Dash: Declarative Behavioural Modelling in Alloy

by

Jose Serna

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

Abstract

An abstract model is a representation of the fundamental characteristics and properties of a system, and its purpose is to provide feedback to stakeholders about the correctness of the system during the early stages of development. This thesis presents Dash, a new language for the formal specification of abstract behavioural models, which combines the control-oriented constructs of statecharts with the declarative modelling of Alloy. From statecharts, Dash inherits a means to specify *hierarchy*, *concurrency*, and *communication*, three useful aspects to describe the behaviour of reactive systems. From Alloy, Dash uses the expressiveness of relational logic and set theory to abstractly and declaratively describe structures, data, and operations.

The purpose of a Dash model is to formally describe a transition system, and for this reason transitions are first-class constructs of the language. Dash provides features such as *factoring*, *transition comprehension*, and *layering*, to systematically declare and organise the transitions of a model.

The integration between statecharts and Alloy is done in Dash at the semantic level. The semantics of Dash use the notion of *big steps* and *small steps* to formally describe changes in a system, and address the mismatch between declarative and control-oriented formalisms regarding the frame problem.

This thesis presents several case studies to demonstrate the modelling capabilities and automated analysis of Dash models. The case studies range from heavily data-oriented systems to highly hierarchical and concurrent systems. Behaviours can be specified using a temporal logic and the Alloy Analyzer is used for performing analyses. We extended the notion of *significance axioms* and *significant scopes* to concurrent Dash models, to avoid spurious instances of a model and ensure that a big enough search space is explored by the Analyzer to check for interesting behaviours and provide useful feedback about a model.

Acknowledgements

I would like to thank my supervisor, Prof. Nancy A. Day, for her constant encouragement and support throughout my Master's programme. I particularly enjoyed our many discussions on the board and thanks to her continuous lessons while playing many different roles (at times she was a teacher, a colleague, a leader, a mentor, etc.), I have learned and grown a lot not only academically and professionally, but also at a personal level.

I thank Prof. Joanne M. Atlee and Prof. Daniel M. Berry who read my thesis and provided insightful comments to improve the presentation of the ideas. I thank Dr. Shahram Esmaeilsabzali who joined on many of the discussions and helped in the development of some of the topics of this thesis.

Thanks to my colleagues Sabria Farheen, Ali Abbassi, and Amin Bandali, with whom I engaged in fruitful conversations and collaborations.

I especially thank my family for their unconditional love and outstanding support, their many sacrifices have made it possible for me to focus on achieving my goals. Finally, I am grateful to my uncle Gabriel who always nurtured and encouraged my intellectual curiosity.

dedicado a má y pá

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of model-driven engineering (MDE) [44] is to reduce the complexity of system development through the use of models that are more abstract than descriptions in design and code. An abstract model is a representation of the fundamental characteristics and properties of a system, which to a certain extent, alleviates the effects of partial information and unknowns in the early stages of system development. Abstract models shield system developers from the complexity of implementation details, allowing them to focus on the core concepts and functionality of the system under development. These abstract models are often specified declaratively, *i.e.,* using constraints to specify *what* the system under consideration is intended to do, instead of defining a procedural or algorithmic specification that details *how* the system operates. Typically, the analysis of a system is decomposed into different views, each of which is approached from different perspectives. For example, a structural model describes the elements and the relationships among them in a system, whereas a behavioural model captures the changes and evolution of a system as it executes over time. Regardless of the nature of an abstract model, whether structural or behavioural, its purpose is the same: to provide feedback to stakeholders about the correctness of a system during early stages of development. For this reason, models should be specified in an unambiguous manner and amenable to automatic analysis.

Several formal languages for behavioural models have been developed that are both abstract and formal in order to be analysable using techniques from formal methods. Existing formal behavioural languages can be characterised along the following spectrum. On one end, there are formal behavioural languages that are data-oriented; they focus on the abstract description of data and its operations (*e.g.,* Alloy [29, 30], TLA+ [54], VDM [31], Z [48],). On the other end, there are languages that are control-oriented; they focus on prescribing the order and priority of the sequence of operations through concurrency and synchronisation, *e.g.,* the statecharts family of languages [25] including UML statemachines [2], and process algebras such as CSP [26]. A few languages exist that combine both data-oriented and control-oriented modelling, *e.g.,* TCOZ [35] and Circus [52], which combine Z [48] and process algebra; CASL-Chart [41] and OZS [24] which combine statecharts with CASL, and Object-Z respectively; CSP-CASL [43] which combines CSP with CASL, an algebraic language. However, these existing languages have one or more of the following problems: 1) lack of expressiveness for specification of complex behaviours, 2) intimidating notations, 3) lack of automated reasoning and analysis, 4) no formal or incongruent semantics.

Formalisms such as B, TLA$^+$, VDM, Z, use primed variables to represent state change, however, they lack support for the specification of more complicated behaviours, leaving modellers with the task of developing ad-hoc procedures and/or relying on guidelines and conventions to describe behavioural aspects such as concurrency and synchronisation.

On the other hand, some languages that have support for the specification of control-oriented behaviour (*e.g.,* statecharts [25], UML statemachines [2], Statemate [28], Stateflow [5]), despite sharing similar syntax, lack the definition of a fully formal semantics or have incompatible semantic variations. This lack limits collaboration because a specification might be interpreted differently by two modellers.

Furthermore, some of these languages (*e.g.,* VDM, Z) use a strong mathematical notation, which might intimidate some modellers, or tool support has been developed around refinement (*e.g.,* TCOZ, CSP-CASL) with generation of proof obligations that need theorem proving for verification (which is time-consuming and requires expertise). Both, a heavy mathematical notation and the need of theorem proving, hinder the widespread adoption of these specification techniques because of a steep learning curve and the need of highly trained users, which is not cost-effective for most applications (except for the development of critical systems).

This thesis presents Dash [1], a new language for the formal specification of abstract behavioural models, which combines the control-oriented constructs of statecharts with the declarative modelling of Alloy. From statecharts, Dash inherits a means to specify *hierarchy*, *concurrency* and *communication*, three useful aspects to describe the behaviour of reactive systems. From Alloy, Dash uses the expressiveness of relational logic and set theory to abstractly and declaratively describe structures, data and operations. The semantics of Dash seamlessly combine the constructs of Alloy with statecharts.

Dash directly addresses the challenge areas, identified by Woodcock *et al.* [53], that call for further research and development to promote the widespread adoption of formal

[1]The name Dash comes from "**D**eclarative **a**bstract **s**tate **h**ierarchy"

methods: lightweight formalisms, robust tool support, increased automation, and cost-effectiveness. Dash is a lightweight formalism because it is a language tailored for the specification of abstract behavioural models of requirements. The purpose of Dash is to allow the identification of inconsistencies in the early stages of system development, contrary to heavyweight formal methods that strive to provide correctness all the way in the development life cycle to the point of implementation. Regarding tool support, Dash uses the Alloy Analyzer for verification of models. The Alloy Analyzer is a robust tool that has been used to uncover insidious bugs on protocols that have been claimed to be proven correct (*e.g.,* [55]). On top of the Alloy Analyzer, we developed a tool with advanced editing capabilities to create and edit Dash models. Regarding automation, verification of Dash models is based on model checking, an automated technique that exhaustively explores the possible states of a system to verify a temporal specification. Lastly, Dash uses Alloy which continues to become more prominent, alongside statecharts features that are widely used in industry (*e.g.,* UML statemachines); this means that the learning curve for Dash is not as steep as alternative languages improving the cost-effectiveness.

> **Thesis statement**. Dash is a new language intended for formally describing the behaviour of systems. Dash permits the specification of behavioural models in an *abstract* and *declarative* manner. A wide range of systems can be described using Dash, from heavily *data* and *operation-oriented*, to highly *hierarchical* and *concurrent* systems. The semantics of hierarchical control state constructs can be seamlessly combined with first-order logic abstractions to create integrated formal specifications. Behavioural properties can be specified in Dash and then checked using the Alloy Analyzer. The notion of *significance axioms* and *significant scopes* can be extended for concurrent and hierarchical models, and help in the visualisation and understanding of model checking results.

The contributions of this thesis are

- a description of the syntax of Dash, a new language for formally describing abstract declarative behavioural models,
- a description of the features of Dash for conveniently describing transition systems, and accommodating different modelling paradigms,
- a definition of the semantics of Dash which seamlessly integrate hierarchical control constructs with declarative modelling,

- a tool that translates Dash models to Alloy,
- case studies that show the modelling capabilities of Dash across the spectrum of systems, and the verification of behavioural properties,
- an extension of the notion of significance axioms for hierarchical and concurrent models, and
- guidelines to use Alloy's support of *themes* for better visualisation of instances of a model.

The fundamentals of the syntax of Dash have been published [47], along with a discussion of some of the characteristics of Dash [8]. A presentation of the semantics of Dash is available as a technical report [46].

1.1 Thesis Outline

Chapter 2 provides background on statecharts and its semantics, fundamental elements of the Alloy language and model checking in Alloy. Chapter 3 presents the syntax and features of Dash to create models, and how to specify temporal properties in Dash. Chapter 4 provides an overview of the semantics of Dash. Chapter 5 explains how a Dash model is transformed to Core Dash, and then formalised in Alloy by describing a transition relation. Additionally, the translator tool is presented with an explanation of the available translation options. Chapter 6 presents some case studies that demonstrate the modelling capabilities and model checking of Dash models, and explains how to use themes in Alloy to improve the visualisation of an instance of a model. Chapter 7 compares Dash with closely related work, covering issues related to syntax, semantics and tool support. Finally, Chapter 8 concludes and points to future directions of research and development to extend and improve Dash.

Chapter 2

Background

This chapter presents a brief background on statecharts and transition systems, the Alloy language and Analyzer, and model checking in Alloy.

2.1 Statecharts

Statecharts [25] is a graphical language for describing the behaviour of reactive systems. A reactive system is usually event-driven, thus, it runs continuously to react to environmental input and internal changes. For example, a graphical user interface (GUI) is a reactive system that continuously responds to commands and input given by a user through a keyboard or a mouse. The behaviour of a reactive system is usually specified by a *transition system*, which consists of a set of snapshots[1] and a transition relation that prescribes the possible movements between snapshots. A snapshot represents a moment in the execution of a system; it is a mapping from the variables of the system and context to their values. The movement between two consecutive snapshots of a transition system is called a *step*. Statecharts provides a formalism with three key features to describe the behaviours of reactive systems: *hierarchy*, *concurrency* and *communication*. Figure 2.1 shows an example of the statecharts representation of a two-bit counter(the model is taken from [19]).

Labelled control states correspond to moments in the behaviour of a system that can naturally be named. There are three types of labelled control states: AND, OR and BASIC control states. AND and OR control states are containers for other states. A basic

[1]The common term used in the literature is state, but we use snapshot to avoid confusion with labelled control states

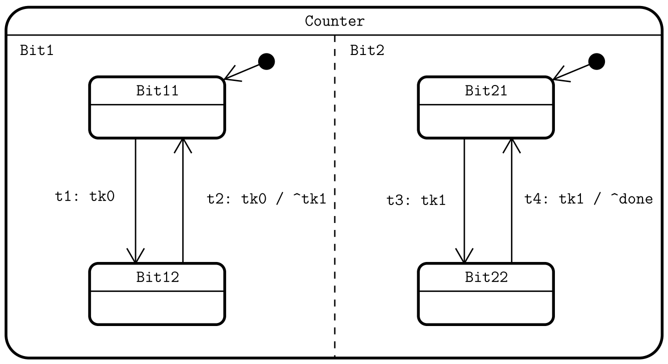


Figure 2.1: Statecharts model of a two-bit counter

labelled control state is one that has no children. It is possible to nest labelled control states and form a state *hierarchy* that defines different levels of abstraction. For example, in Figure 2.1 state `Bit1` is an abstraction of states `Bit11` and `Bit12` that captures a common property of `Bit11` and `Bit12`, namely, that they both correspond to the values of the least significant bit of the counter. In an OR-control state only one of its children is active at any moment, and upon initialisation of the system, the active state is denoted by an incoming arrow pointing to that state. The other type of container state, the AND-state, describes *concurrency*. In statecharts, concurrent regions are identified by dashed lines and the meaning is that if an AND-state is active, then the system is simultaneously active in each of the concurrent regions. For the bit-counter it means that at all moments, one substate of `Bit1` and one substate of `Bit2` are active. 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.

Transitions define how a system moves from one control state to another. Figure 2.2 shows the general syntax of a transition in statecharts, where `A` corresponds to the source state and `B` to the target state. The transition itself is indicated by an arrow from the source state to the target state. The pre-conditions of the transition are specified to the left of the '/' symbol. `e` is the enabling event of the transition, it could be an event generated by the environment or internally by the system. Inside square brackets ('[ ]') the guard condition `c` is specified. Post-conditions of the transition appear to the right side of the '/' symbol. `a` indicates the actions of the transition and any generated event `g` is specified using the ‘^’ symbol . A transition is taken (*i.e.,* the system moves from the source to the destination control state) if the enabling event occurs and the guard condition evaluates to true. Two transitions are *orthogonal*, if they are contained in different concurrent regions. For example, in the bit counter transition `t1` is orthogonal to transitions `t3` and `t4`.

