16         gen_events_t1 + (InternalEvent & s.events)
17     }
18   } else {
19     s'.stable = False
20     env_vars_unchanged_t1[s, s']
21     s.stable = True => {
22       // first small step of the big step
23       // only internal events are those generated by t1
24       s'.events & InternalEvent = gen_events_t1
25       // env events stay the same
26       s'.events & EnvironmentalEvent =
27         s.events & EnvironmentalEvent
```

```
28      } else {
29        // intermediate small step
30        // add t1's gen event to the events
31        // env events don't change
32        s'.events = s.events + gen_events_t1
33      }
34    }
35  }
```

The postcondition is true if the configuration changes between s and s' to exit the source states of t_1 and enter the destination states of t_1 (line 2). On line 3, the variable values for the internal values are updated according to the actions of the transition enforcing our semantic choice for VARIABLE LIFELINE of Immediate change in small step. Within this constraint, internal variables whose primed versions are not mentioned in the action are required to retain their values from the previous snapshot (in keeping with the chosen semantics for the frame problem). Next, we have four cases depending on whether s is stable and whether s' will be stable. We have documented these cases in comments on lines 4–34. The constraints on `s'.events` enforce the choice of Present in reminder of big step for EVENT LIFELINE. On line 20, environmental variables are constrained to keep their previous values when the next snapshot is not stable; otherwise, they are allowed to change. The predicate `testIfNextStable` determines whether any transitions will be enabled in s' if t_1 is taken and we discuss it after the explanation of the semantics predicate.

The semantics predicate for t_1 is true if t_1 is orthogonal to all transitions in the set of transitions already taken in this big step, enforcing the choice of Take one for BIG-STEP MAXIMALITY. Two transitions are orthogonal, if they are contained in different concurrent components. For example, in the bit counter transition `T1` is orthogonal to transitions `T3` and `T4`. This predicate may also include priority-related constraints when necessary. If two transitions have source states related in the hierarchy (*e.g.,* one transition's source is an ancestor or descendant of the other's), then we include the negation of the pre-condition of the higher priority transition in this semantics predicate to enforce the choice of Source state outer hierarchical for the PRIORITY semantic aspect. Additionally, if the snapshot s is stable, then this is the first step of a big step and only t_1 should be included in the set of transitions; otherwise, t_1 is added to the set of transitions taken. Keeping the history of transitions taken ensures that only one transition is taken in a step (enforcing the Single semantic choice for CONCURRENCY). The semantics predicate is as follows:

```
1  pred semantics_t1[s, s': Snapshot] {
2    (s.stable = True) => {
3      s'.taken = t1             // SINGLE semantics
4    } else {
5      s'.taken = s.taken + t1   // SINGLE semantics
6      orthogonal_t1[s.taken]    // TAKE ONE semantics
7    }
8    !pre_t2[s] // higher priority transitions are not enabled
9    !pre_t3[s]
10   ...
11 }
```

The predicate `testIfNextStable` is true if the next snapshot is stable after taking this transition so it relies on `enabledAfterStep` predicates for each transition. It is defined as:

```
1  pred testIfNextStable[s, s', t, genEvents]  {
2    not enabledAfterStep_t1[s, s', t, genEvents] and
3    not enabledAfterStep_t2[s, s', t, genEvents] and ...
4  }
5
6  pred enabledAfterStep_t1[s, s', t, genEvents] {
7    src_state_t1 in s'.conf
8    guard_condition_t1[s']
9    (s.stable = True) => {
10     // only transition taken in big step so far is t
11     // so as long as t1 is orthogonal to t
12     orthogonal_t1[t]
13     // and t1 can be triggered by environmental events of the
14     // big step or by any events generated by t
15     trig_events_t1 in
16          {(s.events & EnvironmentalEvents) + genEvents}
17   } else {
18     // as long as t1 is orthogonal to t + s.taken
19     orthogonal_t1[t + s.taken]
20     // t1 can be triggered by any events present in the big
21     // step or any events generated by t
22     trig_events_t1 in (s.events + genEvents)
23   }
24 }
```

The constraints on lines 7 to 8 are similar to the constraints of the pre-conditions for t_1, however, here they depend on the s' to simulate the effects of executing t_1. The constraints on lines 9 to 23 test whether taking t will make it possible to take t_1 in the next step.

