DASH: Declarative Modelling with Control State Hierarchy (Preliminary Version)

David R. Cheriton School of Computer Science
University of Waterloo
Technical Report 2018-04

Jose Serna, Nancy A. Day, and Shahram Esmaeilsabzali
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{jserna,nday,sesmaeil}@uwaterloo.ca

ABSTRACT

We present a new language, called DASH, for describing formal behavioural models. DASH combines common modelling constructs to describe abstractly both data and control in an integrated manner. DASH uses the Alloy language for describing data and its operations declaratively, and adds syntax for labelled control state hierarchy common in Statecharts descriptions of transition systems. In addition, DASH accommodates multiple factoring paradigms for modelling (control states, events, and conditions) and includes syntactic sugar (*e.g.,* transition comprehension, transition templates) to write models that are concise and easy to understand. We describe the formal semantics of DASH, which carefully mix the usual semantic understanding of control state hierarchy with the declarative perspective, for creating abstract models early in system development. We implement these semantics in a translator from DASH to Alloy taking advantage of Alloy language features. We demonstrate DASH, our tool, and model checking analysis in the Alloy Analyzer using several case studies. The key novel insight of our work is in combining seamlessly common data and control modelling paradigms in a way that will be intuitive for those used to either paradigm, and enabling automatic analysis of the integrated model.

1 INTRODUCTION

The goal of model-driven engineering (MDE) [33] is to reduce the complexity of the system development process through the use of models. The models used early in system development must be more abstract than descriptions in design and code, and must be analyzable to provide the modeller with feedback on a model's correctness. Several formal languages for behavioural models have been developed that are both abstract and formal in order to be analyzable using techniques from formal methods. These languages can be divided into two categories: 1) those paradigms that specify abstractly the data of a system and its operations (*e.g.,* Alloy [23, 24], Z [36], TLA+ [42]); and 2) those that decompose the system via the control-oriented modelling paradigms of concurrent and hierarchical control states and events (*e.g.,* the Statecharts family of languages [21] including UML statemachines [2]). In the data-oriented paradigm, the focus is on describing the data operations abstractly and declaratively. In the control-oriented paradigm, the focus is on prescribing the order and priority of the sequence of operations, with limited capabilities for describing rich and abstract data operations. For many systems, describing both the data and control aspects of the system in an integrated, abstract behavioural model would be beneficial, however, current languages lack the ability to do both easily. A example use case for this combination is having tables of relationships between uninterpreted data that evolve over time, *e.g.,* a mapping from bank customers to the amount of money in their bank account that changes in one step of an automated teller machine (ATM) model.

We present a new language called DASH[1], which unites common modelling constructs to describe abstractly both data and control in an integrated behavioural model. DASH uses the Alloy language as the underlying language for describing data and its operations declaratively, and adds syntax for the control state hierarchy common in Statecharts. In addition, DASH introduces syntactic sugar to improve the conciseness of models and accommodates multiple factoring paradigms for modelling (states, events, and conditions).

A behavioural model describes a transition system. The semantics of a language with control state hierarchy can quickly become complicated because of the potential for multiple transitions in a step and the need to determine which of these steps are observable to the environment of the system. From the semantic framework for these languages given in Esmaeilsabzali *et al.* [16], we have chosen a formal semantics of DASH that carefully mixes the usual semantic understanding of control state hierarchy with the declarative perspective, consistent with the goal of creating abstract models early in system development. As a choice for analysis of DASH, we chose to map all of DASH to Alloy so that no extensions would be required for analysis. 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 [17] for a comparison of modelling in Alloy vs SMV). We describe how we exploit Alloy language features to model the control state hierarchy of DASH. We show the conciseness of DASH models by comparing their size to the equivalent Alloy models resulting from our translation.

The time is right for a modelling language that combines language features for describing data and control abstractly in part because of the improvements to tool support for model checking abstract models. Bounded model checking (BMC) [7] (and its implementation in nuXmv [9] using satisfiable modulo theories (SMT) solvers [5]) and transitive-closure-based model checking (TCMC) [39, 40] are both symbolic model checking methods that support reasoning over abstract datatypes and have been implemented in the Alloy Analyzer. We use TCMC for analysis of our

[1]The name DASH comes from "Declarative Abstract State Hierarchy".

case studies. Since Alloy cannot check the entire reachable state space of most models, we take advantage of the notion of significance axioms [17], which force models to cover an interesting part of the state space for model checking analysis.

Our work can be viewed as either an extension to Alloy or an extension to Statecharts. There has been related work regarding how to model and verify transition systems abstractly in Alloy using either 1) syntactic extensions, such as Electrum [28] and DynAlloy [19, 20, 32]; or 2) style guidelines such as those found in Jackson [24] and Farheen [17]. Like DASH, these efforts all describe means of packaging transitions. DASH is the only one to provide the common control-oriented modelling paradigm of hierarchical and concurrent labelled control states as a means of organizing (sequencing, priority, concurrency) how the transitions are taken in a model. As an extension to Statecharts (or UML-based languages such as UMPLE [18]) DASH permits declarative and abstract representations of data. Other languages (*e.g.,* SAL [6, 13], ASMs [8], TLA+ [42], B [4], Chang and Jackson [10] NuXmv [9]) provide the means to model transition systems abstractly however, none of this work includes language support for modelling hierarchical control states.

We described an initial draft of DASH in [35]. We now present the language in full along with its semantics, tool support, and examples to demonstrate its use for both modelling and analysis. The contributions of our paper are:

(1) The development of a language, DASH, that combines hierarchical control states seamlessly with first-order logic data abstractions to create an integrated, formal model of a system's behaviour.
(2) A choice of semantics for DASH, that matches common meanings of both control-oriented and declarative modelling paradigms.
(3) A translation from DASH to Alloy so no tool extensions are needed for model checking analysis.
(4) Examples that demonstrate the features and analysis of DASH models.