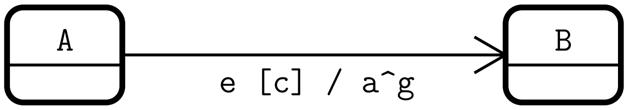


Figure 2.2: Transition in statecharts

In statecharts, events are used as a mechanism for *broadcast communication*. When a transition is taken an event may be generated and then sensed in another part of the model, producing cascading effects. In the bit counter (Figure 2.1) the two orthogonal regions communicate by means of the event `tk1`. When transition `t2` is taken, the event `tk1` is generated, which may enable transition `t3` in the other concurrent region.

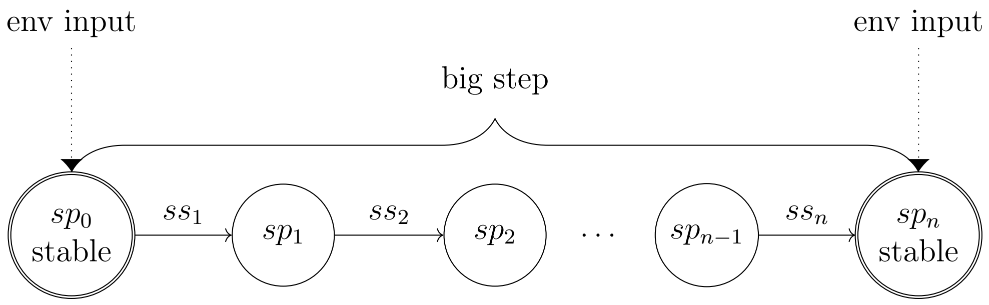


Figure 2.3: Big step (*sp* is a snapshot; *ss* is a small step)

Since AND-states allow several concurrent regions to be active simultaneously, different transitions might be enabled at the same time for a given input and be taken during the same step of a transition system. There are many variations in the semantics of how a set of transitions is chosen to be taken in a step [51, 20], but almost all 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 input as illustrated in Figure 2.3. Small steps are taken until the system cannot take any more transitions, which is when it is considered stable and therefore observable. Multiple small steps exist because there are several concurrent states that take transitions in response to the environment (or possibly a cascading effect from other concurrent states). Esmaeilsabzali *et al.* [20], provide a framework which describes a space of semantic aspects and options for languages that use big steps.

2.2 Alloy

Alloy [29, 30] is a popular modelling language suitable for describing and exploring structures. The language is based on first-order relational logic and set theory. In Alloy, everything constitutes a relation; there is no native notion of sets or scalars. A set is a unary relation, and a scalar is a singleton, unary relation. An Alloy model is a description of a set of relations coupled with some constraints on these relations. Elements of relations are called atoms and a set of atoms is defined using a type signature. Relations are declared as fields in the body of signatures.

```
sig A {}                   // a set called A
abstract sig B {}          // an abstract signature called B
sig C extends B {}
sig D extends B {          // a subsignature of B called D
  R1: C,                   // a relation from D to C
  R2: C -> one A           // a relation from D to C to A
}
```

The type of a field declaration can include constraints such as `lone`, `one`, and `set` to limit the multiplicity of the relations. The keyword `lone` means that a relation contains zero or one element, `one` means that the relation is a singleton (has exactly one element), and `set` declares the relation to have any number of elements (possibly none). If no multiplicity is stated the default is `one`. An Alloy signature that extends another signature is called a subsignature and it declares a subtype. All the immediate subtypes of a signature are disjoint. A signature can be declared as `abstract` meaning that the set exclusively contains atoms that are in the subsignatures.

Constraints in Alloy can be defined as an optional block in a signature declaration or packaged in a predicate, a function or a fact. The first case, known as a signature fact, constraints the elements of a signature. For example, in the following signature declaration `F` is a signature fact.

```
sig S {
  ... // field declarations
} {
  F   // signature fact
}
```

A predicate is a template for a constraint that can be instantiated in different contexts. a predicate is declared using the keyword `pred` and can be parameterised with a list of arguments. Functions are templates for expressions, they are declared using the keyword `fun` and, like predicates, functions also accept a list of arguments. Both predicates and functions must have a name.

```
pred predicate[a: A, b: B] {
  // predicate body
}

fun function[a: A]: B { // a function with type A -> B
  // function body
}
```

Facts are constraints that always hold, they represent assumptions. Facts are introduced using the keyword `fact` and can be anonymous or may be given a name for documentation purposes.

```
fact {
  // at least one C for every D in R1
  all d: D | some c:C | c in d.R1
}
```

The expression `d.R1` conveniently looks like the `R1` field of `D`'s record/class, but is actually using the join operator (`.`) to take the range of the pairs in `R1` that have `d` as their first element. 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). All expressions in Alloy evaluate either to a boolean value (these expressions are referred to as *constraints* or *formulas*) or result in a relation.

Analysis in Alloy is done by mapping a model, using the Kodkod [50] finite model finder, to propositional logic and performing constraint solving to find an *instance* that satisfies the constraints. There are two types of analysis and associated commands supported by Alloy. A `run` command instructs the Alloy Analyzer to search for an *example* that satisfies a model. A `check` command searches for a *counterexample* that shows that an assertion does not hold. Assertions are blocks of redundant constraints that are intended to hold and are checked to ensure consistency of a model. Scopes are indicated when invoking a command to put bounds on the sets of a model. The Alloy Analyzer produces a visual representation of a satisfying instance (values for the sets and relations) when one can be found.

2.3 Model checking in Alloy

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 [49]. Typically, parts of this relation are described separately in predicates [2] and composed using disjunction to form the transition relation [30]. However, there is no explicit language support in Alloy for describing behavioural models.

A transition relation in Alloy can be iterated to do bounded model checking [11] (BMC), which is a technique that uses symbolic model checking to verify temporal properties on the paths of a transition system up to a certain length. Commonly, the set of snapshots is ordered (using a built-in Alloy ordering module) to provide a nice representation for traces of a 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) [15] 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 (even for finite sets), bugs can be found and some conclusions regarding the entire reachable snapshot space 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, significance axioms [21] can be used to instruct the Alloy Analyzer to check parts of the snapshot space with interesting behaviours. The significance axioms are:

- *Reachability axiom*: This axiom ensures that all the snapshots considered during analysis must be reachable from an initial snapshot, and that at least one such initial snapshot exists.
- *Operations axiom*: This axiom states that every transition defined in a model is represented by a pair of snapshots in the transition relation.

These axioms are satisfiable only if the snapshot space is big enough. The minimum scope in Alloy necessary to satisfy the axioms and to obtain a significant instance of a model is called a *significant scope*.

[2]These are called "events" in *Software Abstractions* [30], but we avoid that terminology because of the different meaning of events in control-oriented models.

2.4 Summary

Statecharts is a graphical language for describing the behaviour of reactive systems. Statecharts provides a formalism for the specification of *hierarchy*, *concurrency* and *communication*; three key aspects to describe the behaviour of reactive systems. The semantics of statecharts usually have the notion of *big steps* and *small steps* to define the set of transitions taken when a system reacts to the environment or internal changes. Alloy is a language suitable for modelling and exploring structures. The Alloy language is based on relational logic and set theory, and supports multiple abstract operations. An alloy model consists of a set of relations coupled with constraints, and analysis is done by mapping the model to propositional logic and performing constraint solving. Model checking in Alloy can be done by creating a set of snapshots and defining a binary relation over these snapshots to be the transition relation. Bounded model checking (BMC) and transitive-closure-based model checking (TCMC) are two techniques to do model checking in Alloy. *Significance axioms* and *significant scopes* are used to ensure that the Alloy Analyzer covers an interesting portion of the snapshot space during model checking.

Chapter 3

Dash Syntax and Features

Dash is a language for the specification of behavioural models that are formally described by a transition system. For this reason, transitions are first-class constructs in Dash. Many of the syntactic constructs and features of Dash overlap and offer multiple ways to specify and systematically organise transitions, giving users the ability to write models using different paradigms or modelling styles. This chapter presents a description of the syntax and features to write transitions and temporal properties, and concludes with the grammar of Dash.

3.1 Examples

Dash syntax and features are explained with the use of two running examples: the game musical chairs and a two-bit counter. Excerpts of the Dash representation of the models are used throughout the following sections to help in the explanation. The complete Dash specifications of musical chairs and of the bit counter are available in Appendix B and Appendix C, respectively.

3.1.1 Musical Chairs

Musical chairs is a well known children's game. The model was originally presented by Nissanke and formalised in Z. An informal specification of the game is given next.

> "The game starts off with a collection of chairs and a collection of players. There is always exactly one player more than the number of chairs. In every

round of the game, the players first dance to some music played by a third party. In the middle of their dance, the music is terminated abruptly with no prior notice. The players must immediately occupy a chair. Before the next round, the person who happens to be without a chair is eliminated from the game, and so is one of the chairs. The winner is the person who managed to survive until the last round and to occupy the only remaining chair. Obviously, there are other rules which are taken for granted, for example, that any chair can accommodate only one person, that nobody is allowed to be seated while the music is being played and that the game is finitary (*i.e.,*. it does not go on forever)." [38]

3.1.2 Bit Counter

The second running example is the two-bit counter introduced in section 2.1. The model is taken from Esmaeilsabzali's PhD thesis, and the following is a description of the system.

> "Control states `Bit1` and `Bit2` model the least and most significant bits of the counter, respectively. Each time the environmental input event `tk0`, which represents a clock tick, is received, the counter increments by one. After an even number of ticks, `Bit1` sends event `tk1`, thereby instructs `Bit2` to toggle its status. After counting four clock ticks, the counter generates the `done` event." [19]

3.2 Alloy Constructs

Dash extends Alloy to include dedicated syntax for the specification of states and behaviour. Most of the syntactic constructs available in Alloy [1] can be directly used in a Dash model. For example, sets are declared using signatures, data abstractions are modelled as relations, and constraints can be packaged in predicates and facts. Section 2.2 describes some of the fundamental Alloy syntax to create an Alloy model.

[1]The current version of Dash is based on Alloy 4, however, features such as macros and meta-capabilities are not recognised by the Dash parser (They can still be included in a model as part of a escape block, see section 3.4.7)

3.3 Core Dash

The minimal set of constructs necessary to describe a behavioural model in Dash is called Core Dash. Core Dash is composed of the description of a state hierarchy, a set of transitions, initial constraints, and invariants. The semantics of the language are based on this minimal set as explained in Chapter 4. The following sections describe the constructs of Core Dash.

3.3.1 States and State Hierarchy

A labelled control state is a named moment in the execution of a model. A control states is an abstraction that groups past behaviours that have common future behaviours. In Dash the key word `state` is used to declared such abstractions. A states is declared by providing a name and, optionally, a modifier to identify the type of state. A concurrent (AND-) states is declared using the modifier `conc`. A default states is declared using the modifier `default`. State declarations can be nested to represent a state hierarchy (OR-states, which do not need a modifier keyword for their declaration). At the same level of the hierarchy all states must be of the same type. Additionally, a top-level state must be declared as concurrent. The reason is that for modularity and compositionality, we envision that future versions of Dash will support larger specifications to be defined over multiple files, each containing the description of a different state machine. From a semantic perspective, multiple files contain models that run in parallel. Figure 3.1 shows the state hierarchy of the bit counter in Dash.

```
conc state Counter {         // top-level state
  conc state Bit1 {          // concurrent state
    default state Bit11 {}
    state Bit12 {}
  }

  conc state Bit2 {
    default state Bit21 {}  // default state
    state Bit22 {}
  }
}
```

Figure 3.1: State hierarchy in Dash

3.3.2 Namespaces

A state region defines a namespaces in Dash. A reference to an element from another state must be given by its fully qualified name. A qualified name is formed by following the state hierarchy separating state names with '/' and then adding the element name. While the semantics of Dash uses global communication (as in most statecharts languages), enforcing the namespaces means that duplicate names are not an issue and modellers are aware of locality.

3.3.3 State Variables

A state variable is an abstractions of the information of a system. A variable is declared as an Alloy relation inside the block of a Dash state. Furthermore, variable declarations can be preceded by the keyword `env` meaning that the value of a variable is not controlled by the system being modelled, but by the environment. In other other words, an environmental variable is an input of a model. Figure 3.2 shows the variable declarations of the musical chairs model.

```
sig Chair, Player {}                    // set declarations

conc state Game {
  // Game variables
  active_players: set Player            // the players of the game
  active_chairs: set Chair              // the chairs of the game
  occupied: Chair set -> set Player     // who is sitting where
  ...
}
```

Figure 3.2: State variables declaration in Dash

3.3.4 Transitions

A transitions indicates a change in a system. A transition is declared using the keyword `trans`, and a name may be provided for documentation purposes and/or as handle to refer to the transition. The general schema to declare a transition is the following:

```
trans name {
  from source_state
  on trigger_event
  when guard_condition
  goto destination_state
  do actions
  send generated_events
}

```

Figure 3.3: General schema of a transition declaration

Where `source_state` is a state name that indicates the source state of a transition. The `trigger_event` is the name of an event that triggers a transition. A comma-separated list of event names is also supported, in which case all events on the list must occur simultaneously to trigger the transition. The `guard_condition` is an Alloy expression that relates a model state's variables; when the expression evaluates to true a transition is enabled. The `destination_state` is the name of a transition's destination. The `actions` component is an Alloy expression that constrains primed version of a state's variables to model the effects of executing a transition. The `generated_event` is the name of an event or a comma-separated list of events that are generated when taking a transition to model broadcast communication and cascading effects.