Through the use of Alloy's subtyping to represent the control state hierarchy and careful decomposition of the predicates, we avoid introducing any unnecessary atoms, steps, and snapshots in capturing the semantics of Dash in Alloy. Alloy's declarative nature makes it possible to place constraints on the source and destination snapshots of a small step together.

Our translation is optimised to exclude parts of the configuration and definitions when they are unnecessary. If a model does not declare any event to trigger transitions, the events relation is removed from the snapshot signature definition. If a model does not have concurrency, every snapshot is stable, making every small step equivalent to a big step. This case greatly simplifies the constraints of the post-conditions of transitions, and the predicates to determine if a next snapshot is stable are no longer needed. Additionally, the stable flag is removed from the snapshot signature. These simplifications are automatically performed based on static analysis of a model.

6 Case Studies

We use several case studies[13] across the control-oriented and data-oriented spectrum to demonstrate our translation of Dash to Alloy and the model checking analysis of Dash models. Figure 5 places our case studies on a spectrum from data-oriented to control-oriented, and Table 2 summarises some of the characteristics of the models. Data-oriented models have few control states and a flat state hierarchy but richer operations on data. Control-oriented models have many control states and a deep state hierarchy. Most modelling languages are geared towards one or the other type of models. We investigate models spanning the data-oriented to control-oriented spectrum because Dash provides modellers with the ability to use both the abstract data modelling features of Alloy and the control-oriented structuring of Statecharts in one model. Our most significant case study is a partial model of the mode logic of the **NASA Flight Guidance System (FGS)** [9]. We model its flight mode logic subsystem without the event processing.

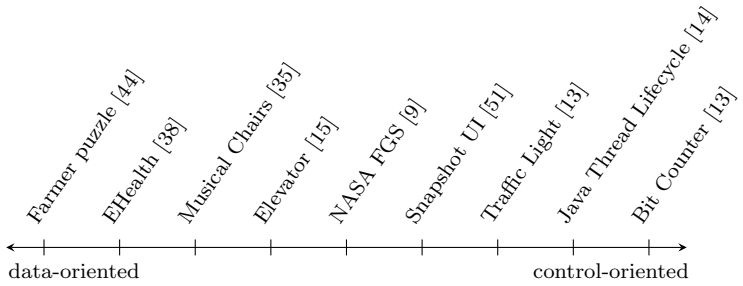

Fig. 5: Model spectrum.

Evaluating Dash Models and their Translations. We show the sizes of the Dash models and their translation to Alloy in Table 3 with respect to source lines of code (SLOC) (skipping comments, blank lines and statements of properties). Dash models are considerably more concise compared to their equivalent translation in Alloy. For example, the Java Thread Lifecycle model in Dash has roughly one fifth of the SLOC of the generated Alloy model. One of the major causes of the difference in terms of SLOC between Dash and Alloy is concurrency. The models that use concurrency (NASA FGS, Traffic Light and Bit Counter) are far more concise in Dash than in Alloy, which is explained by the constraints needed in the Alloy models to handle the semantics of big steps. We had access to a few hand-crafted Alloy models (prepared prior to this work). Some of these are slightly smaller than our Dash models, and these are all smaller than our generated Alloy models, however, the Dash translation has to cover the generality of all the variations of control state

[13] The case studies are available as sample models at http://dash.uwaterloo.ca:8080/.

Table 2: Characteristics of case studies. Conc: concurrent components, D: depth of state hierarchy, E: # events, T: # transitions.

Model	Control States				E	T	Other characteristics
	Total	Basic	Conc	D			
Farmer puzzle	1	1	0	1	0	2	-
EHealth	1	1	0	1	0	6	Environmental variables, state invariants.
Musical chairs	5	4	0	1	2	4	-
Elevator	1	1	0	1	0	7	Environmental variables, named actions.
NASA FGS	63	33	16	6	2	43	Many variables (53 in/out), named conditions, state invariants.
Snapshot UI	8	5	0	3	7	7	-
Traffic light	9	6	2	3	2	6	-
Java Thread Lifecycle	9	8	0	1	19	19	Transition comprehension, transition templates
Bit counter	7	4	2	3	3	4	Generated events

Table 3: SLOC comparison of Dash and Alloy. The citation indicates the source of the original Alloy models.