The key novel insight of our work is in combining these two common modelling paradigms in a way that is intuitive for those used to either paradigm, and enabling automatic analysis of the integrated model. DASH examples and our tool as a web service are available at http://129.97.7.33:8080/dash/.

2 BACKGROUND

In this section, we briefly present background on the Alloy language and analyzer, hierarchical control state models, and model checking in Alloy.

Alloy is a popular language for describing models based on first-order logic, sets, and relations. Finite sizes (scopes) for each set are chosen at the time of analysis to permit finite model finding by mapping the satisfiability problem to propositional logic using a solver called kodkod [38]. In an Alloy model, a set is described using a signature. Relations between this set and others are specified in the signature of the set as in:

```
1
2  sig A {        // a set called A
3    R1: B,       // a relation from A to B
4    R2: B -> C // a relation from A to B to C
5  }
```

The type (sort) of the relations can include constraints such as `lone` and `one` to limit the multiplicity of the relations.

Alloy's type system has simple yet versatile subtyping capabilities. The elements of a signature are called atoms. Signatures that extend other signatures are called subsignatures and they declare subtypes, as in:

```
1  sig D,E extends A {}
```

All the immediate subtypes of a signature are disjoint. A signature can be declared as `abstract` meaning that the set is constrained to only contain atoms defined by subsignatures. There cannot exist an atom of the type defined by the abstract signature that is not included in the sets defined by the subsignatures.

Constraints in Alloy over the sets and relations are described in facts as in:

```
1  fact {
2    // at least one b for every k in dom(R1)
3    all k: A | some b in k.R1
4  }
```

The expression `k.R1` conveniently looks like the `R1` field of an `A`'s' record/class, but is actually using the join operator (.) to take the range of the pairs in `R1` that have `k` as their first element[2]. The association of relations directly with signatures gives Alloy modelling an object-oriented flavour, although there is no association of behavioural changes with the signature. Alloy provides abstract operations on relations and functions (such as join, union, *etc.,*). Alloy goes beyond first-order logic by including the transitive closure operator on relations (which can be computed for a finite set). The facts can be decomposed into predicates and functions that take arguments. The Alloy Analyzer produces a visual representation of a satisfying instance (values for the sets and relations) when one can be found.

A **transition system** describes a behavioural model as a set of snapshots[3] and a transition relation that prescribes the possible movements between snapshots, which are called **steps**. A **snapshot** is an encapsulation of a mapping from variables to values. The Statecharts [21] family of languages (which includes UML statemachines [2]) was developed for modelling transition systems of control-oriented, reactive systems, where the system runs continuously and responds to environment events. Figure 3 shows an example of a statecharts model. A system is control-oriented if there are moments in the system's behaviour that can be naturally named (the control states), such as a traffic light showing red, green, or yellow lights, and the transitions are relevant based on these control states. Control states provide a means of sequencing transitions in a behavioural model. Hierarchical states (OR-states) are a further means of decomposing the system's behaviour and express priority of transitions (usually outer state over inner state). Concurrent states (AND-states) permit separation of concerns in a model for components whose behaviours are mostly independent of each other. There are many variations in the semantics of how a set of transitions is chosen to be taken in a step [16, 41], but almost all

[2]Technically, Alloy has no scalars so k is a subset of A.
[3]These are typically called states, but we use snapshots to avoid the confusion with labelled control states.

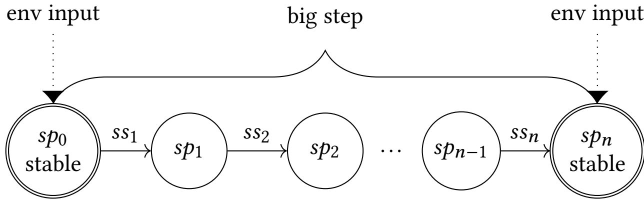

Figure 1: Big step (*sp* is a snapshot; *ss* is a small step)

variations agree on the notion of a **big step** (called macro-steps by Harel) consisting of a number of **small steps** (called micro-steps by Harel) as a way to represent the system's response to environmental stimuli as illustrated in Figure 1. Small steps are taken until the system cannot take any more, which is when it is considered stable and therefore observable. Multiple small steps exist because there are multiple concurrent states that take transitions in response to the environment (or possibly a cascading effect from other concurrent states). Additionally, Statecharts languages explicitly support named **events** as observable user actions (such as "key_swiped" or "button_pushed") that cause a response in the system. Hierarchical and concurrent control states provide two major advantages to modellers. First, the reaction of a model to an environmental input can be conveniently modelled as multiple small steps, without worrying about a new environmental input being missed during the reaction of the model to the current environmental input. And second, since the reaction of a model to an environmental input can consist of more than one transition, a model can be decomposed into orthogonal parts, each of which can take part separately in the reaction. As such, a modeller can decompose a model into parts, each of which either corresponds to a physical component of a system under study or is used to facilitate the separation of concerns in modelling.

Modelling transition systems in Alloy can be accomplished by creating a set of snapshots and constraining a binary relation over these snapshots to be the transition relation. There is a relation mapping snapshots to the values of the variables in that snapshot. A comparison of a few approaches for structuring snapshots for building a transition relation in Alloy can be found in [37] (such as wrapping in a snapshot signature or passing variables directly as arguments). Typically, parts of this relation are described separately in predicates [4] and composed using disjunction to form the transition relation [24]. However, there is no explicit language support for describing behavioural models.

A transition relation in Alloy can be iterated to do bounded model checking. Commonly, the set of snapshots is ordered (using a built-in Alloy ordering module) to provide a nice representation for traces of the behavioural model. In an alternative method for model checking in Alloy, called scoped transitive-closure-based model checking (TCMC), the meaning of all temporal operators in Computation Tree Logic with fairness constraints (CTLFC) [11] are described in terms of the transitive closure operator. While it is usually not possible to check the properties over the entire reachable snapshot space in Alloy (event for finite sets), bugs can be found and some conclusions regarding the entire reachable snapshot spaces can be concluded (such as liveness). In an effort to provide some confidence that a large enough fraction of the reachable snapshot space has been checked, we introduced significance axioms [17], which force the analysis to check parts of the snapshot space with some interesting behaviours.