All components of a transition declaration are optional and their values are given by the transition's context (see section 3.4.3). If the `from` or `goto` part of a transition declaration is omitted, the container state where the transition is declared is assumed to be the source or destination state of the transition, respectively. Both parts can be omitted to model looping transitions. The latter case is particularly useful for modelling systems without state hierarchy. Figure 3.4 shows the declaration of some transitions of the game musical chairs.

```
state Walking {
  trans Sit {
    on MusicStops
    goto Sitting
    do  {
      occupied' in active_chairs -> active_players
      active_chairs' = active_chairs
      active_players' = active_players
      // forcing occupied to be total and
      // each chair mapped to only one player
      all c : active_chairs' | one c .(occupied')
      // each " occupying " player is sitting on one chair
      all p : Chair.(occupied') | one occupied'. p
    }
  }
}
```

Figure 3.4: Transition declarations in Dash

3.3.5 Events

An events is a signal of the occurrence of something noteworthy in a system. An event is declared using the keyword `event` and must be given a name. The modifier `env` may precede an event declaration to designate an environmental event. Figure 3.5 shows some event declarations of the bit counter.

```
conc state Counter {
  env event Tk0 {}    // declaration of environmental event

  conc state Bit1 {
    event Tk1 {}      // declaration of internal event
    ...
  }
  ...
  conc state Bit2 {
    event Done {}     // declaration of internal event
    ...
    trans T4 {
        from Bit22
        on Bit1/Tk1   // event is referenced using its qualified name
        goto Bit21
        send Done     // generation of internal event
    }
  }
}
```

Figure 3.5: Event declarations in Dash

3.3.6 Initial Constraints

An initial constraint describes what should be true of a system upon initialisation to put the system in a known state. An initial constraint is declared using the keyword `init` and any expression written in Alloy can be included in the body. A name may be given to an `init` block for documentation purposes. Figure 3.6 shows the initial constraints of musical chairs.

```
init {
  #active_players > 1
  // there is exactly one player more than chairs
  #active_players = (#active_chairs).plus[1]
  // force all Chair and Player to be included
  active_players = Player
  active_chairs = Chair
  occupied = none -> none     // empty relation
 }
```

Figure 3.6: Initial constraints in Dash

3.3.7 State Invariants

A state invariant describes an assumption about a state of a system. A state invariant is local and the assumption should hold when the state an invariant is declared in is active. If one wants to describe an invariant that should always hold for a system, the invariant must be declared inside the top-most state block. A state invariant is declared using the keyword `invariant` and any expression written in Alloy can be included in the body. However, primed versions of state variables cannot be present on an a state invariant expression. A name may be given to an `invariant` block for documentation purposes. Figure 3.7 shows an example of invariant declarations.

```
conc state Root {
  invariant Global {     // assumption should always hold
    ...
  }

  state S1 {
    invariant Local {    // assumption should hold when S1 is active
      ...
    }
  }
  default state S2 {...}
}
```

Figure 3.7: Invariants in Dash

In musical chairs an invariant that states that when the game is in the `Walking` state, no one is occupying a chair can be expressed as:

```
conc state Game {
  active_players: set Player
  active_chairs: set Chair
  occupied: Chair set -> set Player

  default state Start {...}
  state Walking {
    invariant {     // local invariant
      no occupied   // no chair is occupied while players are walking
    }
    ...
  }
  state Sitting {...}
  state End {}
  ...
}
```

Figure 3.8: State invariant of musical chairs

3.4 Additional Syntactic Features

Dash offers several syntactic constructs to ease the description of the behaviour of a system. These constructs are meant to help in code reuse, systematic organisation and facilitate writing a model in different manners. The following sections describe these additional features of Dash.

3.4.1 Named Conditions

A conditions is a boolean expression that, when evaluates to true, enables a transition. A named condition is declared using the keyword `condition` and must be given a name. A named condition in Dash is a template for any Alloy expression that can be used in the declaration of different transitions. The expression is written inside square brackets after the name of a condition. Figure 3.9 shows an example of named conditions.

```
condition ReusableCondition [
  ...     // expression written in Alloy
] {}

trans T1 {
  from S1
  when ReusableCondition
  goto S2
}

trans T2 {
  from S2
  when !ReusableCondition
  goto S1
}
```

Figure 3.9: Named condition in Dash

3.4.2 Named Actions

An action is a change in a system as the effect of taking a transition. State change is represented in Dash by the use of primed expressions that stand for the value of variables in the next moment. A named action is declared using the keyword `action` and must be given a name. A named action in Dash is a template for any Alloy expression that can be used in the declaration of different transitions. The expression is written inside square brackets after the name of an action. Figure 3.10 shows an example of named actions.

```
action ReusableAction [
  ...     // expression written in Alloy
] {}

trans T1 {
  from S1
  on E1
  goto S2
  do ReusableAction
}

trans T2 {
  from S2
  on E1
  goto S1
  do ReusableAction
}
```

Figure 3.10: Named action in Dash

3.4.3 Factoring

A labelled control state is an abstraction that groups common past and future behaviours of a system; it is a way to factor transitions of a model. Factoring in Dash refers to grouping together transitions based on a common element. Since Dash is a text-based language, factoring of transitions can go beyond state hierarchy: transitions can also be factored by conditions and events. These factoring constructs can be nested 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).

Factoring by Events

When a transition is declared in the body of an event or in one of its nested elements, the container event is added to the list of event triggers of the transition.

```
event E1 {
  trans T1 {       // T1 is triggerd by E1
    goto S2        // source state is S1 and destination is S2
  }

  event E2 {       // nested event declaration
    event E3 {}
    trans T2 {     // T2 is triggered by E1 and E2
      goto S3
      send E3
    }
  }
}
```

Figure 3.11: Factoring by events in Dash

Factoring by Condition

When a transition is declared in the body of a condition or one of its nested elements, the container condition is conjuncted to the guard condition of the transition.

```
condition C1[...] {
  trans T1 {        // T1 is enabled when C1 is true
      goto S2       // source state is S1 and destination is S2
  }

  // factoring by condition and by event
  condition C2[...] {    // nested condition declaration
    event E1 {
      // T2 is enabled when C1 and C2 are true and E1 occurs
      trans T2 {
        goto S3
      }
    }
  }
}
```

Figure 3.12: Factoring by condition in Dash

3.4.4 Transition Comprehension

Transition comprehension is used to declare several transitions with a single statement. It extends the capability of transition declaration (see Figure 3.3) to support a list of state names, or a wildcard $*$, for the source and destination state parts. In the first case, a new transition with identical definition (*i.e.,* trigger event, guard condition, action, and generated events) is created for every name provided in the list. If a wildcard $*$ is provided, an identical transition is created for every state under the current scope (*i.e.,* for every child state of the state element that contains the transition comprehension declaration). Transitions that are created from a transition comprehension expression are given a unique names as described in section 5.2. Figure 3.13 shows an example of a telephone system, which models the behaviour of a user hanging up the phone at any moment. The statements from line 3 to line 13, can be replaced by the transition comprehension on line 17.

```
1  // without transition comprehension
2  event HangUp {}
3  trans T1 {
4    from Calling on HangUp goto Idle
5  }
6
7  trans T2 {
8    from Talking on HangUp goto Idle
9  }
10
11 trans T3 {
12   from Busy on HangUp goto Idle
13 }
14 ...
15
16 // alternative using transition comprehension
17 trans ToIdle {
18   from * on HangUp goto Idle
19 }
```

Figure 3.13: Transition comprehension in Dash

3.4.5 Layering

Layering allows modellers to describe *addon* parts of a transition in different places in a model, which are then combined together to create the complete description of the transition. Addons facilitate aspect-oriented modelling which is based on Separation of Concerns (SoC), and describes the modularisation of aspects (*i.e.,* cross cutting concerns like logging, security, etc) that are then combined with the main functionality [18]. An addon is declared using the keyword `addon` and a name may be provided for documentation purposes. Figure 3.14 shows the general template for declaring addons.

```
addon name (addon_effect) to target_transitions

addon_effect := do action
| send event_names
target_transitions := trans_names
| from (* | state_names) to (* | state_names)
| *

```

Figure 3.14: Template for declaring addons in Dash

`addon_effect` represents the purpose of the addon. It can be either the specification of an action or the generation of some events. In the first case, the effect is specified as a `do action`, where `action` is an Alloy expression or the identifier of a named action. In the latter case, the effect is specified as `send event_names`, where `event_names` is a name or a comma-separated list of the name of the events to be generated.

`target_transitions` is the transitions to which the addon will apply. There are three different ways to define the target transitions. First, the transitions can be explicitly identified with a comma-separated list of transition names. Another options is to use a wildcard *, which means that an addon will be added to all of the transitions found in the current scope (*i.e.,* all transitions declared inside the state that contains an addon declaration). The third option provides a mid-range control; transitions can be selected based on their source and/or destination state. Figure 3.15 shows some examples of addon declarations.

```
action LogError[...]{}
action IncrementCounter[...]{}
event ErrorNotification {}

// addon is specifed for a explicit list of transitions
addon LogErrors (do LogError) to T1, T2, T3, T4

// send notification for every transition that goes to Error state
addon RaiseError (send ErrorNotification) to (from * to Error)

// increment internal counter for every transition taken
addon Increment (do IncrementCounter) to *
```

Figure 3.15: Addon declarations in Dash

3.4.6 Transition Templates

A transition template allows code reuse of transitions that have a common structure. A transition template is defined similarly to a normal transition, however, the keyword `def` precedes the declaration, a name must be given and a list of parameters. To instantiate a transition from a template, in the body of a transition declaration the name of the template is included along with the actual parameters of the transition. Figure 3.16 shows an example of using transition templates.

```
event E1 {}
event E2 {}

def trans template[src: State, e: Event] {    // defines a template
  from src                                    // formal parameter
  on e                                        // formal parameter
  goto DestinationState
  do Action
}

trans FromS1 {           // instantiates a transition from a template
  template[S1, E1]       // S1 and E1 are the actual paremeters
}

trans FromS2 {
  template[S2, E2]
}
```

Figure 3.16: Transition templates in Dash

3.4.7 Escape Blocks

Sometimes it is desirable to write expressions that refer directly to the Alloy code generated from a Dash model (for example, to write model checking properties that are based on BMC), and to do that Dash provides *escape* blocks. As the name suggests, the content of an escape block is copied verbatim from a Dash model to the generated Alloy model. Another possible use of escape blocks is to preserve comments and documentation of a model. Escaped content is included within an opening tag `{escape}` and a closing tag `{/escape}`. Figure 3.17 shows the definition of a temporal property that cannot be described directly in Dash (the current version does not support referring to the transitions that have been taken, because that property refers to a value introduced in the semantics).

```
{escape}
/** checks that the counter always reacts to a TK0 event */
assert model_responsive {
  all s: Snapshot | s.stable = True and Counter_Tk0 in s.events =>
    some s': s.*nextStep | s.stable = True and
      (Counter_Bit1_T1 in s'.taken or
      Counter_Bit1_T2 in s'.taken or
      Counter_Bit2_T3 in s'.taken or
      Counter_Bit2_T4 in s'.taken)
}
{/escape}
```

Figure 3.17: Escaped blocks in Dash

3.5 Temporal Properties

Temporal property specification in Dash is done using CTL logic. A CTL formula can be specified in the body of an assertion (*i.e.,* to instruct the Analyzer to find a counterexample) or in the body of a predicate (*i.e.,* to try and find an example that satisfies a model). Figure 3.18 shows the list of supported CTL operators.

ex	ef	eg	eu
ax	af	ag	au

Figure 3.18: CTL operators in Dash

Temporal operators can be nested to specify complex properties. Figure 3.19 shows the specification of some temporal properties of musical chairs.

```
assert ctl_safety {
  // number of active_players is always 1 greater than number of
    active_chairs
  ag (#Game/active_players = (#Game/active_chairs).plus[1])

}

one sig Alice extends Player {}
pred ctl_existential {
  // Alice wins in some instance
  // the expression ‘state_name in conf’
  // is used to test if a state is active
  ef (Game/End in conf and Game/active_players = Alice)
}
```

Figure 3.19: Temporal specification in Dash

3.6 Grammar

An abbreviated version of the Dash grammar that showcases the syntactic additions to the Alloy language is shown in Figure 3.20 (the complete grammar definition, including Alloy constructs, is available on Appendix A).The grammar is structured in a way that facilitates its presentation. The actual implementation in Xtext has a different structure; in part to take advantage of some features of Xtext, and also because Xtext uses ANTLR [3] (a parser that implements an LL(*) algorithm), which does not support left recursive grammars. Common BNF operators are used to define the grammar and some other conventions are followed as indicated.