Model	Hand-crafted Alloy	Dash	Generated Alloy
Farmer puzzle	27 [44]	41	100
Ehealth	-	70	209
Musical Chairs	116 [15]	51	160
Elevator	-	114	257
NASA FGS	-	723	5246
Snapshot UI	32 [51]	49	205
Traffic Light	-	44	431
Java Thread Lifecycle	-	120	577
Bit Counter	-	37	349

hierarchy and the big steps that concurrency creates. Given the popularity of UML statemachines and this modelling paradigm, we think Dash provides a natural transition for these modellers into abstract formal representations of data operations. Dash enhances Alloy with the ability to model transition systems that include control state hierarchy and events. Thus, providing structure to Alloy behavioural models. Dash can be used to model systems all across the spectrum, ranging from data-intensive models to highly hierarchical and control-oriented models.

Evaluating Model Checking Performance of Dash Models. To demonstrate the feasibility of model checking models written in Dash, we use a constrained version of transitive-closure-based model checking (TCMC) to check CTL properties on the Alloy models resulting from our translation, al-

Table 4: Model checking performance of case studies.The average time of three runs is reported in milliseconds. SS is the significant scope. A cross (✕) means the scope is not appropriate for the model. **Entries** are properties that fail.

Model	Property	SS	Snapshot Scope 7	8	9	11
Farmer puzzle	No quantum objects	2	13.3	23.7	33.3	49.3
	Solve puzzle[a]		**4.3**	9.0	11.0	13.7
EHealth	Operation	6	12.0	6.3	6.0	7.3
Musical Chairs	Always more players than chairs	8,11[b]	✕	3.7	✕	17.3
	Alice wins the game		✕	3.7	✕	9.0
	Players sit during the game		✕	2.0	✕	6.0
	Game eventually finishes		✕	2.7	✕	12.7
Elevator	Called floor eventually reached	7	246.7	614.3	2423.7	35572.0
	Always one current floor		12.3	38.7	57.7	121.3
	Eventual maintenance		26.0	68.3	213.0	733.7
NASA FGS	At most one lateral mode active[c]	9	✕	✕	**1691.3**	**9797.3**
	At least one lateral mode active		✕	✕	11745.3	26326.0
	AP engaged implies modes On		✕	✕	183037.7	21197871.0
	Onside FD on implies modes On[d]		✕	✕	**6581.3**	**12940.7**
	Offside FD on implies modes On		✕	✕	17503.7	28923.7
	AP engaged turns FD On		✕	✕	23033.7	65708.3
	ROLL Selected iff ROLL active		✕	✕	616.0	2003.7
Snapshot UI	Answers through students	8	2.7	7.7	13.3	42.3
	Logs out and logins back		2.3	3.0	4.7	6.7
Traffic Light	Both lights not green	7	6.0	6.3	✕	✕
Bit Counter	Model is responsive	7	4.7	5.0	✕	✕
	Final bit status		4.7	6.7	✕	✕

[a] Property fails because the minimum number of moves to solve the puzzle is 8.
[b] We checked two variations of the model, one with 2 chairs, and one with 3 chairs. Conclusion of the game can be reached in a model with 8 and 11 snapshots, respectively.
[c] Property fails because we did not constrain the event processing.
[d] Property fails because there is no fixed order of execution.

though any model checking method in Alloy can be used for checking properties of Dash models. TCMC supports checking loops and infinite paths (although the path goes through a finite set of states because Alloy only checks finite models). By supporting infinite paths, we can check liveness properties. TCMC returns entire models as counterexamples. Since we want to view only counterexample paths, we add a path constraint to TCMC to require that from every snapshot there is at most one successor snapshot (similar to [26]), and require that the last snapshot of a trace must be stable (to produce complete big steps). Since it is usually not possible for Alloy to analyze the complete reachable state space of a model, we use a method called significance axioms [16] to determine scopes for the models in which either every transition or every control state is reachable (called the significant scope). Our model checking results for some of the models are summarised in Table 4. The analysis was executed running Alloy 5.1.0 on an Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz x 8 machine running Linux version 4.4.0-137-generic with up to 64GB of user-space memory. An ✕ entry in the table means that either the scope was below the significant scope for the model or the scope of the snapshot set does not result in paths consisting of complete big steps. If the

user selects a scope for which the model constraints cannot hold, then Alloy fails to produce an instance so it is important to ensure that the model itself has instances at a scope prior to checking properties. In a Dash model, the observable points are at the big step boundaries, so the CTL properties are written to only check the points when the snapshot is stable. For example, a property that would be typically written as $AG(p \Rightarrow EX\ q)$ is written as $AG(stable\ \wedge\ p \Rightarrow EX(\neg stable\ EU(stable\ \wedge\ q)))$.