[4]These are called "events" in [24], but we avoid that terminology because of the different meaning of events in control-oriented models.

```
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      // all Chairs and Players in game
12      players = Player
13      chairs = Chair
14      occupied = none -> none  // empty relation
15    }
16    default state Start { ... }  // default state
17    state Walking {
18      trans Sit {                 // transition
19        on MusicStops             // event trigger
20        goto Sitting              // dest state
21        do {                      // action
22          occupied' in chairs -> players
23          chairs' = chairs
24          players' = players
25          // occupied is a total fcn
26          all c : chairs' | one c .(occupied')
27          // occupied is injective
28          all p : Chair.(occupied') | one occupied'.p
29        }
30      }
31    }
32    state Sitting { ... }
33    state End {}
34  }
```

Figure 2: DASH model for musical chairs

3 DASH

A DASH model describes a set of possible changes to snapshots that combine to be the behaviour of the model. Users can describe snapshot variables using Alloy constructs and the set of transitions using new syntactic features introduced by DASH. Figure 2 shows part of a DASH model for the game musical chairs. The musical chairs example comes from [31] where it was modelled in Z.

DASH extends Alloy with keywords for creating hierarchical and concurrent control states and transitions. In Figure 2, on line 2 a concurrent state called `Game` is created, which means the text within these brackets describes a transition relation. Nested within `Game`, there are control states `Start` (the default state on line 16), `Walking` (line 17), `Sitting` (line 32), and `End` (line 33), which are the phases of the game where players walk around the chairs while the music is one and then when the music stops have to find a chair to sit in. This loop repeats (the transition from `Sitting` to `Start` is not shown) until there is only one player and the game goes to the state `End`. States (AND and OR) can be arbitrarily nested to represent the state

hierarchy in the model. An example of a DASH model with more concurrent states is in Figure 4 represented graphically in Figure 3 (from [15]).

A variable of a snapshot can consist of any type of value representable in Alloy. Notably, this includes uninterpreted sorts (to represent abstract data at the time of modelling) and relations and functions to represent collections of data abstractly. In Figure 2, on line 1, `Chair` and `Player` are uninterpreted sorts. Declarations within a state are variables that are part of the snapshot (*i.e.,* they change value). In musical chairs, the set of players, chairs, and the relationship between chairs and players (*i.e.,* who is sitting where) are the snapshot variables (line 3–5). The events of the music starting and stopped are declared on lines 6 and 7; these are declared to be environmental using the keyword `env`. Variables can also be declared environmental, which means the model does not control their values and their values can change non-deterministically at big step boundaries.

Initial constraints on the variables are shown on lines 8–15 of the musical chairs model. These ensure that when the game starts there is one more chair than players and no one is sitting down.

Transitions are described within a `trans` block as in:

```
1  trans tlabel {
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 were chosen to match the way a transition is described in English. An example transition is on lines 18-30 in Figure 2. Each component of the 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 [36], unprimed variables are the current values of snapshot elements and primed variables are the variable values in the next snapshot. For example, on line 26, in the Alloy formula `all c : chairs' | one c .(occupied')`, we enforce the constraint that every chair has someone sitting on it in the next snapshot. Notably, 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 is any formula in Alloy but may only refer to unprimed snapshot variables. The source state, guard condition, and the event trigger together form a pre-condition for a transition and the action, generated event, and destination state are the post-condition.

The state regions define namespaces in DASH. A reference to a variable from another state must be prefixed by its home state as on line 28 in Fig 4. While the semantics uses global communication (as in most Statecharts languages), enforcing the namespaces means that duplicate names are not an issue and the modeller is very aware of locality.

The keywords for states, transitions, and events are Core DASH and are the only necessary extensions to Alloy. DASH includes

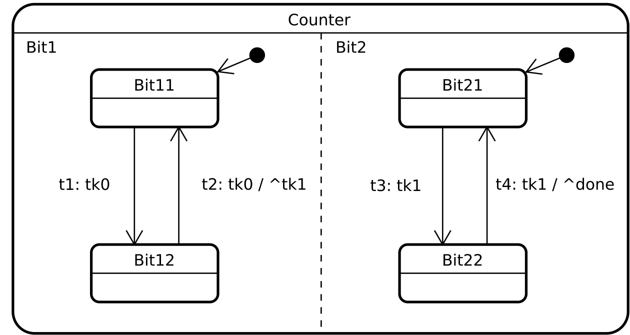

Figure 3: Two-bit counter