- a^* means zero or more repetition of a
- a^+ means at least one or more repetitions of a
- $a \mid b$ means choice between a and b
- $[a]$ means that a is optional
- $a,^*$ means zero or more repetitions of a separated by comma
- $a,^+$ means at least one or more repetitions of a separated by comma

- ***a*** (elements in **bold type**) means that *a* is a terminal (including special characters such as parentheses, square brackets, star, plus and vertical bar)
- *a* (elements in *italics type*) means that element definition is not shown for brevity

module ::= *[AlloyModuleDecl]* import* dashParagraph*
dashParagraph::= *AlloyParagraph* | stateDecl | escapeBlock
stateDecl ::= [**default**] [**conc**] **state** *name* **{**stateItem***}**
stateItem ::= stateVar | invariantDecl | initDecl
  | addOnDecl | factoringDecl | enterDecl | exitDecl
invariantDecl ::= **invariant** *[name]* block
initDecl ::= **init** *[name]* block
enterDecl ::= **enter** block
exitDecl ::= **exit** block
addOnDecl ::= **addon** *[name]* **((do** *expr* | **send** *qualName*,$^+$)) **to** transPattern
transPattern ::= * | *qualName*,$^+$
  | ([**from** (* | *qualName*,$^+$)] [**goto** (* | *qualName*,$^+$)])
factoringDecl ::= eventDecl | actionDecl | conditionDecl
  | transDecl | stateDecl | transTemplate
eventDecl ::= [**env**] **event** *name* **{**factoringDecl***}**
actionDecl ::= **action** *name* **[***expr***]{}**
conditionDecl ::= **cond** *name* **[***expr***]{**factoringDecl***}**
transDecl ::= **trans** *[name]* **{**transBody | transInstance**}**
transInstance :: = *qualname***[***expr*,$^+$**]**
transTemplate :: = **def trans** *name* paraDecls **{**transBody**}**
transBody ::= [**from** (* | *qualName*,$^+$)]
  [**on** *qualName*,$^+$] [**when** *expr**]
  [**goto** (* | *qualName*,$^+$)] [**do** *expr**]
  [**send** *qualName*,$^+$]
stateVar ::= **env** *AlloyDecl*
paraDecls ::= **(***AlloyDecl*,* **)** | **[** *AlloyDecl*,* **]**
block ::= **{***expr****}**
escapeBlock ::= **{escape}** *string* **{/escape}**

Figure 3.20: Grammar of Dash

Figure 3.21 lists the keywords of Dash. The new additions to the Alloy language are in **bold** typeface:

abstract	**condition**	**ex**	Int	private
action	**conf**	exactly	int	run
addon	**def**	**exit**	**invariant**	**send**
af	**default**	expect	let	seq
ag	disjoint	extends	lone	set
all	**do**	fact	**many**	sig
and	**ef**	for	module	some
as	**eg**	**from**	no	**State**
assert	else	fun	none	**state**
au	**enter**	**goto**	not	sum
ax	**env**	iden	**on**	**taken**
Boolean	**eu**	iff	one	**to**
but	**Event**	implies	open	**trans**
check	**event**	in	or	univ
conc	**events**	init	pred	when

Figure 3.21: Dash keywords

3.7 Summary

Dash is a language for the specification of system behaviour, which is formally described using a transition system. For this reason, transitions are first-class constructs in Dash. Core Dash provides the minimum set of constructs to describe a behavioural model in Dash. Many other syntactic constructs and features such as *factoring*, *transition comprehension*, and *layering*, provide modellers with the ability to systematically define and organise their transitions, accommodating different modelling paradigms and styles. Temporal properties are supported and can be specified using CTL.

Chapter 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 a model (*i.e.,* the meaning of a model cannot be described by combining individual descriptions of each transition). 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 4.1. A big step consists of one or more small steps, each of which can be one or more transitions. The big step continues until a snapshot 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 a snapshot contains its source state, its trigger event is in the set of current events and its guard condition is satisfied.

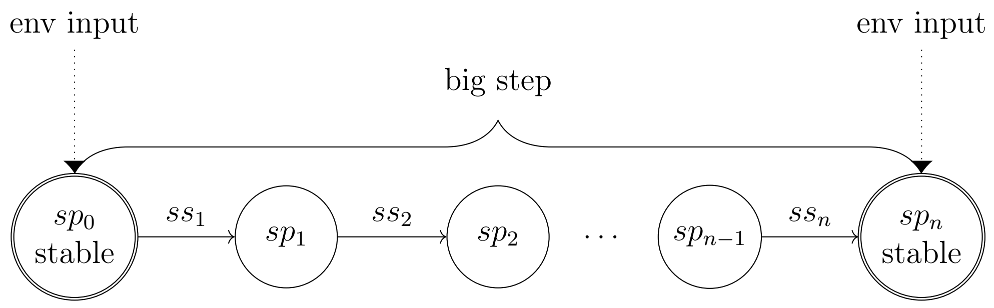


Figure 4.1: Big step (*sp* is a snapshot; *ss* is a small step) (Copied from page 7)

The semantics of Dash are stated in terms of the semantic framework of Esmaeilsabzali *et al.* [20], which describes a space of semantic aspects and options for languages that use big steps. This semantic framework promotes systematicness and clarity, and its modularity lends itself to be declaratively instantiated as Alloy predicates. The choices for each of the semantic options are summarised in Table 4.1 and described on the next sections, 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 a model.

Table 4.1: Semantics of Dash

Semantic Option	**Value in Dash**
CONCURRENCY	Single
BIG STEP MAXIMALITY	Take One
EVENT LIFELINE	Present in remainder of big step
VARIABLE LIFELINE	Immeadiate change in small step
PRIORITY	Source state outer hierarchical

4.1 Concurrency

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[1] are confusing to debug since they make a model inconsistent.

4.2 Big Step Maximality

The BIG-STEP MAXIMALITY aspect specifies the termination criteria for a sequence of small steps, *i.e.,* when the system is stable. The option Take One is chosen for Dash, meaning that at most one transition per concurrent state can be taken in a big step. For reason (2) above, this choice is desired 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

[1]While namespaces force users to recognise when a state is referring to a variable outside of itself, a transition in one state can place constraints on variables outside of its own state.

one transition in each concurrent state should be allowed in a big step because of reason (1) above.

4.3 Event Lifeline

For the EVENT LIFELINE aspect, the option Present in remainder of big step is chosen, which indicates that 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 wanted the small steps to be causal.

4.4 Variable Lifeline

For the VARIABLE LIFELINE[2], the option Immediate change in small step makes 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, the 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 a big step throughout the big step are hard to follow. Because of reason (1) above, it is expected that 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.

4.5 Priority

For PRIORITY, the option Source state outer hierarchical specifies that transitions whose source state is a parent state take precedence over those from a child state. This choice is the most common one in statechart languages, is easier to understand than priority based on scope (source and destination state), and less tedious than specifying explicit relative values for each transition.

[2] In [20], this aspect is decomposed into multiple aspects.

4.6 Characteristics

The set of semantic values chosen 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` to no longer hold, disabling the transition. The non-determinism in the semantics of Dash means that if the same environmental input is applied to two big steps of a model that have the same initial snapshot, the final result and 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. Finally, the semantics of Dash are priority consistent, meaning that always transitions that have higher priority are taken before other transitions with lower priority.

4.7 Frame Problem

Finally, the semantics of Dash 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 transitions, it is assumed that an action will constrain them; if their primed version is not mentioned in any action then variables retain their values from the previous snapshot. This last semantic choice can be overridden and it is controlled by an option of the translation process (see section 5.14.1)

4.8 Summary

The semantics of Dash are not compositional in the structure of a model, and must address the problem of identifying the set of transitions that are enabled by an environmental input.

Environmental input may enable different transitions on different concurrent regions, and for this reason the semantics of Dash use the notion of big and small steps. The big step boundaries are demarcated by stable snapshots, which describe moments when no more transitions are enabled. The semantics of Dash are stated in terms of the framework of Esmaeilsabzali *et al.* [20], which describes a space of semantic aspects and options for these languages. Two main reasons were considered for choosing the semantics of Dash: first, Dash is a declarative language and each transition can describe large changes, and second, ease of understanding of a model. Besides materialising some of the aspects described by the semantic framework, the semantics of Dash also address the mismatch between declarative and control-oriented languages regarding the frame problem.

Chapter 5

From Dash to Alloy

This chapter describes the process of translating a Dash model to an Alloy model that describes a next snapshot relation based on the semantics described in Chapter 4. Additionally, this chapter explains how temporal properties are transformed to be used with TCMC. Throughout the following sections, fragments of abstract code along with excerpts from the bit counter and musical chairs models are used to help in the presentation. The structure of the generated Alloy model follows the guidelines proposed by Farheen [21].

5.1 Next Snapshot Relation

The purpose of a Dash model is to define a next snapshot relation containing pairs that are the possible *small steps* of a system. In Dash, a stable snapshot is the first snapshot of the current big step, and at the same time, is the last snapshot of the previous big step. Usually, in big-step representations there is a *reset* step that clears the outputs generated in a big step and prepares the system for the next environmental input (*e.g.,* [39], [33]). This reset step makes a clear distinction between two consecutive big steps, there are no overlapping snapshots so the last snapshot of the previous big step is different from the first snapshot of the current big step. However, the next snapshot relation of a Dash model is compacted and a stable snapshot is shared between two consecutive big steps. This compact representation avoids having to add an extra reset step and an extra snapshot for every big step which would greatly impact the snapshot scope used for analysis (see section 6.3). Snapshots that are *stable* are characterised by having:

- an unconstrained set of environmental events that can trigger transitions in the next big step;
- internal events that were generated in the previous 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 previous big step.

Figure 5.1 shows a schematic of the next snapshot relation where each small step contains one transition $(t_1, \dots t_n, k_1)$. Each pair of snapshots related in the diagram is in the next snapshot relation. In Dash, an event is either environmental or internal but cannot be both at the same time. Similarly, a variable can either be internal or environmental. *envev* and *envvar* are environmental events and variables, respectively; *gen_ev* represents events generated by transitions; *actions* represents the effects of the actions of transitions where + is used informally; *intvar* represents internal variables and their values; `new` means a non-deterministic choice of values. The internal events generated during a big step, *gen_ev*, are available in the last snapshot of a big step, which is stable, to make them observable because they are outputs of a model. However, these generated events cannot trigger transitions in the next big step, only the environmental events present in the stable snapshot can trigger transitions.

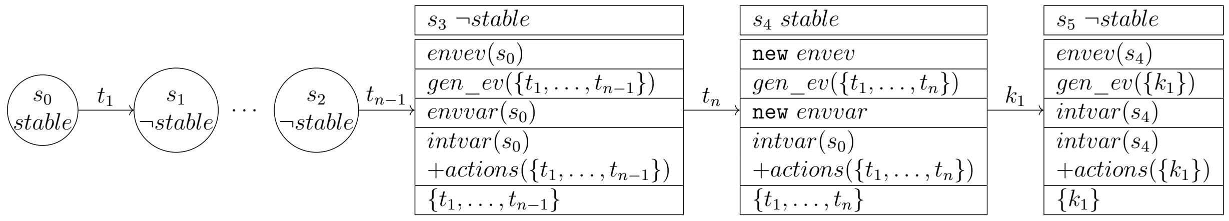


Figure 5.1: Next snapshot relation

5.2 Transformation to Core Dash

The first step in the translation of a Dash model to Alloy is a transformation to Core Dash. The transformation involves unfolding the effects of factoring, expanding transition comprehensions and layering addons. At the end of the transformation, a model consists of a description of a state hierarchy, a set of transitions completely defined, initial constraints and state invariants.

During the unfolding of factoring, transition declarations are complemented with the factoring elements. For example, if unfolding some transitions that have been declared factored by an event, the factoring event is added to the list of triggering events of the transitions. A similar process is done when unfolding the factoring by condition, in which case, the condition is conjuncted to the guard condition of the transitions. If the `from` or `goto` part of a transition declaration are missing, the container state is added to complete the definition of the source and/or destination state. Additionally, during this step transitions declared using a template are instantiated and formal parameters are replaced with actual values.

Next, transition comprehension declarations are expanded. A new transition, identical to the one being expanded (*i.e.,* same trigger events, guard condition, action, and generated events), is created for either: 1) every state name present on the list of source and/or destination states or 2) for every state under the current scope if a wildcard `*` is used. Newly created transitions have the same name as the original declaration, but a numeric suffix is appended to uniquely identify the transitions. Since names of transitions are optional, if no name is given, a generated name with the format `T_counter` will be used, where `counter` is a internal variable that keeps track of the number of transitions generated.

Finally, addons are applied to complete transition declarations. Addon actions are conjuncted to the action part of transitions, and generated events are added to the list of names of generated events.

Once the transformation to Core Dash is completed, the priority (according to the semantic option PRIORITY, see section 4.5) and the set of possible conflicting transitions (according to the semantic option BIG-STEP MAXIMALITY, see section 4.2) are computed for each fully defined transition, and this information is stored to be used on a later step in the translation to Alloy process.

5.3 Snapshot Definition

A snapshots contains the information that represents the state of a system at a moment during its execution. A snapshot is composed of a *configuration* set that contains the label controlled states that are active, a set of *events* that have been generated (either internally by the system or by the environment) and the current values of the system variables. Additionally, a set of transitions is included to keep track of the transitions *taken* during a step, and a boolean flag is used to mark *stable* snapshots. Figure 5.2 shows the representation of a snapshot of musical chairs[1].

```
sig Snapshot {
  conf: set StateLabel,             // set of active control states
  events: set EventLabel,           // events generated
  taken: set TransitionLabel,       // transitions taken during the step
  stable: one Bool,                 // indicates big-step boundaries
  // system variables
  Game_active_chairs : set Chair,
  Game_occupied : Chair set -> set Player,
  Game_active_players : set Player
}
```

Figure 5.2: Musical chairs snapshot representation in Alloy

5.4 State Hierarchy

State hierarchy is encoded using Alloy's subtyping. The state hierarchy of a Dash model is exactly represented by creating a subtyping tree in Alloy. An abstract signature `StateLabel` is the base type for all control states. AND and OR- control states are defined as abstract because they are containers of other control states. Concrete (*i.e.,* non-abstract) signatures are used for basic control states. Signatures of basic states are disjoint. The keyword `one` means that the set contains only one element. Figure 5.3 shows the state hierarchy representation of the bit counter.