While the Alloy Analyzer cannot check the entire reachable state space, these analysis times show that it is possible to get useful model checking results for Dash models in a reasonable amount of time. Properties are valid or fail while investigating a model that has an instance of every transition or every control state is potentially reachable.

Evaluating and Understanding Counterexamples. Through careful design of the structure of the snapshots signature in Alloy and by taking advantage of the Alloy Analyzer's support for themes, we facilitate the interpretation of model checking results in terms of the original Dash models. In our translation, the snapshot signature acts as a package for a model's variables and some context information. We have configured a theme for the Analyzer that displays snapshot atoms as rectangles, and other signature elements as attributes. The only relation displayed as an arc is `small_step`, which relates snapshots and highlights the steps. Figure 6 shows an instance of the bit counter. The representation clearly shows the transitions taken during a step, and the value of variables in each snapshot, which facilitates the understanding of a model.

7 Related Work

Languages that Combine Abstract Data and Control. TCOZ [30] is a language that combines Object-Z to describe data and its operations with Timed CSP for the formalization of real time constraints, concurrency, and synchronization. Although the language does not directly support analysis and verification of models, some transformations have been developed to reuse existing tools (*e.g.,* [11] where a specification is projected into Timed Automata). Circus [52] is another integrated language that combines Z and CSP, and ongoing work is being done on the development of a model checker [33], [19]. Similarly, CSP-CASL [42] blends CSP with CASL, a language that allows modular and hierarchical specifications. The aforementioned languages use process algebras to describe control-oriented behaviour. The semantics of a process algebra can usually be described in a compositional manner. On the other hand, Dash is based on Statecharts (widely known and used in UML statemachines [50]). The use of Statecharts in Dash presents different challenges for stating its semantics because global communication allows a transition in one state to enable or disable a transition in another state, so it is difficult to state its semantics in a compositional manner and instead global context is needed within the snapshot. The semantics of Dash use the notion of big and small steps, which allow the system to react to environmental input in a con-

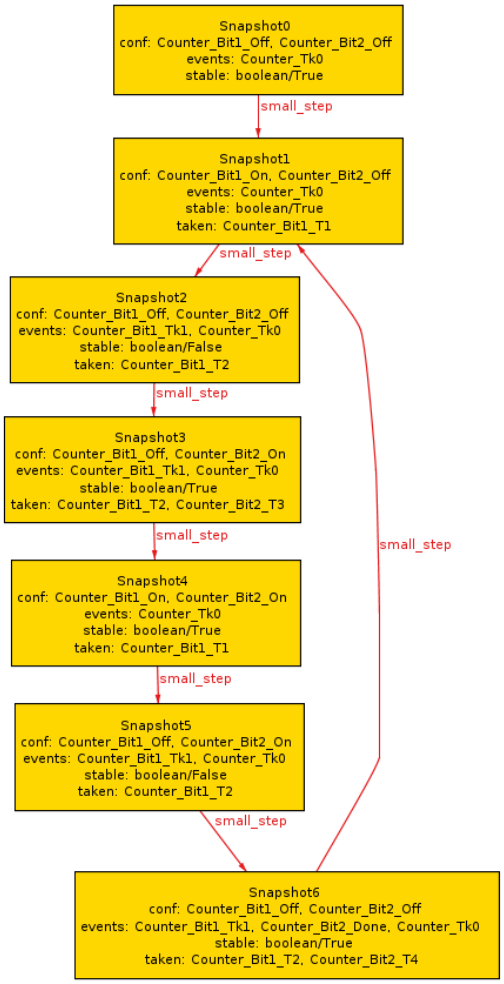

Fig. 6: Visualisation of a Dash model instance.