```
1   conc state Counter {
2     env event Tk0 {}
3
4     def trans Count[src, des: State, e: Event] {
5       from src on e goto des
6     }
7
8     conc state Bit1 {
9         event Tk1 {}
10
11        default state Bit11 { }
12        state Bit12 {}
13
14        trans T1 { Count[Bit11, Bit12, Tk0] }
15        trans T2 {
16          from Bit12 on Tk0 goto Bit11 send Tk1
17        }
18    }
19
20    conc state Bit2 {
21      event Done {}
22
23      default state Bit21 { }
24      trans T3 { count[Bit21, Bit22, Bit1/Tk1] }
25
26      state Bit22 {
27        trans T4 {
28          on Bit1/Tk1 goto Bit21 send Done
29        }
30      }
31    }
32  }
```

Figure 4: Two-bit counter model in DASH

some additional syntactic sugar for convenience that can be easily transformed into Core DASH. 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 state that goes to the `ErrorState` on an `error` event. Additionally, part of the definition of a transition can be described in a different part of a model, similar to

aspect-oriented modelling [14], by using addons that are **layered** together to get the full description of a transition. For example,

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

adds the action `incErrorCounter` to every transition whose destination is the `ErrorState`. A new feature that was not previously described in [35] is **transition templates**, which capture similarities in transitions to avoid duplication in the model. A template is a parameterized definition of a transition that can be instantiated. Line 4–6 of Figure 4 show a transition template. Uses of this template are on lines 14 and 24. Also, after recognizing the role that control states play in factoring snapshots into sets that have the same possible future behaviours, we realized that transitions can also be **factored by events and conditions**. There are models where control state are not useful and for these control states can be omitted (except for the one outermost 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. In this way, DASH accommodates multiple factoring paradigms for modelling (control states, events, and conditions).

4 SEMANTICS OF DASH

Stating the semantics of a language such as DASH is difficult because its semantics are not compositional in the structure of the model (*i.e.,* we cannot describe the meaning of each transition individually and combine these meanings to describe the meaning of the model). It is reasonable for transitions in multiple concurrent states to respond to an environmental input, thus the semantics of DASH must address the question of which transitions can be taken together in a big step as depicted in Figure 1. A big step consists of one or more small steps, each of which can be one or more transitions. The big step continues until the system of the model is stable, *i.e.,* no more transitions are enabled. 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 is true. Currently, we do not permit a stuttering big step where no transitions are taken.

We rely on the semantic framework of Esmaeilsabzali *et al.* [16], which describes a space of semantic aspects and options for these languages, to state our semantics for DASH. Our choices for each of the semantic options are described in Table 1, and are based on two reasons: 1) as a declarative model, a transition action can describe a "large" change (*i.e.,* a sequence of operations is rarely needed); and 2) ease of understanding of the model.

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 (2) above since race conditions, which could occur if multiple concurrent states place constraints on the same variable[5] are confusing to debug since they make the model inconsistent.

The big-step maximality aspect specifies the termination criteria for a sequence of small steps, *i.e.,* when the system is stable. We chose the option Take One, meaning that at most one transition

[5]While namespaces force a user to recognize when a state is referring to a variable outside of itself, it can place constraints on variables outside of itself in an action.

Semantic Option	Value 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

Table 1: Semantics of DASH

per concurrent state can be taken in a big step. For reason (2) above, we want this choice because it guarantees termination of big steps. One concurrent region can generate events that cause transitions to be enabled in another concurrent state and taken later in the big step. In an abstract model, it seems reasonable that at most one transition in each concurrent state should be allowed in a big step because of reason (1) above.

For the event lifeline aspect, we chose the option Present in Remainder of big step where a generated event can trigger transitions in the small steps after its generation until the end of a big step. For reason (2) above, we want the small steps to be causal.

For the variable lifeline[6], we chose to make the effects of 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, our variable lifeline choice cannot cause a non-terminating big step where two transitions keep enabling each other. This choice was made for reason (2) above: semantic choices that refer to the value of variables at the beginning of the big step throughout the big step are hard to follow. Because of reason (1) above, we expect the number of small steps in a big step to be reasonably small, thus there should not be many event and variable changes within a big step.

For priority, we give higher priority to transitions whose source state is a parent state over those from a child state. This choice is the most common one in Statechart languages and is easier to understand than priority based on scope (source and destination state).

Finally, we have to address the frame problem where there is a mismatch between the usual choices of declarative and control-oriented languages. In declarative languages, if a variable is not constrained in an action, it is allowed to 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. In DASH, by declaring a variable `env`, it is allowed to change when the system is stable, but otherwise retains its value. For non-environmental variables, if their primed version is mentioned in the action of the transitions, we assume the action will constrain them; if their primed version is not mentioned in the action then we enforce its value to retain its value from the previous snapshot. If the user does not like this default semantic choice, it can be overridden, by adding a constraint that a variable has a value within its range of values, thus allowing it to change non-deterministically in an action.

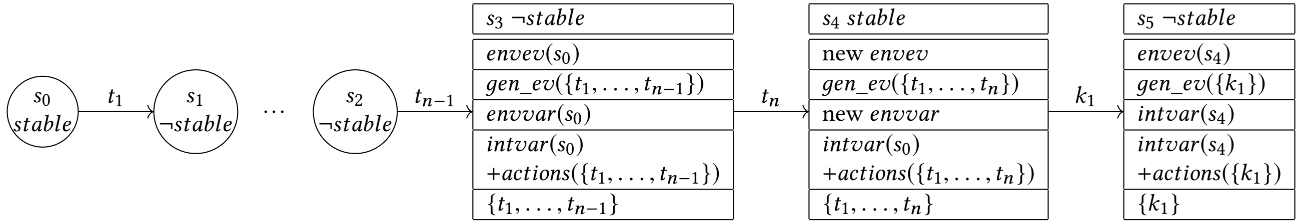

Figure 5: Snapshots in a big step (*envev* and *envvar* are environmental events and variables respectively; *gen_ev* is events generated by transitions; *actions* is the effects of the actions of transitions where + is used informally; *intvar* is internal variables and their values; "new" means a non-deterministic choice of values)

5 TRANSLATION TO ALLOY

We use the semantics we have chosen to define a translation of a DASH model to an Alloy model for formal analysis. Because we use the Alloy language within DASH for describing transition guards and actions, a DASH modeller is expected to have knowledge of the Alloy language and tools. Our goal is 1) to create a mapping that will make it as easy as possible for the user to understand counterexamples from Alloy in terms of the original DASH model; and 2) utilize features of the Alloy language as much as possible to produce a concise representation of a DASH model's behaviour.

In Alloy, the snapshots are a set of atoms with relations that link each snapshot to its variable values. The snapshot for the musical chairs model is:

```
1  sig Snapshot  {
2    Game_occupied : Chair set -> set Player,
3    Game_chairs : set Chair,
4    Game_players : set Player,
5    conf: set StateLabel,        // Control states
6    events: set EventLabel,      // Events
7    taken: set TransitionLabel, // trans taken
8    stable: one Bool
9  }
```

In addition to relations for the model's variables, it also includes values for the set of control states of the snapshot called its configuration (`conf`), the set of events (`events`), a history variable of the transitions that have been taken in the big step (`taken`), and a boolean flag to indicate if the snapshot is stable or not (`stable`).

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