[1]The actual translated model differs slightly because an Alloy module is used that contains reusable definitions (see Appendix D), and also because some of the optimisations that are used when generating the Alloy code (see section 5.13).

```
abstract sig StateLabel {}      // base type of all control states
abstract sig Counter extends StateLabel {}
abstract sig Counter_Bit1 extends Counter {}        // container state
one sig Counter_Bit1_Bit11 extends Counter_Bit1 {}  // basic state
one sig Counter_Bit1_Bit12 extends Counter_Bit1 {}
abstract sig Counter_Bit2 extends Counter {}
one sig Counter_Bit2_Bit21 extends Counter_Bit2 {}
one sig Counter_Bit2_Bit22 extends Counter_Bit2 {}
```

Figure 5.3: State hierarchy representation in Alloy

5.5 Events

An event is represented using a signature in Alloy. The distinction between internal and environmental events is materialised by declaring the disjoint Alloy subtypes `EnvironmentalEvent` and `InternalEvent` of a universal abstract `EventLabel` signature. Figure 5.4 shows the event representation of the bit counter.

```
abstract sig EventLabel {}
abstract sig EnvironmentalEvent extends EventLabel {}
abstract sig InternalEvent extends EventLabel {}
one sig Counter_Tk0 extends EnvironmentalEvent {}
one sig Counter_Bit1_Tk1 extends InternalEvent {}
one sig Counter_Bit2_Done extends InternalEvent {}
```

Figure 5.4: Events representation in Alloy

5.6 Transitions

A transitions is identified using an Alloy signature. An abstract signature `TransitionLabel` is declared to be the base type of all transitions. Figure 5.5 shows the transition signatures of musical chairs.

```
abstract sig TransitionLabel {}
one sig Game_Start_Walk extends TransitionLabel {}
one sig Game_Start_DeclareWinner extends TransitionLabel {}
one sig Game_Walking_Sit extends TransitionLabel {}
one sig Game_Sitting_EliminateLoser extends TransitionLabel {}
```

Figure 5.5: Transitions representation in Alloy

A transition relates two snapshots that belong to the step relation. The relation is structured in pre- and post-conditions of the transition, and this is reflected in the model by generating corresponding Alloy predicates. Additionally, a third predicate is generated that contains constraints to enforce some of the semantics options chosen for Dash. The following is the predicate structure for a transition `t1`.

```
pred t1[s: Snapshot, s': Snapshot] {
  pre_t1 [s]              // describes pre-conditions
  post_t1 [s ,s']         // describes post-conditions
  semantics_t1[s, s']     // enforces semantic choices
}
```

Figure 5.6: Predicates of a transition in Alloy

5.6.1 Pre-conditions

The pre-conditions of a transition `t1` are represented using an Alloy predicate which is evaluated relative to the current snapshot, `s`. The predicate is satisfied if the source state of the transition is present in the configuration of the snapshot, and the value of the snapshot variables satisfy the guard condition. Trigger events are evaluated depending whether the current snapshot is the beginning of a bigstep (*i.e.,* stable or not). When the current snapshot is stable, `t1`'s trigger event must be one of the new events from the environment; otherwise its event must be in the snapshot's set of events, which includes the environmental events generated at the beginning of the big step and any internal events generated so far in this big step. Figure 5.7 shows an abstract representation of the preconditions of transition `t1`.

```
pred pre_t1[s:Snapshot] {
  src_state_t1 in s.conf
  guard_cond_t1[s]
  s.stable = True => {
    // beginning of a big step
    // transition can be triggerd only by environmental events
    trig_events_t1 in (s.events & EnvironmentalEvent)
  } else {
    // intermediate snapshot
    // transition can be triggered by any type of event
    trig_events_t1 in s.events
  }
}
```

Figure 5.7: Preconditions predicate of a transition

5.6.2 Post-conditions

Postconditions of a given transition `t1` are modelled using an Alloy predicate which is evaluated relative to the current snapshot, `s`, and the next snapshot, `s'`. Figure 5.8 shows an abstract version of the predicate that models the postconditions of `t1`. The predicate is satisfied if the configuration changes between `s` and `s'` exit the source states and enter the destination states of transition `t1` (line 2). Variable values are updated according to the actions of transition `t1` (line 3, Figure 5.9 shows a concrete example of a transition's actions), to enforce the semantic choice for VARIABLE LIFELINE of Immediate change in small step (see section 4.4). Within this constraint, internal variables whose primed versions are not mentioned in an action are required to retain their values from the previous snapshot[2].

Next, four cases arise to constrain events and environmental variables, depending whether `s` is stable and `s'` will be stable. The predicate `testIfNextStable` evaluates to true, if after taking transition `t1` no more transitions would be enabled, which makes `s'` stable. Then, depending on if the current snapshot, `s`, is stable, two types of small step are possible. When the current snapshot is also stable, the transition is between two stable snapshots, meaning that the small step is equivalent to a big step. In such case, only internal events generated by `t1` are allowed to be contained in the set of events of `s'`. No further constraints are declared, so environmental events and environmental variables can take on new values on `s'`. If the current snapshot is not stable, then the small step is

[2]This behaviour can be turn on/off by a configuration option of the translation tool (see section 5.14.1).

the last one of a big step. In this case, the events generated by `t1` are added to the set of events of `s'` and any previous events generated during the big step are preserved. No further constraints are declared, allowing environmental events and environmental variables to take on new values on `s'`.

On the other hand, if the next snapshot, `s'`, is not stable, environmental variables do not change and two other types of small steps are possible: either the transition corresponds to the first step of a big step, or corresponds to an intermediate small step. In the first case, the current snapshot, `s`, is stable. The only internal events of `s'` are the ones generated by `t1` (clearing any internal events generated on previous big steps), and the environmental events remain the same. When a small step corresponds to an intermediate step, the set of events `s'` is equivalent to the events present in the current snapshot, `s`, with the addition of any event generated by `t1`.

```
pred pos_t1[s:Snapshot, s':Snapshot] {
  s'.conf = s.conf - exit_src_state_t1 + enter_dest_state_t1
  actions_t1[s,s']
  testIfNextStable[s, s', t1, gen_events_t1] => {
     s'.stable = True
     s.stable = True => {
       // big step = one small step
       // only internal events are the ones generated by t1
       // allow env events to change
      no ((s'.events & InternalEvent) - gen_events_t1)
     } else {
       // last small step of the big step
       // add t1's generated events to the internal events
       // allow env events to change
      no ((s'.events & InternalEvent) -
         (gen_events_t1 + InternalEvent & s.events))
    }
  } else {
    s'.stable = False
    env_vars_unchanged_t1[s,s']
    s.stable = True => {
      // first small step of the big step
      // only internal events are the generated by t1
      s'.events & InternalEvent = gen_events_t1
      // env events stay the same
      s'.events & EnvironmentalEvent = s.events & EnvironmentalEvent
    } else {
      // intermediate small step
      // add t1's generated events to the events
      s'.events = s.events + gen_events_t1
    }
  }
}
```

Figure 5.8: Post-conditions predicate of a transition

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 of the `Sit` transition are:

```
pred pos_Game_Walking_Sit[s, s':Snapshot] {
  ...
  s'.Game_occupied in s.Game_active_chairs -> s.Game_active_players
  s'.Game_active_chairs = s.Game_active_chairs
  s'.Game_active_players = s.Game_active_players
  all c : (s'.Game_active_chairs) | one c.s'.Game_occupied)
  all p : Chair.(s'.Game_occupied) | one (s'.Game_occupied).p
   ...
 }
```

Figure 5.9: Actions in a post-conditions predicate of musical chairs

The auxiliary predicate `testIfNextStable`, is evaluated relative to the current snapshot, `s`, the next snapshot, `s'`, the transition to be taken, `t`, and its set of generated events, `genEvents`. 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. For example, to determine if transition `t1` will be enabled after taking transition, `t`, the predicate `enabledAfterStep_t1` is defined using constraints similar to the pre-conditions for `t1`. However, these constraints depend on the variable values of `s'` and the generated events of `t`, to simulate the effects of executing `t`. Additional constraints are added to enforce that only transitions orthogonal to `t1` have been taken.

```
pred testIfNextStable[s,s',t,genEvents] {
  not enabledAfterStep_t1[s,s',t,genEvents] and
  not enabledAfterStep_t2[s,s',t,genEvents] and
  ...
}

pred enabledAfterStep_t1[s, s',t, genEvents] {
  src_state_t1 in s'.conf
  guard_condition_t1[s']
  (s.stable = True) => {
    // only transition taken in big step is t
    // as long as t1 is orthogonal to t
    // then t1 is enabled in next snapshot
    orthogonal_t1[t]
    // t1 can be triggered by environmental events of the big step
    // or by any events generated by t
    trig_events_t1 in {(s.events & EnvironmentalEvents) + genEvents}
  } else {
    // as long as t1 is orthogonal to t + s.taken
    // then t1 is enabled in next snapshot
    orthogonal_t1[t + s.taken]
    // t1 can be triggered by any events present in the big step
    // or any events generated by t
    trig_events_t1 in (s.events + genEvents)
  }
}
```

Figure 5.10: Determining if a transition will be enabled on next snapshots

5.6.3 Semantics

The semantics predicate for `t1` is evaluated relative the current snapshot, `s`, and the next snapshot, `s'`. The predicate is true if `t1` 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 (see section 4.2). 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 the negation of the pre-condition of the higher priority transition is included in this semantics predicate to enforce the choice of Source state Outer Hierarchical for the PRIORITY semantic aspect (see section 4.5). Additionally, if the snapshot `s` is stable, then this is the first step of a big step and only `t1` should be included in the set of transitions; otherwise, `t1` is added to the set of transitions. This last constraint ensures that only one transition is taken in a step (enforcing Single semantic choice for CONCURRENCY) (see section 4.1).

```
pred semantics_t1[s,s':Snapshot] {
  s.stable = True => {
    s'.taken = t1             // SINGLE semantics
  } else {
    s'.taken = s.taken + t1   // SINGLE semantics
    orthogonal_t1[s.taken]    // TAKE ONE semantics
  }
  !pre_t2[s]                  // transition with higher priority
  !pre_t3[s]
  ...
}
```

Figure 5.11: Semantics predicate of a transition

For the bit counter the semantics predicate of one of the transitions is as follows:

```
pred semantics_Counter_Bit1_T1[s, s': Snapshot] {
  s.stable = True => {
    s'.taken = Counter_Bit1_T1              // SINGLE semantics
  } else {
    s'.taken = s.taken + Counter_Bit1_T1    // SINGLE semantics
    // TAKE ONE semantics
    no s.taken & {                 // no other transition in the
        Counter_Bit1_T2 +          // same region as T1 has been taken
        Counter_Bit1_T1
    }
  }
}
```

Figure 5.12: Semantics predicate for `T1` of the bit counter

5.7 Initial Constraints

Initial constraints are modelled by an Alloy predicate. The generic initial constraints on snapshots are that the system is in its default states, no transitions have been taken, and there are no internal events. Environmental events can be present in an initial snapshot to enable transitions. Furthermore, an initial snapshot must be stable. Additional constraints defined by the model are appended to the predicate. Figure 5.13 shows the initial constraints for musical chairs.

```
pred init[s: Snapshot] {
  s.conf = default_states
  no s.taken
  no s.events & InternalEvent
  s.stable = True
  // model-specific constraints
  #(s.Game_active_players) > 1
  #(s.Game_active_players) = (#(s.Game_active_chairs)).plus[1]
  s.Game_active_players = Player
  s.Game_active_chairs = Chair
  s.Game_occupied = none -> none
}
```

Figure 5.13: Initial constraints for musical chairs

5.8 State Invariants

A state invariant is modelled using an Alloy fact on a snapshots. Since a state invariant is local (it depends on the state that contains its declaration), it is expressed as an implication. Figure 5.14 shows the associated Alloy fact for an invariant of musical chairs that states that when the game is in the `Walking` state, no one is occupying a chair.

```
fact Game_Walking_emptyChairs {
  all s: Snapshot | Game_Walking in s.conf => {
    no s.Game_occupied
  }
}
```

Figure 5.14: State invariants in Alloy

5.9 Model Definition

A Dash model is translated to Alloy to describe a next snapshot relation, which is defined in terms of the small steps. To model the small steps, a predicate is created which contains the disjunction of the transition predicates of a model, and an Alloy fact is included to formally define the next snapshot relation. As described in [21], the small steps can also be modelled as the conjunction of the individual transition predicates, granted that transitions are structured in a manner that the preconditions of a transition imply the post-conditions of the transition (`pre_t1 => pos_t1`). However, the disjunction of the transitions provides more structure and modularity; adding a new transition does not change the behaviour of existing transitions.

Snapshots that satisfy the initial constraints are defined as the initial snapshots of a model, and pairs of snapshots that satisfy the `small_step` predicate conform the next step relation. An additional predicate, `equality`, is defined to avoid duplicate snapshots. In Alloy, by default every atom is different even when a pair of atoms contain the same values. Lastly, a `significance` predicate may be included, which contains constraints to get a significant model (see section 5.10).