current and causal manner, demarcating specific observable moments (*i.e.*, stable snapshots). CASL-Charts [39] integrates Statecharts with the algebraic specification language CASL. Data operations are axiomatized and transition triggers are in CASL, but transition actions are still a sequence of assignments or event triggers as in Statecharts (rather than declarative constraints). Its semantics are defined as a combination of the languages rather than a mapping to CASL. In OZS [20], Statecharts are combined with Object-Z. The actions of a transition are described using a Z schema and the semantics of the language are given by a mapping to Object-Z. In Dash, Statecharts is novelly coupled with Alloy, a popular language for the specification of complex structural systems, in a seamless manner: Dash extends the Alloy language so both pre- and

post-conditions of transitions are described in Alloy. Via our semantics and translator, we have demonstrated how the Alloy Analyzer can be used for the verification of temporal properties of Dash models.

Alloy and its Extensions. Chang and Jackson [6] describe a translation from Alloy to SMV for model checking, but no new modelling constructs are introduced. Electrum [28] is a language that extends Alloy with temporal operators. Expressions are described using the Alloy language and linear temporal logic (LTL) operators. Primed variables denote the value of a snapshot element in the next state. Electrum uses LTL to describe both the model and the properties to check in model checking. To describe the meaning of Dash models in Electrum would require all the Alloy specification provided in our translation. The `small_step` relation we create would be used with the "always" temporal operator in Electrum to create the model. Then, the Electrum features that model check LTL properties could be used on our Dash models. In summary, Dash is non-overlapping and compatible with Electrum.

DynAlloy [17, 18, 40] extends Alloy with imperative-programming-like constructs (atomic actions composed using sequential composition, iteration, and non-deterministic choice). In DynAlloy, the properties to prove are integrated into the model description as pre/post conditions[14] as in the Floyd-Hoare approach to partial program correctness. The elements of the snapshot are determined implicitly in that they are passed to actions as parameters. DynAlloy does not have labelled control states. The state hierarchy of Dash models would have to be flattened and encoded as changes to variables in DynAlloy. The complications of big-steps, small-steps, and events would also have to be explicitly encoded in DynAlloy. Thus the abstractions of hierarchical control states for model decomposition that Dash provides for writing transition systems would be lost. The goals of Dash and DynAlloy are distinct. DynAlloy is aimed at describing abstract models of program behaviour integrated with properties to be proven using proof techniques based on partial program correctness. Dash provides the abstractions of Statecharts for describing control-oriented behaviour of abstract transition systems and the properties to be proven are described separately from the model as is common in typical model checking.

Statecharts Family. The Statecharts family of languages usually have a fixed condition and action language that does not allow for declarative specification of user-declared datatypes and operations. OCL [37] is a formal language for expressing invariants, pre- and post- conditions, which can be added onto parts of a UML model (described in a context). In contrast, Dash permits the use of Alloy formulas directly in transition conditions and actions, and has a fully formal semantics.

Declarative Modelling Languages. Declarative behavioural modelling languages (such as Z [47], VDM [24], B [1], ASMs [5], TLA$^+$ [54], SAL [3][34]) describe basic transition systems through the use of unprimed and primed snapshot variables. Control state and hierarchy can be encoded in variables

[14] The pre/post conditions of transitions in Dash and Statecharts constrain the model's behaviours.

(*e.g.,* [45]). However, none of these languages explicitly support the representation of control state hierarchy.

8 Conclusion

We have described the syntax and semantics of Dash, a novel behavioural modelling language that allows a modeller to use the common control-oriented modelling paradigm of hierarchical and concurrent control states together with declarative descriptions of data and its operations in Alloy. The hierarchy and concurrency of control states can express sequencing, priority, and concurrency of transitions, providing structure to Alloy models of transition systems. Using our semantics, we translate Dash to Alloy for analysis taking advantage of features of the Alloy language. Our key insight in creating the semantics of Dash is to match the semantics of Statecharts for the control-oriented changes of a model and match the declarative perspective for the data-oriented changes of that model. Through case studies, we have evaluated our translation and model checking of Dash models in the Alloy Analyzer. The conclusion from our case studies is that via our translation it is possible to check interesting properties of models that combine data and control abstractions in Dash. In future work, we plan to investigate how to optimize the translation and model checking analysis in order to examine models of larger scope.