```
1  abstract sig Counter extends StateLabel {}
2  abstract sig Bit1, Bit2 extends Counter {}
3  one sig Bit11, Bit12 extends Bit1 {}
4  one sig Bit21, Bit22 extends Bit2 {}
```

An abstract signature `StateLabel` is the base type for all control states. On line 1, the *AND* control state `Counter` is declared to extend `StateLabel`. *AND* and *OR* control states are also declared as abstract, since they are containers for other control states. The concurrent regions Bit_1 and Bit_2 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*; only one atom is created and used by the Alloy Analyzer and these atoms are distinct from each other.

[6]In [16], this aspect is decomposed into multiple aspects.

Subtyping directly matches the meaning of control state hierarchy. The relation `conf` contains elements of type `StateLabel` to determine the control states of the snapshot. For example, if the system is in state Bit_{11}, it is also in state Bit_1 because of the subtype hierarchy. Thus, we can check if a state is in the current snapshot without searching through its ancestors or descendants.

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

```
1  one sig Tk0 extends EnvironmentEvent {}
2  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  one sig T1, T2, T3, T4 extends TransitionLabel {}
```

The initial generic constraint on snapshots is that the system is in its default states (defined through helper functions), 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 constraints defined by modellers are added to the model. For example, the initial constraint for the musical chairs model is:

```
1  pred init[s: Snapshot] {
2    s.conf = default_State
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_players = Player
9    s.Game_chairs = Chair
10   s.Game_occupied = none -> none
11 }
```

The purpose of a DASH model is to define a next snapshot relation containing pairs that are the possible **small steps** of the system. Snapshots that are stable are the snapshots at the end/beginning of **big steps**. Figure 5 shows a sequence of snapshots where each small step contains one transition (due to the choice of single for CONCURRENCY). In Figure 5, we can see that when a snapshot is stable, it contains:

- an unconstrained set of environmental events that can trigger transitions in the next big step;

- internal events that were generated in the last big step;
- unconstrained environmental variables values that can trigger transitions in the next big step;
- internal variable values that have the accumulated effects of all transitions taken so far;
- the set of transitions taken in the last big step.

Properties can examine the snapshot at big step boundaries by checking the property only when the system is stable. We are careful to avoid creating another snapshot to reset the history variables and incorporate environmental input as is done in [27] for SMV. In SMV, there is no penalty to this reset function, but in Alloy it increases the snapshot space with the extra reset snapshots.

A small step is defined as the disjunction of predicates for each transition of the model. For example, the small step relation for the bit counter is:

```
1  pred small_step [s, s': Snapshot] {
2    Counter_Bit1_T1[s, s'] or
3    Counter_Bit1_T2[s, s'] or
4    Counter_Bit2_T3[s, s'] or
5    Counter_Bit2_T4[s, s']
6  }
```

Later, we describe how only one of these predicates can be true in a small step. For each transition, we define five predicates that work together to define the semantics of a DASH model. For ease of explanation, we show an abstract version of these five predicates in Figures 6 and 7[7].

A predicate for the pre-condition of the transition t_1 (`pre_t1`) is evaluated relative to the current snapshot. It is true if the source state of t_1 is in the snapshot's configuration and the guard of t_1 is true in the snapshot's variable values. The evaluation of the presence of t_1's trigger event 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 6); otherwise its event must be in the snapshot's set of events (line 8), which includes the environmental events generated at the beginning of the big step and the internal events generated so far in this big step. Note that in the first step of a big step, the guard 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 (`post_t1`) is evaluated relative to the current snapshot, s, and the next snapshot s'. It 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 13). Our translation produces helper functions to calculate these state changes. On line 15, the variable values for the internal values are updated according to the actions of the transition enforcing our semantic choices 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. Next, we have four cases depending on whether s is stable and whether s' will be stable. We have documented these in comments on lines 16- 45 for how the internal and environmental variables are allowed to

```
1  // for transition t1
2  pred pre_t1[s:Snapshot] {
3    src_state_t1 in s.conf
4    guard_cond_t1[s]
5    s.stable = True => {
6      trig_events_t1 in (s.events & EnvironmentEvent)
7    } else {
8      trig_events_t1 in s.events
9    }
10 }
11
12 pred pos_t1[s:Snapshot, s':Snapshot] {
13   s'.conf = s.conf - exit_src_state_t1 +
14     enter_dest_state_t1
15   act_t1[s,s']
16   testIfNextStable[s, s', gen_events_t1, t1] =>
17      s'.stable = True
18      s.stable = True => {
19        // big step = one small step
20        // only internal event is one gen by t1
21        // allow env events to change
22       no ((s'.events & InternalEvent) -
23         gen_events_t1)
24      } else {
25        // last small step of the big step
26        // add t1's gen event the internal events
27        // allow env events to change
28       no ((s'.events & InternalEvent) -
29         (gen_events_t1 + InternalEvent & s.events))
30      }
31   } else {
32     s'.stable = False
33     s.stable = True => {
34       // first small step of the big step
35       // only internal event is one gen by t1
36       s'.events & InternalEvent = gen_events_t1
37       // env events stay the same
38       s'.events & EnvironmentalEvent =
39         s.events & EnvironmentalEvent
40     } else {
41       // intermediate small step
42       // add t1's gen event to the events
43       s'.events = s.events + gen_events_t1
44     }
45     env_vars_unchanged_t1[s,s']
46   }
47 }
```

Figure 6: Transition model in Alloy Part 1

change. On line 45, environmental variables are constrained to keep their previous values when the next snapshot is not stable.

The musical chairs example illustrates that complex actions can refer to the previous and next values of snapshot variables. The constraints for the variables in the post-condition are:

```
1  s'.Game_occupied in
2    (s.Game_chairs -> s.Game_players)
3  s'.Game_chairs) = s.Game_chairs
4  s'.Game_players) = s.Game_players
5  all c : s'.Game_chairs | one c.(s'.Game_occupied)
6  all p : Chair.(s'.Game_occupied) |
7    one (s'.Game_occupied).p
```

The remaining predicates we discuss are in Figure 7. The testIfNextStable predicate (`testIfNextStable`) looks at the current snapshot, s, the next snapshot s', the transition to be taken t, and its set of

[7] In Alloy, boolean is not a built-in type and so it must be stated as `stable = True`, rather than just `stable`.