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

fact {
  all s: Snapshot | s in initial iff init[s]
  all s, s': Snapshot | s->s' in nextStep iff small_step[s, s']
  all s, s': Snapshot | equals[s, s'] => s = s'
  significance
}

pred equals[s, s': Snapshot] {
  s'.conf = s.conf
  s'.events = s.events
  s'.taken = s.taken
  // Model specific declarations
  s'.Game_active_chairs = s.Game_active_chairs
  s'.Game_active_players = s.Game_active_players
  s'.Game_occupied = s.Game_occupied
}
```

Figure 5.15: Model definition in Alloy

5.10 Significance Axioms

The significance axioms help in ensuring the exploration of a portion of the snapshot space that contains interesting behaviours, and in avoiding the generation of spurious counterexamples when model checking. Farheen [21] defines two axioms: the reachability axiom and the operations axiom.

The reachability axiom ensures that every snapshot generated is reachable from an initial snapshot, and that such an initial snapshot exists. The operations axiom ensures that there is at least one representative of every transition in a model.

```
pred significance {
  reachabilityAxiom
  operationsAxiom
  completeStepsAxiom
}

/** Every snapshot is reachable from an initial snapshot */
pred reachabilityAxiom {
  all s : Snapshot | s in Snapshot.((initial) <: * (next_step))
}

/** There exists at least one representative of every transition */
pred operationsAxiom {
  some s, s': Snapshot | T1[s, s']
  some s, s': Snapshot | T2[s, s']
  some s, s': Snapshot | T3[s, s']
  ...
}
```

Figure 5.16: Significance axioms in Alloy

For Dash, we extended the significance axioms to concurrent models. A new axiom is introduced to ensure that instances of a model are composed by complete big steps. The axioms is defined as:

Complete big steps axiom: All big steps of a concurrent model must be complete. This axiom prevents spurious instances where branching small steps are generated but do not end in stable snapshots, which would produce incomplete big steps.

```
/** An unstable snapshot cannot be the last one of a trace */
pred completeStepsAxiom {
  all s: Snapshot | s.stable = False => some s.nextStep
}
```

Figure 5.17: Complete steps significant axiom in Alloy

5.11 Temporal Properties

Temporal properties are specified in Dash using CTL formulas in the body of Alloy predicates and/or assertions. When a model is translated to Alloy, temporal expressions are converted to set comprehension expressions that can be used with the Alloy implementation of TCMC. Additionally, since the next snapshot relation of models with concurrency may contain big steps composed by several small steps, an expansion of temporal properties is required to correctly check properties at stable snapshots. At the Dash level when a property uses the Next temporal operators (`ax, ex`), it should refer to the next stable snapshot of the transition relation, which might not be the same as the actual next snapshot in the small step relation. In these cases, the temporal property is transformed to use the corresponding Until operator (`au, eu`). For example, a property expressed in Dash with the form $AG(p \implies EX\ q)$ is translated to Alloy as $AG(stable \wedge p \implies EX(\neg stable\ EU(stable \wedge q)))$. These transformation apply at any level in a nested property. Figure 5.18 shows some of the translated temporal properties of musical chairs.

```
/**
 * Dash model
 */
assert ctl_safety {
  ag (#Game/active_players = (#Game/active_chairs).plus[1])
}

pred ctl_existential {
  ef (Game/End in conf and Game/active_players = Alice)
}

/**
 * Generated Alloy model
 */
assert ctl_safety {
  ctl_mc[
    ag[{s: Snapshot |
      #(s.Game_active_players) = (#(s.Game_active_chairs)).plus[1]
    }]
  ]
}

pred ctl_existential {
  ctl_mc[
    ef[{s: Snapshot |
      Game_End in s.conf and s.Game_active_players = Alice
    }]
  ]
}
```

Figure 5.18: Temporal properties in Alloy

5.12 Alloy Paragraphs and Escape Blocks

Regular Alloy constructs supported by Dash such as signatures, predicates, functions and facts, are copied as they are originally declared at the Dash level (with the exception of temporal expressions which require a transformation). Escape blocks are copied verbatim to the generated Alloy model.

5.13 Optimisations

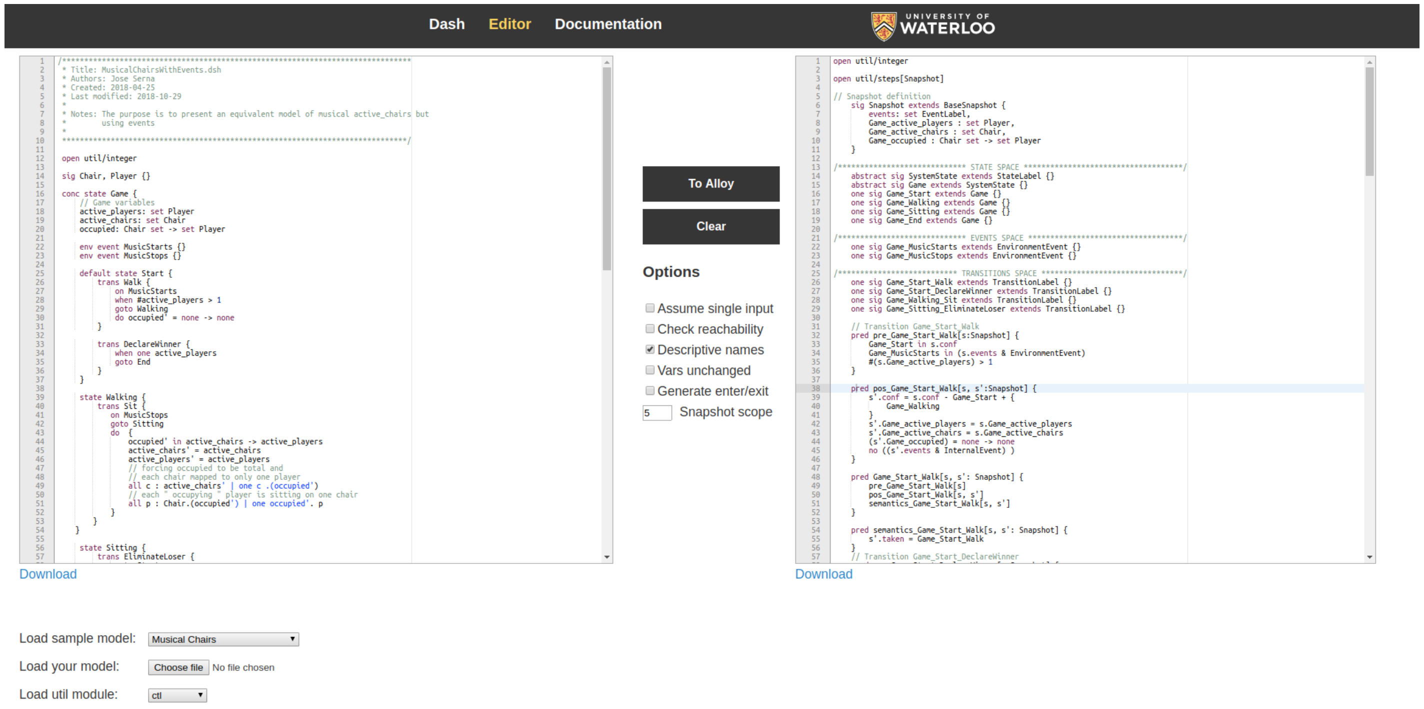
Some sensible modifications are made to signatures and predicates to simplify generated models. The optimisations depend on the following conditions of a model: events are not used and/or no concurrent states are declared.

When 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 can be assumed to be stable, making every small step equivalent to a big step. This assumption, 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.

5.14 Implementation

Dash has been implemented using Xtext [7], a framework for the development of programming and domain-specific languages. The framework provides the necessary tool support for parsing, linking, type checking and compiling. Advanced editing capabilities such as syntax highlighting, code completion, and marking of occurrences, can be easily developed and shipped as an Eclipse plugin or a web editor. Our implementation checks several well-formedness constraints such as that all state declarations are of the same type at the same level of a state hierarchy, environmental events are not generated by a transition, and actions do not constrain the next value of environmental state declarations. Furthermore, a simple type checking [3] is performed on all expressions to ensure for example, that only formulas are used with boolean operators or that relations with the appropriate arity and type are used for set operations. The translator tool for Dash is available as a web interface at `http://129.97.7.33:8080/dash`. The website contains further documentation about the language, and an online editor with several sample models to help users familiarise with the syntax and features.

[3]The type checking of our implementation is not as complete or advanced as the implementation in Alloy, and many errors will only be reported on the translated Alloy model.



```
/*******************************************************************************
 * Title: MusicalChairsWithEvents.dsh
 * Authors: Jose Serna
 * Created: 2018-04-25
 * Last modified: 2018-10-29
 *
 * Notes: The purpose is to present an equivalent model of musical active_chairs but
 *        using events
 *
 *******************************************************************************/

open util/integer

sig Chair, Player {}

conc state Game {
    // Game variables
    active_players: set Player
    active_chairs: set Chair
    occupied: Chair set -> set Player

    env event MusicStarts {}
    env event MusicStops {}

    default state Start {
        trans Walk {
            on MusicStarts
            when #active_players > 1
            goto Walking
            do occupied' = none -> none
        }

        trans DeclareWinner {
            when one active_players
            goto End
        }
    }

    state Walking {
        trans Sit {
           on MusicStops
           goto Sitting
           do  {
               occupied' in active_chairs -> active_players
               active_chairs' = active_chairs
               active_players' = active_players
               // forcing occupied to be total and
               // each chair mapped to only one player
               all c : active_chairs' | one c .(occupied')
               // each " occupying " player is sitting on one chair
               all p : Chair.(occupied') | one occupied'. p
           }
        }
   }

    state Sitting {
        trans EliminateLoser {
```

```
open util/integer

open util/steps[Snapshot]

// Snapshot definition
    sig Snapshot extends BaseSnapshot {
        events: set EventLabel,
        Game_active_players : set Player,
        Game_active_chairs : set Chair,
        Game_occupied : Chair set -> set Player
    }

/*************************** STATE SPACE **********************************/
    abstract sig SystemState extends StateLabel {}
    abstract sig Game extends SystemState {}
    one sig Game_Start extends Game {}
    one sig Game_Walking extends Game {}
    one sig Game_Sitting extends Game {}
    one sig Game_End extends Game {}

/*************************** EVENTS SPACE *********************************/
    one sig Game_MusicStarts extends EnvironmentEvent {}
    one sig Game_MusicStops extends EnvironmentEvent {}

/************************* TRANSITIONS SPACE ******************************/
    one sig Game_Start_Walk extends TransitionLabel {}
    one sig Game_Start_DeclareWinner extends TransitionLabel {}
    one sig Game_Walking_Sit extends TransitionLabel {}
    one sig Game_Sitting_EliminateLoser extends TransitionLabel {}

    // Transition Game_Start_Walk
    pred pre_Game_Start_Walk[s:Snapshot] {
        Game_Start in s.conf
        Game_MusicStarts in (s.events & EnvironmentEvent)
        #(s.Game_active_players) > 1
    }

    pred pos_Game_Start_Walk[s, s':Snapshot] {
        s'.conf = s.conf - Game_Start + {
            Game_Walking
        }
        s'.Game_active_players = s.Game_active_players
        s'.Game_active_chairs = s.Game_active_chairs
        (s'.Game_occupied) = none -> none
        no ((s'.events & InternalEvent) )
    }

    pred Game_Start_Walk[s, s': Snapshot] {
        pre_Game_Start_Walk[s]
        pos_Game_Start_Walk[s, s']
        semantics_Game_Start_Walk[s, s']
    }