Acknowledgements

We thank Ali Abbassi, Amin Bandali, Sabria Farheen, and Tamjid Hossain for their help in discussions regarding Dash and Alloy. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Abrial JR (1996) The B Book: Assigning Programs to Meanings. Cambridge University Press
2. von der Beeck M (1994) A comparison of statecharts variants. In: Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer, Lecture Notes In Computer Science, vol 863, pp 128–148
3. Bensalem S, Ganesh V, Lakhnech Y, et al (2000) An overview of SAL. In: Proceedings of the 5th NASA Langley Formal Methods Workshop
4. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y (2003) Bounded model checking. In: Advances in Computers, vol 58 Supplement C, Elsevier, pp 117 – 148
5. Börger E, Stärk R (2012) Abstract state machines: a method for high-level system design and analysis. Springer

6. Chang FSH, Jackson D (2006) Symbolic Model Checking of Declarative Relational Models. In: International Conference on Software Engineering, pp 312–320
7. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NuSMV: A New Symbolic Model Verifier. In: Proceedings of the 11th International Conference on Computer Aided Verification, pp 495–499
8. Clarke EM, Grunberg O, Peled DA (1999) Model Checking. MIT Press
9. Cofer D, Miller SP (2014) Formal methods case studies for DO-333. Tech. Rep. NASA/CR2014-218244, NASA Langley Research Center
10. Cunha A (2014) Bounded model checking of temporal formulas with Alloy. In: International Conference on Abstract State Machines, Alloy, B, VDM, and Z, Springer Berlin Heidelberg, pp 303–308
11. Dong JS, Hao P, Qin SC, Sun J, Yi W (2004) Timed patterns: TCOZ to timed automata. In: International Conference on Formal Engineering Methods, Springer, pp 483–498
12. Elrad T, Aldawud O, Bader A (2002) Aspect-oriented modeling: Bridging the gap between implementation and design. In: Generative Programming and Component Engineering, Springer, pp 189–201
13. Esmaeilsabzali S, Day NA, Atlee JM, Niu J (2010) Deconstructing the semantics of big-step modelling languages. Requirements Engineering Journal 15(2):235–265
14. Fakhroutdinov K (n.d.) Java 6 thread states and life cycle. `https://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html`, [Online; accessed 28-May-2021]
15. Farheen S (2018) Improvements to transitive-closure-based model checking in Alloy. MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science
16. Farheen S, Day NA, Vakili A, Abbassi A (2020) Transitive-closure-based model checking (TCMC) in Alloy. Journal of Software and Systems Modelling 19:721–740
17. Frias MF, Galeotti JP, López Pombo CG, Aguirre NM (2005) DynAlloy: Upgrading Alloy with actions. In: International Conference on Software Engineering, ACM, pp 442–451
18. Frias MF, López Pombo CG, Baum GA, Aguirre NM, Maibaum TSE (2005) Reasoning about static and dynamic properties in Alloy. ACM Transactions on Software Engineering Methodology 14(4):478–526
19. Gomes AO, Butterfield A (2019) Circus2csp: A tool for model-checking circus using FDR. In: International Symposium on Formal Methods, Springer, pp 235–242
20. Gruer JP, Hilaire V, Koukam A, Rovarini P (2004) Heterogeneous formal specification based on object-Z and statecharts: semantics and verification. Journal of Systems and Software 70(1-2):95–105
21. Harel D (1987) Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3):231–274
22. Jackson D (2012) Software Abstractions, 2nd edn. MIT Press