```
1  pred testIfNextStable[s,s',t,gev]  {
2    not enabledAfterStep_t1[s,s',t,gev] and
3    not enabledAfterStep_t2[s,s',t,gev] and ...
4  }
5
6  pred enabledAfterStep_t1[s, s',t, gev] {
7    src_state_t1 in s'.conf
8    trig_events_t1 in (s.events + gev)
9    guard_cond_t1[s']
10   (s.stable = True) => {
11     // only trans taken in big step is t
12     // as long as t1 is orthogonal to t
13     // then t1 is enabled in next snapshot
14     orth_t1[t]
15   } else {
16     // as long as t1 is orthogonal to t + s.taken
17     // then t1 is enabled in next snapshot
18     orth_t1[t + s.taken]
19   }
20 }
21
22 pred semantics_t1[s,s':Snapshot] {
23   (s.stable = True) => {
24     s'.taken = t1
25   } else {
26     s'.taken = s.taken + t1
27     orth_t1[s.taken]
28   }
29   !pre_t2[s]
30   !pre_t3[s]
31   ...
32 }
33
34 pred t1 [s:Snapshot, s':Snapshot] {
35   pre_t1 [s]
36   post_t1 [s,s']
37   semantics_t1[s, s']
38 }
```

Figure 7: Transition model in Alloy Part 2

generated events *gev*. The purpose of this predicate is to determine whether any transitions will be enabled in s' if t is taken so it relies on `enabledAfterStep` predicates for each transition. The constraints on lines 7 to 9 are similar to the constraints of the pre-conditions for t_1, however, here they depend on the variable values of s' and the generated events of t_1 to simulate the effects of executing t_1. The constraints on lines 10 to 19 test whether taking t will make it impossible to take t_1 in the next step because of orthogonality restrictions (only one transition per orthogonal region).

The semantics predicate for t_1 (`semantics_t1`) 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. This predicate may also include priority-related predicates when necessary. If two transitions have source states related in the hierarchy (*e.g.,* one transition's source is a 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 enforced 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.

Model	DASH LOC	Alloy LOC
Musical Chairs	76	471
Bit Counter	53	468
Traffic Light	60	640

Table 2: LOC comparison of DASH and Alloy

This last constraint ensures that only one transition is taken in a step (enforcing single semantic choice for CONCURRENCY) because if multiple transitions try to enforce this change then the model would be inconsistent in Alloy.

Predicate `t1`, on lines 34-38 of Figure 7, combines the pre, post and semantics predicates for transition t_1, meaning for t_1 to be taken, its pre-condition must be true; its post-condition must be true and its semantics constraint must hold.

Finally, anything in the DASH model that is outside of a state is copied directly to Alloy. We use some helper modules to avoid duplicating common parts of translated models.

Our translator is implemented in Xtext [3], which automatically provides editing tools. We have created a number of Alloy helper files for common definitions to avoid cluttering each model.

6 CASE STUDIES

We developed three case studies[8] to demonstrate DASH, its translation to Alloy and analysis of DASH models using TCMC. We show the sizes of the DASH models and their translation to Alloy (without the helper files) in Table 2 with respect to lines of code (LOC). The Alloy language is remarkable in its ability to capture complexity in a concise model. With DASH, we have enhanced this ability.

Since the Alloy Analyzer is designed for small scope analysis, it is rare that it can explore the entire reachable snapshot space of a model, thus we need to chose a scope of the snapshot space. In BMC, the model is viewed as a set of traces and the snapshot scope is the maximum length of a trace. TCMC, however, views the model as a transition system (potentially with loops). It explores all sub-transition systems (from the initial state) of the size of the snapshot size. All CTLFC properties can be expressed in TCMC, but their results must be interpreted relative to the snapshot scope. Thus, verification of a safety property cannot be definitive, however, a liveness property can be definitively verified. Notably, because it is checking transition systems, a liveness property that requires a loop in the model can be definitively verified. To ensure that the scope is sufficiently large to check interesting behaviours of the snapshot space (and avoid spurious instances in Alloy), we use significance axioms [17]. These axioms state that the transition system checked must have an initial snapshot; every snapshot must be reachable; and every transition in the model must be represented by at least one pair in the next snapshot relation. These axioms require that Alloy check the property for snapshot spaces that have at least one representative of every behaviour in the model. In all of our examples, Alloy was able to check models that satisfied the significance axioms.

[8]The case studies are available as sample models on our online tool http://129.97.7.33:8080/dash/.

Model	Property	Type	Snapshot Scope	Time(sec)
Musical Chairs	Always more players than Chairs	S	8	0.428
			10	6.408
			12	40.111
	Alice wins the game	E	8	0.042
			10	0.415
			12	0.325
	Players sit during the game	FL	8	0.023
			10	0.040
			12	0.099
	Game eventually finishes	IL	8	0.457
			10	6.957
			12	87.863
Bit Counter	Model is responsive	S	7	0.220
			9	1.084
			11	3.459
			13	9.277
	Final Bit Status	S	7	0.242
			9	1.160
			11	3.586
			13	13.886
Traffic Light	Both lights not green	S	7	0.089
			9	0.780
			11	4.770
			13	16.114

Table 3: Model checking performance of case studies

For all models described below, we began with some automatically generated, application-independent properties such as the reachability of basic states (EF properties). Then, we proceeded to check more interesting properties, specific to each model. For now, the properties are written directly in Alloy and refer to parts of the snapshot, however, we plan to soon provide a way to write these properties directly in DASH. Most properties are checked only when the snapshot is stable (observable).

Following [17], we categorize the properties we checked as safety (S), existential (E), finite liveness (FL), meaning it can be satisfied by a finite path, and infinite liveness (IL), meaning it is only satisfied by an infinite path. All performance results in Table 3 are from execution on a an Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz x 8 machine running Linux version 4.4.0-92-generic with up to 64GB of user-space memory. All properties are valid for the scopes that we checked. The conclusion from our case studies is that interesting properties of models that combine data and control abstractions in DASH are possible in the Alloy Analyzer using our translation.