    pred semantics_Game_Start_Walk[s, s': Snapshot] {
        s'.taken = Game_Start_Walk
    }
    // Transition Game_Start_DeclareWinner
```

- [ ] Assume single input
- [ ] Check reachability
- [x] Descriptive names
- [ ] Vars unchanged
- [ ] Generate enter/exit
- 5 Snapshot scope

Load sample model: Musical Chairs

Load your model: Choose file No file chosen

Load util module: ctl

Figure 5.19: Dash online tool interface

5.14.1 Tool Options

The online tool of Dash lets users control some of the parameters of the translation process. The available options are:

- *Assume single input*: This option controls the generation of an Alloy fact that constraints the number of events present in snapshots. When active, at most one environmental event can be present in every snapshot, allowing users to verify their models using the single event hypothesis.
- *Check reachability*: This option controls the automatic generation of temporal properties to check that every basic state of a model is reachable (*i.e.,* properties of the form EF).
- *Descriptive names*: When this option is checked, the names of Alloy elements are generated based on the Dash qualified names, which makes it easier to relate Alloy constructs with their corresponding Dash declarations. When the option is not checked, an auto-generated name is used instead.
- *Vars unchanged*: This option controls what happens to snapshot variables whose primed version are not present on the actions of a transition. When the option is selected, predicates are included to force unconstrained snapshot variables to retain the same value as in the previous snapshot.
- *Generate enter/exit*: This option controls the generation of Alloy functions to compute the entered and exited states of a transition. When the option is not selected, these states are directly listed on transitions post-conditions. This option is an additional optimisation of the translation; it makes a model more concise, however, the impact on analysis time is yet to be determined.
- *Snapshot scope*: This option indicates the number of snapshots to be used when generating the scopes for auto-generated reachability properties. The option helps to find a significant scope (see section 6.3.1).

5.15 Summary

A Dash model is a description of the behaviour of a system that is formalised by declaring a next snapshot relation in Alloy. The translation process of a Dash model to Alloy begins with a transformation to Core Dash, by unfolding the effects of *factoring*, expanding *transition comprehensions* and layering *addons*. At the end of the transformation a description of a state hierarchy, a set of transitions, initial constraints and state invariants is obtained. Core Dash models are then formalised using Alloy signatures, predicates and facts that contain constraints to enforce the semantics choices of Dash. Temporal properties are expanded to check their validity at the boundaries of big steps. *Significance axioms* are extended for concurrent models and implemented in the generated Alloy code to avoid spurious model instances and force the Alloy Analyzer to explore parts of the snapshot space with interesting behaviours. A translator tool is available at `http://129.97.7.33:8080/dash` that contains multiple sample models to help users familiarise with the syntax and features of Dash.

Chapter 6

Case Studies

This chapter presents several case studies[1] that have been developed to demonstrate the modelling capabilities of Dash, the translation to Alloy, and model checking analysis of Dash models. The models range over the control-oriented versus data-oriented spectrum, and are characterised based on features such as state hierarchy, concurrency and use of events.

6.1 Models and Characteristics

In addition to the game musical chairs and the two-bit counter (introduced on section 3.1), four more case studies were developed to evaluate Dash. Each model has several features such as the use of events, state hierarchy, concurrency and data abstractions that help to characterise the model and place it on the control versus data-oriented spectrum. Table 6.1 summarises the characteristics of each model and Figure 6.1 places each model on the spectrum.

[1]The case studies are available as sample models at `http://129.97.7.33:8080/dash`.

Table 6.1: Characteristics of case studies

Model	**Control States**				**Events**	**Transitions**	**Data Abstractions**
	Total	**Basic**	**Concurrent**	**Hierarchy Depth**			
Farmer puzzle	1	1	0	0	0	2	2 relations
Musical chairs	5	4	0	1	2	4	3 relations
NASA FGS	63	33	16	5	2	43	53 boolean in/out
Snapshot UI	8	5	0	2	7	7	0
Traffic light	9	6	2	2	2	6	0
Bit counter	7	4	2	2	3	4	0

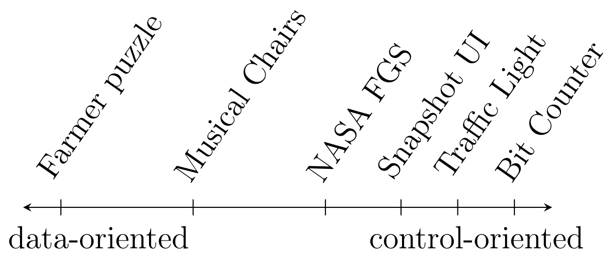


Figure 6.1: Model spectrum.

Each of the case studies was developed based on an existing Alloy model or a statecharts representation. The models were constructed to guide the integration between Alloy and statecharts, so none of the novel features of Dash were used, with the exception of factoring transitions by state and the specification of some state invariants.

6.1.1 Flight Guidance System

The NASA Flight Guidance System (FGS) [16] is a component that generates pitch and roll values for an aircraft to minimise the difference between a desired state (position, speed, altitude) and measured values. The mode logic is the sub component of the FGS that specifies the modes, which are abstractions of the current state of the flight control, and the rules for transitioning between them. The mode logic is divided into two different components: event processing to prioritise events, and flight modes to control the logic for transitions. As part of the case study only the flight modes have been modelled and the event processing has been left unconstrained; hence some of the properties checked fail because there is no explicit priority of the occurrence of events.

6.1.2 Farmer Puzzle

The farmer puzzle [6] is a classic riddle where a farmer is to cross a river taking a chicken, a sack of grain and a fox safely to the other shore. The farmer crosses using a small boat and can only take one of the items at a time. Some constraints are in place that dictate which item can be safely carried on the boat or left on the shore. For example, the fox and the chicken cannot be left alone on the shore while the farmer crosses with the sack of grain, because then the fox would eat the chicken.

6.1.3 Snapshot UI

Snapshot [2] [4] is an application that allows teachers to assign quizzes or "snaps" to students. Results are summarised and presented as several reports. The model captures the behaviour of the application and how users can navigate through the different reports.

6.1.4 Traffic Light

The traffic light [19] example models the behaviour of a traffic light controller at an intersection that controls the lights on the north-south direction and east-west direction. The lights change according to two environmental events: *end* and *change*, which are assumed to occur following the sequence $end, change, end, \ldots$

6.2 Translation

Table 6.2 shows the sizes of the Dash models and their translation to Alloy (without helper files) 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. The available hand-crafted Alloy models (prepared

[2]Snapshot is the name of the app and should not be confused with the definition used in this thesis as a mapping to values that represents a moment in the execution of a system

prior to this work) are all smaller than the generated Alloy models, however, the Dash translation has to cover the generality of all the variations of control state hierarchy and the big steps that concurrency creates. The Dash models are similar in size (or smaller in one case) than the hand-crafted Alloy models.

Given the popularity of UML statemachines and this modelling paradigm, Dash provides a natural transition for these modellers into abstract formal representations of data operations. Dash has enhanced Alloy with the ability to model transition systems that include control state hierarchy and events, thus providing structure to Alloy models. This enhancement makes it possible for Dash to be used to model systems all across the spectrum, ranging from data-intensive models to highly hierarchical and control-oriented models.

Table 6.2: SLOC comparison between Dash and Alloy

Model	**Hand-crafted Alloy**	**Dash**	**Generated Alloy**
Farmer puzzle	27	41	100
Musical Chairs	116	51	160
NASA FGS	-	723	5246
Snapshot UI	32	49	205
Traffic Light	-	44	431
Bit Counter	-	37	349

6.3 Model Checking

After translating a Dash model to Alloy, Model checking can be done using TCMC or BMC. BMC checks properties of finite traces. TCMC checks infinite traces within finite subsets of the reachable snapshot space. Both techniques in Alloy are currently substantially limited in the number of snapshots that they can check; usually much less than the reachable snapshot space even for a finite model. However, we expect the performance of constraint-based model checking (*e.g.,* IC3 [12]) will continue to improve. Thus, the purpose of this evaluation is to show that model checking analysis of Dash models is possible with some useful results now. The model checking results are summarized in Table 6.3. The analysis was executed 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.

Table 6.3: Results of case studies. Properties were checked with *TCMC* or BMC. <u>**Entries**</u> are properties that fail. N/A means below significant scope.

Model	**Property**	**Snapshot Scope**				
		7	**8**	**9**	**10**	**12**
Musical Chairs	Always more players than chairs	N/A	0.7s	1.4s	3.1s	20.0s
	Alice wins the game	N/A	0.3s	0.04s	0.1s	1.6s
	Alice wins the game	N/A	0.1s	0.2s	0.02s	4.6s
	Players sit during the game	N/A	0.05s	0.1s	0.1s	0.5s
	Game eventually finishes	N/A	0.6s	3.4s	13.7s	4.4m
Bit Counter	Model is responsive	0.03s	0.05s	0.04s	0.1s	0.01s
	Final bit status	0.2s	0.3s	0.6s	1.4s	4.0s
Traffic Light	Both lights not green	0.1s	0.3s	0.8s	2.3s	14.6s
Farmer puzzle	No quantum objects	0.6s	3.9s	1.1m	19.3s	10.6m
Snapshot UI	Answers through students	0.2s	0.5s	1.5s	2.8s	10.0s
	Logs out and logins back	0.01s	0.01s	0.01s	0.02s	0.13s
NASA FGS	*At most one lateral mode active*[a]	N/A	N/A	<u>**1.6m**</u>	<u>**7.4m**</u>	<u>**20.9m**</u>
	At most one lateral mode active[a]	N/A	N/A	<u>**2.0m**</u>	<u>**1.5m**</u>	<u>**12.3m**</u>
	AP engaged implies modes on	N/A	N/A	1.2m	4.6m	44.7m
	AP engaged implies modes on	N/A	N/A	>10hr	>10hr	>10hr
	Onside FD on implies modes on[b]	N/A	N/A	<u>**35.2s**</u>	<u>**23.4s**</u>	<u>**2.41m**</u>
	ROLL Selected iff ROLL active	N/A	N/A	35.8s	1.1m	6.1m

[a] Property fails because the event processing is unconstrained.
[b] Property fails because there is no fixed order of execution of orthogonal regions.

To enable model checking analysis of Dash models, three problems were addressed: 1) ensuring the properties to be checked are stated at the Dash language level (not the Alloy level) so users do not have to be aware of the distinction between big steps and small steps; 2) determining a reasonable snapshot scope; and 3) evaluating how easy it is to understand the counterexamples from the Alloy Analyzer.

For all the case studies, the analysis begins with auto-generated, application-independent properties such as the reachability of basic states (see 5.14.1). Then, properties specific to each model are checked.

As explained on section 5.11, temporal properties specified in Dash are transformed and expanded to guarantee that they are correctly checked on stable snapshots, so users do not need to be aware of the distinction between the types of steps. For example, for the FGS system the requirement that *The mode annunciations shall be on in the next step if the AP is engaged* is specified in Dash as

$$AG(isAPEngaged \Rightarrow AX\ modesOn)$$

The “next” step may include several small steps before the model reaches a stable snapshot because of concurrency in the model, so the original CTL *next* (AX) operation is converted and expressed in Alloy as an *until* operation

$$AG(stable \land isAPEngaged \Rightarrow AX(\neg stable\ AU\ (stable \land modesOn)))$$

6.3.1 Significant Scopes

To determine a reasonable snapshot scope, we follow the guidance of *significant scopes* [21] to ensure that the subset of the reachable snapshot space checked has interesting behaviour in it. For simple models, the significant scope is one where every transition can be taken at least once. For Dash models, we extended the concept of significant scope based on the hierarchical control state structure of a model. The significant scope is one where every labelled control state can be reached. For the FGS model, this scope is reasonable because the concurrent states are mostly independent. Longer traces are formed by interleaving shorter subtraces, so verifying these shorter traces provides useful feedback about the complete system. The significant scope for the case studies are: bit counter (7), farmer puzzle (8), Snapshot UI (7), traffic light (7), musical chairs (8), and NASA FGS (9).

6.3.2 Model Visualisation

Through careful design of the structure of the snapshots signatures in Alloy and the Alloy Analyzer's support for themes, instances of a model can be made to resemble the schematic of the next snapshot relation depicted in Figure 5.1, which clearly shows the transitions taken in steps making it possible to interpret model checking results in terms of the original Dash models[3].

Model instances in Alloy can be presented in different formats for inspection. Alloy supports presentation of a model as text, a table, a tree or a graph. For behavioural models,

[3]A complete description of how to configure the Alloy Analyzer to display instances is available at `http://129.97.7.33:8080/dash/documentation/tutorial.html#visualizing-model`

the graph representation is appropriate because it can clearly show the steps and related changes of a model. Figure 6.2 shows a significant instance of the bit counter model, as a graph, using the default settings of Alloy. By default, all atoms are displayed as rectangles and relations are displayed as arcs. These settings clutter the visualisation of a model making it hard to understand.

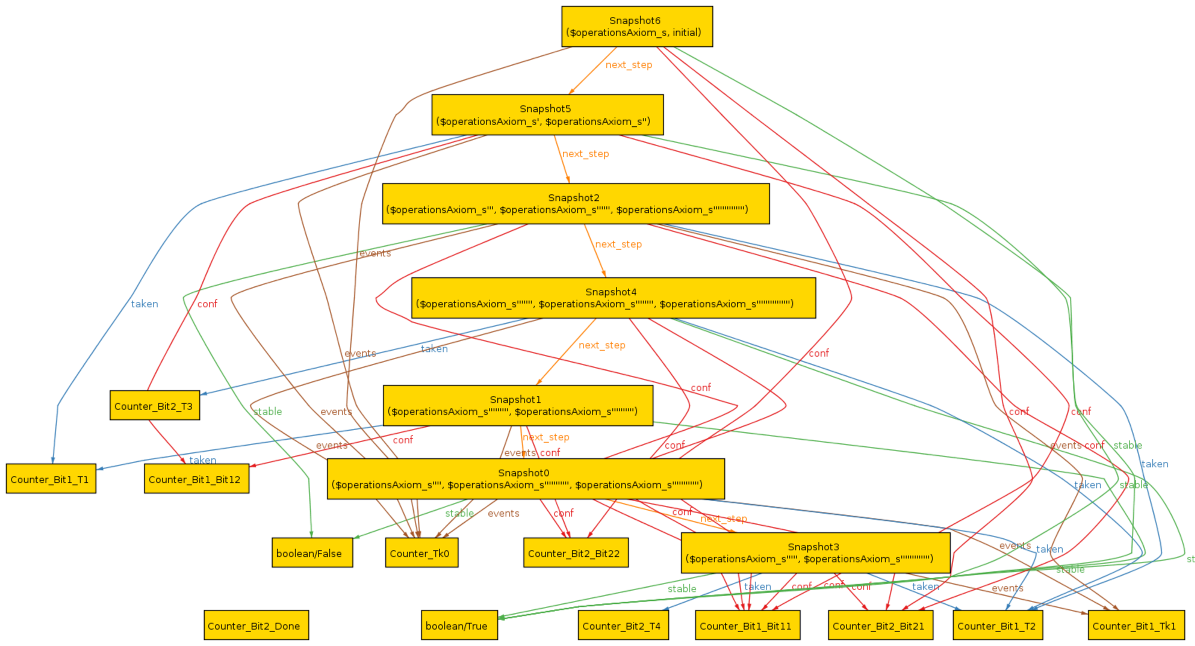


Figure 6.2: Default visualisation of an Alloy model

In our translation, the snapshot signature act as a package for a model's variables and some context information. For this reason, we choose to display snapshot atoms as rectangles, and the other signature elements as attributes. The only relation displayed as an arc is `next_step`, which relates snapshots and highlights the steps. Figure 6.3 shows the same instance of the bit counter, but using a theme with the settings described above. The graph clearly presents the transitions taken during a step, and the value of variables in each snapshot, which facilitates the understanding of an model.