23. Jackson D (2019) Alloy: a language and tool for exploring software designs. Communications of the ACM 62(9):66–76
24. Jones CB (1990) Systematic Software Development Using VDM, 2nd edn. Prentice-Hall, Inc.
25. Kang E, Adepu S, Jackson D, Mathur AP (2016) Model-based security analysis of a water treatment system. In: IEEE/ACM 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), IEEE, pp 22–28
26. Kember M, Tran L, Gao G, Day NA (2019) Extracting counterexamples from transitive-closure-based model checking. In: Modelling in Software Engineering (MISE), a workshop of the International Conference on Software Engineering, ACM, pp 47–54
27. Lu Y, Atlee JM, Day NA, Niu J (2004) Mapping template semantics to SMV. In: Automated Software Engineering, IEEE Computer Society, pp 320–325
28. Macedo N, Brunel J, Chemouil D, Cunha A, Kuperberg D (2016) Lightweight specification and analysis of dynamic systems with rich configurations. In: Foundations of Software Engineering, ACM, pp 373–383
29. Maggiolo-Schettini A, Peron A, Tini S (2003) A comparison of statecharts step semantics. Theoretical Computer Science 290(1):465–498
30. Mahony B, Dong JS (1998) Blending object-Z and timed CSP: an introduction to TCOZ. In: International Conference on Software Engineering, IEEE, pp 95–104
31. McCarthy J, Hayes PJ (1969) Some philosophical problems from the standpoint of artificial intelligence. In: Machine Intelligence 4, Edinburgh University Press, pp 463–502
32. McMillan KL (1992) The SMV system. http://www.kenmcmil.com/language.ps
33. Mota A, Farias A, Didier A, Woodcock J (2014) Rapid prototyping of a semantically well founded circus model checker. In: International Conference on Software Engineering and Formal Methods, Springer, pp 235–249
34. de Moura L, Owre S, Rueß H, Rushby J, Shankar N, Sorea M, Tiwari A (2004) SAL 2. In: Computer-Aided Verification, Springer, pp 496–500
35. Nissanke N (1999) Formal Specification: Techniques and Applications. Springer Verlag
36. Niu J, Atlee JM, Day NA (2003) Template semantics for model-based notations. IEEE Transactions on Software Engineering 29(10):866–882
37. OCL (2014) Object constraint language. `http://www.omg.org/spec/OCL/2.4/PDF`, [Online; accessed 28-May-2021]
38. Ostroff JS (2017) Validating software via abstract state specifications. Tech. Rep. EECS-2017-02, York University
39. Reggio G, Repetto L (2000) CASL-CHART: a combination of statecharts and of the algebraic specification language CASL. In: International Conference on Algebraic Methodology and Software Technology, Springer, pp 243–257

40. Regis G, Cornejo C, Gutiérrez Brida S, Politano M, Raverta F, Ponzio P, Aguirre N, Galeotti JP, Frias M (2017) DynAlloy analyzer: A tool for the specification and analysis of Alloy models with dynamic behaviour. In: Foundations of Software Engineering, ACM, pp 969–973
41. Reynolds MC (2013) Modeling the java bytecode verifier. Science of Computer Programming 78(3):327–342
42. Roggenbach M (2006) CSP-CASL—a new integration of process algebra and algebraic specification. Theoretical Computer Science 354(1):42–71
43. Schmidt D (2006) Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer 39(2):25–31
44. Seater R, Dennis G (n.d.) Tutorial for Alloy Analyzer 4.0. `http://alloytools.org/tutorials/online/index.html`, [Online; accessed 28-May-2021]
45. Sekerinski E (1998) Graphical design of reactive systems. In: International B Conference, Springer, pp 182–197
46. Serna J, Day NA, Farheen S (2017) Dash: A new language for declarative behavioural requirements with control state hierarchy. In: IEEE International Requirements Engineering Conference Workshops (REW), IEEE, pp 64–68
47. Spivey JM (1992) The Z Notation: A reference manual, 2nd edn. International Series in Computer Science, Prentice Hall
48. Svendsen A, Møller-Pedersen B, Haugen Ø, Endresen J, Carlson E (2010) Formalizing train control language: automating analysis of train stations. In: Comprail, pp 245–256
49. Torlak E, Jackson D (2007) Kodkod: A relational model finder. In: Tools and Algorithms for the Construction and Analysis of Systems, pp 632–647
50. UML (2017) Unified modeling language. `https://www.omg.org/spec/UML/2.5.1/PDF`, [Online; accessed 28-May-2021]
51. Wayne H (2018) Formally specifying UIs. `https://hillelwayne.com/post/formally-specifying-uis/`, [Online; accessed 28-May-2021]
52. Woodcock J, Cavalcanti A (2001) A concurrent language for refinement. In: Proceedings of the 5th Irish Conference on Formal Methods, BCS Learning & Development Ltd., pp 93–115
53. xtext (2021) Xtext. `https://eclipse.org/Xtext/`, [Online; accessed 28-May-2021]
54. Yu Y, Manolios P, Lamport L (1999) Model checking TLA+ specifications. In: Correct Hardware Design and Verification Methods, pp 54–66
55. Zave P (2017) Reasoning about identifier spaces: How to make chord correct. IEEE Transactions on Software Engineering 43(12):1144–1156