The **musical chairs** example (originally in [31] and modelled directly in Alloy in [17]) has interesting data abstractions plus control state hierarchy (but not concurrency). The relation from people to the chairs that they sit on is concisely modelled as a relation between the abstract sets of people and chairs. The progression through the stages of the game (music playing or not) is modelled using control states. We checked some safety and liveness properties and that it is possible for a particular player (Alice) to win via an existential property in CTLFC.

We chose the **bit counter** example (adopted from [29] and [15]) to exercise the semantics of concurrent states and cascading effects between them. Our properties check that the model always responds to an environmental event, and that individual transitions complete their actions.

The **traffic light controller** (originally from [22] and previously modelled in Alloy in [39]) is a typical Statecharts example with an easy-to-understand safety property that the light cannot be green in both directions at the same time.

We acknowledge that our analysis performance is likely poor compared to hand-crafted models in NuSMV or even hand-crafted models in Alloy due to the overhead of expressing the semantics of big steps. However, we believe there is significant advantage to the ease and conciseness of expression of the models in DASH compared to these alternatives. Expressing relations in SMV is cumbersome (see [10, 17]) and the size of the sets must be known when writing the model. Hand-crafted models in Alloy cannot support concurrent and hierarchical control states without some representation of the semantics we describe.

7 RELATED WORK

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. For example, UMPLE [18] includes programs as actions. OCL [1] 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), somewhat similar to DASH's add-on construct. In contrast, DASH permits the use of FOL formulae directly in transition conditions and actions, and has a fully formal semantics. In addition, DASH offers modelling flexibility through factoring, layering, transition comprehension, and transition templates to describe a model. Although several extensions to UML (*e.g.,*[43] [26]) have been proposed to express temporal constraints, the official specification does not support this type of constraint.

Model checking tools usually have fairly primitive input languages with no support for abstract datatypes and operations. In SMV [30], transitions can be described using case/switch statements, and labelled control state hierarchy and its semantics can be encoded, but it is not supported natively. Abstract datatypes and operations can be translated to these languages as in Chang and Jackson [10]. The nuXmv [9] tool supports multiple model checking algorithms for infinite state systems, but its input language supports limited types of data and operations (*e.g.,* integers, reals).

Declarative behavioural modelling languages (such as Z [36], VDM [25], B [4], ASMs [8], TLA+ [42], SAL [6], [13]) often use unprimed and primed snapshot variables and some support packaging mechanisms (*e.g.,* schemas in Z) to describe transitions. Control state and hierarchy can be encoded in variables (*e.g.,* [34]). However, none of these languages explicitly support the representation of control state hierarchy or other methods of factoring, which are included in DASH. Previously we created a way to represent

labelled control state hierarchy as a datatype in SMT-LIB [12], but no user-level language was presented or semantics.

DynAlloy [19, 20, 32] is an extension to Alloy to describe behavioural models. DynAlloy's approach is based on transitions as programs, and the Floyd-Hoare approach to program correctness. Atomic actions are described by pre- and post- conditions and these can be composed sequentially, non-deterministically, or iteratively using DynAlloy operators. The elements of the snapshot are determined implicitly in that they are passed to actions as parameters. Snapshot elements that are not modified retain their values. Analysis is done via (sometimes optimized) translation to the Alloy Analyzer and extensions.

Electrum [28] extends Alloy with actions that have pre- and post-conditions; declares variables of the snapshot (*i.e.,* variables that can change) with the keyword "var"; and uses primed variables to refer to next snapshot values. Users can combine the actions into a transition relation using Alloy operators. Additionally, Electrum includes keywords to indicate which variables are modified in an action addressing the frame problem, and extends Alloy with keywords for LTL temporal operators and links directly to kodkod (rather than Alloy) to do BMC and translates to NuXMV to do complete model checking.

Compared to these extensions to Alloy, DASH explicitly supports the common modelling paradigm of hierarchical and concurrent control states and events to compose snapshot changes described as transitions. In particular, DASH's support for model decomposition accomplished by concurrency is not easily captured in DynAlloy or Electrum. We use a designation of a variable as environmental to guide the default behaviour for whether a variable retains its previous value or not in a step (rather than explicitly labelling variables as modified as in Electrum), which matches the idea of reactive systems as describing the system interaction with its environment. For analysis, through scoped TCMC, we provide support for scoped CTLFC temporal logic model checking, without relying on extensions to the Alloy Analyzer.

8 CONCLUSION

We have presented DASH, a novel behavioural modelling language that allows a user to create models using the common control-oriented modelling paradigm of hierarchical and concurrent control states together with declarative descriptions of data and its operations. DASH permits declarative specification of rich data operations, such as relations that evolve over time. DASH allows control states to serve their purpose in sequencing transitions. The hierarchy and concurrency of control states can express priority and independence of transitions. The ability to decompose the model into concurrent components, somewhat like object-oriented modelling is a key distinguishing feature of DASH. Through a small syntactic extension to Alloy, and careful decisions regarding its semantics, a DASH model is a fully formal, integrated model of abstract behaviour. Through a set of examples, we have shown how DASH provides a modeller with the ability to capture and analyze a complex concept in a concise model. Through more case studies, we plan to explore more analysis options (such as simulation) and optimizations. We are exploring consistency and completeness of DASH models.

REFERENCES

[1] 2014. OMG Object Constraint Specification (OCL) specification. http://www.omg.org/spec/OCL/2.4/PDF. (2014). [Online; accessed 07-June-2017].
[2] 2015. OMG Unified Modeling Language. http://www.omg.org/spec/UML/2.5/PDF/. (2015). [Online; accessed 05-June-2017].
[3] 2017. Xtext. https://eclipse.org/Xtext/. (2017). https://eclipse.org/Xtext/ [Online; accessed 05-June-2017].
[4] Jean-Raymond Abrial. 1996. *The B Book: Assigning Programs to Meanings.* Cambridge University Press.
[5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. 2009. *Satisfiability Modulo Theories.* Frontiers in Artificial Intelligence and Applications, Vol. 185. Chapter 26, 825–885.
[6] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Munoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saıdi, Natarajan Shankar, et al. 2000. An overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop.*
[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded Model Checking. Advances in Computers, Vol. 58. Elsevier, 117 – 148.
[8] Egon Börger and Robert Stärk. 2012. *Abstract state machines: a method for high-level system design and analysis.* Springer.
[9] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV (LNCS)*, Vol. 8559. Springer, 334–342.
[10] Felix Sheng-Ho Chang and Daniel Jackson. 2006. Symbolic Model Checking of Declarative Relational Models. In *ICSE.* 312–320.
[11] Edmund M. Clarke, Orna Grunberg, and Doron A. Peled. 1999. *Model Checking.* MIT Press.
[12] Nancy A. Day and Amirhossein Vakili. 2016. Representing Hierarchical State Machine Models in SMT-LIB. In *MISE @ ICSE.* ACM, 67–73.
[13] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. 2004. SAL 2. In *CAV.* Springer, 496–500.
[14] Tzilla Elrad, Omar Aldawud, and Atef Bader. 2002. Aspect-oriented modeling: Bridging the gap between implementation and design. In *Generative Programming and Component Engineering.* Springer, 189–201.
[15] Shahram Esmaeilsabzali. 2011. *Prescriptive Semantics for Big-Step Modelling Languages.* Ph.D. Dissertation. Univ. of Waterloo, Cheriton School of Comp. Sci.
[16] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. 2010. Deconstructing the semantics of big-step modelling languages. *REJ* 15, 2 (2010), 235–265.
[17] Sabria Farheen. 2018. *Improvements to Transitive-Closure-based Model Checking in Alloy.* MMath thesis. Univ. of Waterloo, Cheriton School of Comp. Sci.
[18] A. Forward, T. C. Lethbridge, and D. Brestovansky. 2009. Improving program comprehension by enhancing program constructs: An analysis of the Umple language. In *2009 IEEE 17th International Conference on Program Comprehension.* 311–312.
[19] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. 2005. DynAlloy: Upgrading Alloy with Actions. In *ICSE.* ACM, 442–451.
[20] Marcelo F. Frias, Carlos G. López Pombo, Gabriel A. Baum, Nazareno M. Aguirre, and Thomas S. E. Maibaum. 2005. Reasoning about static and dynamic properties in Alloy. *ACM. Trans. on Software Engineering* 14, 4 (2005), 478–526.
[21] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274.
[22] i Logix Inc. 1991. Statemate 4.0 Analyzer User and Reference Manual. (1991).
[23] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM TOSEM* 11, 2 (2002), 256–290.
[24] Daniel Jackson. 2012. *Software Abstractions* (2nd ed.). MIT Press.
[25] Cliff B. Jones. 1990. *Systematic Software Development Using VDM (2nd Ed.).* Prentice-Hall, Inc.
[26] Bilal Kanso and Safouan Taha. 2013. Temporal Constraint Support for OCL. In *Software Language Engineering.* Springer, 83–103.
[27] Yun Lu, Joanne M. Atlee, Nancy A. Day, and Jianwei Niu. 2004. Mapping Template Semantics to SMV. In *Automated Software Engineering (ASE).* IEEE Comp. Soc., 320–325.
[28] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. 2016. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *FSE.* ACM, 373–383.
[29] Florence Maraninchi and Yann Rémond. 2001. Argos: an Automaton-Based Synchronous Language. *Computer Languages* 27 (2001), 61–92.
[30] K. L. McMillan. 1992. The SMV system. http://www.kenmcmil.com/language.ps. (Nov. 06 1992).
[31] Nimal Nissanke. 1999. *Formal Specification: Techniques and Applications.* Springer.
[32] Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo Frias. 2017. DynAlloy Analyzer: A Tool for the Specification and Analysis of Alloy Models with Dynamic Behaviour. In *FSE.* ACM, New York, NY, USA, 969–973.

[33] D.C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer* 39, 2 (2006), 25–31.

[34] Emil Sekerinski. 1998. Graphical design of reactive systems. In *International B Conference*. Springer, 182–197.

[35] Jose Serna, Nancy A. Day, and Sabria Farheen. 2017. DASH: A New Language for Declarative Behavioural Requirements with Control State Hierarchy. In *International Workshop on Model-Driven Requirements Engineering (MoDRE) @ RE*. IEEE Computer Society, 64–68.

[36] John Michael Spivey. 1992. *The Z Notation: A reference manual*. Prentice Hall.

[37] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2017. Evaluating State Modeling Techniques in Alloy. In *SQAMIA 2017 - Proceedings of the 6th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*. 11–13.

[38] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*. 632–647.

[39] Amirhossein Vakili. 2016. *Temporal Logic Model Checking as Automated Theorem Proving*. Ph.D. Dissertation. Univ. of Waterloo, Cheriton School of Comp. Sci.

[40] Amirhossein Vakili and Nancy A. Day. 2012. Temporal Logic Model Checking in Alloy. In *ABZ (LNCS)*, Vol. 7316. Springer, 150–163.

[41] Michael von der Beeck. 1994. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (LNCS)*, Vol. 863. Springer, 128–148.

[42] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *CHARME*. 54–66.

[43] Paul Ziemann and Martin Gogolla. 2003. OCL Extended with Temporal Logic. In *Perspectives of System Informatics: 5th International Andrei Ershov Memorial Conference*. Springer, 351–357.