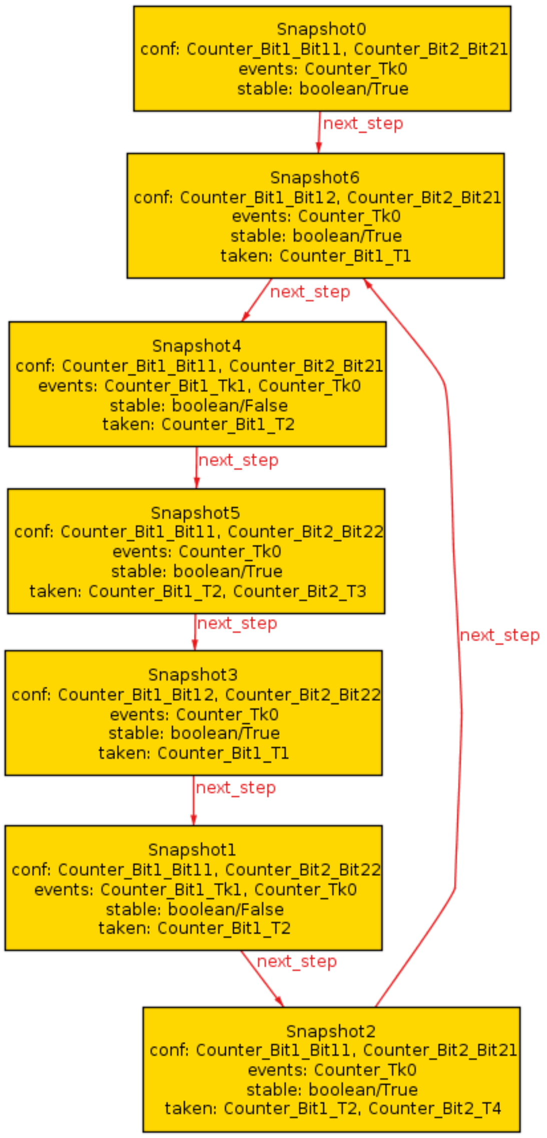

Snapshot0
conf: Counter_Bit1_Bit11, Counter_Bit2_Bit21
events: Counter_Tk0
stable: boolean/True
next_step
Snapshot6
conf: Counter_Bit1_Bit12, Counter_Bit2_Bit21
events: Counter_Tk0
stable: boolean/True
taken: Counter_Bit1_T1
next_step
Snapshot4
conf: Counter_Bit1_Bit11, Counter_Bit2_Bit21
events: Counter_Bit1_Tk1, Counter_Tk0
stable: boolean/False
taken: Counter_Bit1_T2
next_step
Snapshot5
conf: Counter_Bit1_Bit11, Counter_Bit2_Bit22
events: Counter_Tk0
stable: boolean/True
taken: Counter_Bit1_T2, Counter_Bit2_T3
next_step
Snapshot3
conf: Counter_Bit1_Bit12, Counter_Bit2_Bit22
events: Counter_Tk0
stable: boolean/True
taken: Counter_Bit1_T1
next_step
Snapshot1
conf: Counter_Bit1_Bit11, Counter_Bit2_Bit22
events: Counter_Bit1_Tk1, Counter_Tk0
stable: boolean/False
taken: Counter_Bit1_T2
next_step
Snapshot2
conf: Counter_Bit1_Bit11, Counter_Bit2_Bit21
events: Counter_Tk0
stable: boolean/True
taken: Counter_Bit1_T2, Counter_Bit2_T4
next_step

Figure 6.3: Improved visualisation of an Alloy model by using themes

6.4 Summary

Several case studies were developed to demonstrate the modelling and model checking capabilities of Dash. In Dash, we can model systems all over the control-oriented versus data-oriented spectrum. Our case studies are characterised based on features such as state hierarchy, concurrency and use of events. While hand-crafted Alloy models may be more concise, the translation of Dash models handles the generality that all variations of state hierarchy and concurrency creates. Properties can be specified directly at the Dash level, so users do not need to be aware of the distinction between small and big steps. An extension to the notion of significant scope is introduced for Dash models and used to ensure that a big enough portion of the snapshot space is searched for behaviours. Careful design of the snapshot signature definition and Alloy's support for themes are used to present instances of a model that clearly show the transitions taken on a step helping users to interpret the results in terms of original Dash models.

Chapter 7

Related Work

Dash is a *declarative* language for the specification of behavioural models that *integrates statecharts*-like constructs with *Alloy*, which makes it possible to integrate descriptions of systems with complex behaviours and rich data structures and operations. This chapter describes some similarities and differences between Dash and several other formalisms. The chapter is organised by first comparing Dash with other declarative languages, then a comparison between Dash and other languages based on statecharts and Alloy is given, respectively, and finally, a comparison between Dash and other integrated languages is presented.

7.1 Declarative modelling languages

Declarative modelling languages such as B [9], TLA$^+$ [54], VDM [31], Z [48], are based on first-order logic and/or set theory abstractions to formally describe systems with complex structure, and behaviour is captured by means of state change representation through the use of primed expressions. However, these formalisms lack support for the specification of more complicated control-oriented systems, leaving modellers with the task of developing ad-hoc procedures and/or relying on guidelines and conventions to describe behavioural aspects of reactive systems such as concurrency and synchronisation (*e.g.,* [45, 27]). Dash provides statecharts-like constructs for the specification of control-oriented behaviour and offers different syntactic features such as factoring, transition comprehension and layering to accommodate various modelling paradigms. Furthermore, some of these languages have a strong mathematical notation that may intimidate some users (*e.g.,* VDM, Z), and most of the tool support has been developed around refinement with the generation of proof obligations that require theorem proving for verification. In contrast, Dash is based on Alloy whose syntax would be familiar to most programmers (Alloy's syntax resembles common object-oriented constructs) and which has been designed with an emphasis on automatic analysis [29].

7.2 Languages based on statecharts

The statecharts family of languages (prominently represented by UML statemachines [2]) usually have a fixed condition and action language that does not allow for declarative specification of user-defined data types and operations. 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 `addon` 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.,*[57] [32]) have been proposed to express temporal constraints, the official specification does not support this type of constructs.

Zhao and Krogh [56] developed *sf2smv*, a tool that generates the stepping transition relation of Stateflow diagrams [5] (a variant of the statecharts formalism developed by MathWorks) in SMV [36] to model check their behaviour. Temporal constraints, expressed in CTL, are expanded to take into account the difference between small steps and big steps, making sure that properties are checked at stable snapshots. In Dash, a similar approach is followed for the translation of properties to ensure the correct verification of behaviour. On the other hand, the semantics of Dash differ from the semantics of *sf2smv*. In *sf2smv*, it is assumed that the transition relation has deterministic steps, however, in Dash any valid sequence of interleaving steps is allowed. Non-deterministic steps is a desired quality in Dash because the language is tailored at the specification of abstract models of requirements.

CASL-Charts [41] is a language that integrates statecharts with the algebraic specification language CASL. Data operations are axiomatized and transition triggers are expressed in the CASL language, however, transition actions are still a sequence of assignments or event triggers as in statecharts. Its semantics are defined as a combination of the languages rather than a mapping to CASL.

In OZS [24], 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.

7.3 Languages based on Alloy

Chang and Jackson [14] propose a simple modelling language that integrates relational and temporal logic. The language allows writing specifications in a declarative or imperative manner, and supports relational operators from Alloy, imperative constructs (*e.g.,* assignments, procedures), control operators (*e.g.,* if-then-else, for loops), integer arithmetic, and temporal logic (CTL). A prototype model checker was developed based on BDDs demonstrating that is possible to embed expressive relational operations into standard CTL symbolic model checkers.

DynAlloy [22, 23, 42] extends Alloy with actions to represent state change, and uses 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. Similar to Dash, variables in DynAlloy whose primed versions do not appear in a post-condition retain their values. Analysis is done via optimized translation of a DynAlloy model to the Alloy language and then run in the Alloy Analyzer.

Electrum [34] is a language that combines Alloy with TLA$^+$, designed for the specification of systems with rich structural properties and their evolution over time. Expressions are described using the Alloy language, and temporal operators and primed variables (used to refer to next snapshot values) from TLA$^+$are used to specify behaviour. Additionally, signatures and fields may be tagged as variable so their values can change over time. Electrum offers two modes of model checking: bounded and unbounded model checking. In the first case, an Electrum model is translated to Alloy and a time signature is explicitly introduced and ordered to represent traces. Analysis is done via the regular Alloy Analyzer. In the unbounded model checking version, an Electrum model is reduced to an LTL specification and then directly encoded into nuXmv [13] for the analysis.

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 either of the previously discussed languages. Furthermore, in Dash, variables can be marked 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 modifiable as in Electrum), which matches the idea of reactive systems as describing the system interactions with its environment. For analysis, through scoped TCMC, Dash supports scoped CTL temporal logic model checking, without relying on extensions to the Alloy Analyzer.

7.4 Integrated languages

TCOZ [35] 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 as in [17] where a specification is projected into Timed Automata.

Circus [52] is another integrated language that combines Z and CSP, and its purpose is to provide a refinement calculus for the development of concurrent programs. Imperative constructs are supported (*e.g.,* assignments, conditionals and loops) alongside declaration of channels, processes and actions. Ongoing work is being done on the development of a model checker for specifications written in the Circus [37]. In Circus, the integration between Z and the process algebra is done at the syntactical level; it allows for deeper integration of the notations but the semantics of Z and CSP constructs have to be defined. In contrast, the integration in Dash is at the semantic level. Dash combines the meaning of control constructs provided by statecharts with relational notions of Alloy.

CSP-CASL [43] blends CSP with CASL [10] (an algebraic language that allows modular and hierarchical specifications). Data types are defined in CASL libraries, and declaration of processes and communication channels are used for control specification. Refinement can be applied to both data and processes, and tool support is available [40].

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 which presents different challenges for stating its semantics. Broadcast communication allows a transition in one state to enable/disable a transition in another state so context is needed and it is difficult to define its semantics in a compositional manner. The semantics of Dash use the notion of big and small steps, which allow the system to react to environmental input in a concurrent and causal manner, demarcating specific observable moments. Additionally, refinement is not a characteristic of Dash because Dash is intended to be used during the early stages of development to quickly get feedback about the correctness and consistency of requirements, which usually are expressed with a high level of abstraction.

7.5 Summary

Many specification languages have been proposed to formally describe the structure and behaviour of systems. Dash is a new modelling language intended for the declarative specification of abstract behavioural models. This chapter presents some similarities and differences between Dash and other formalisms, covering issues related to syntax, semantics, and tool support.

Chapter 8

Conclusion

This thesis presents Dash, a new language for the formal specification of abstract behavioural models, which combines the control-oriented constructs of statecharts with the declarative modelling of Alloy. From statecharts, Dash inherits a means to specify hierarchy, concurrency and communication, three useful aspects to describe the behaviour of reactive systems. From Alloy, Dash uses the expressiveness of relational logic and set theory to abstractly and declaratively describe structures, data and operations.

A behavioural model is formalised using a transition system, so transitions are particularly important for this type of specifications. In Dash, transitions are first-class constructs, and the language provides features such as factoring, transition comprehension and layering, to systematically declare and organise the transitions of a model. Furthermore, CTL formulas can be used to formally specify behavioural properties.

In Dash, the integration between statecharts and Alloy is done at the semantic level. The semantics of Dash are not compositional in the structure of a model and must address the question of determining which transitions are enabled by an environmental input and internal changes. Since many transitions can be enabled in different concurrent regions, the semantics of Dash use the notion of big and small steps. A step represents a change in a system when a transition or group of transitions is taken. When no more transitions are enabled, the system becomes stable which demarcates the boundary of a big step. The semantics of Dash are stated in terms of the framework by Esmaeilsabzali *et al.* [20], which describes a space of semantic aspects and options for modelling languages that use big steps. Two main reasons were considered for choosing values for the semantic options: first, by being a declarative language, Dash can describe large changes, and second, ease of understanding of a model. Additionally, the semantics address the mismatch between declarative and control-oriented languages regarding the frame problem.

Transition systems described by Dash models are formalised in Alloy by creating a next snapshot relation. The formalisation begins with a transformation to Core Dash, which is the description of a state hierarchy, a set of transitions, initial conditions and state invariants. Alloy signatures and predicates are created to describe the next snapshot relation, and to enforce the semantic choices for Dash. Temporal properties are expanded and transformed to be used with TCMC, and to ensure that behaviours are properly checked at the boundary of each big step. We extended the notion of significance axioms to concurrent Dash models, and they are implemented in the generated Alloy code to avoid spurious instances of a model. A tool for editing and translating Dash models to Alloy is available online at `http://129.97.7.33:8080/dash`.

We developed several case studies to demonstrate the modelling and model checking capabilities of Dash. The case studies were selected to range across the control-oriented versus data-oriented spectrum. While hand-crafted Alloy models may be more concise, the translation of Dash models to Alloy handles the generality of all the variants of state hierarchy and concurrency creates. We extended the notion of significant scopes to hierarchical concurrent models to ensure a snapshot space big enough for checking properties. We carefully designed the Alloy signature definition of snapshots and used Alloy's support for themes to present to users instances of a model that facilitate the interpretation of model checking results in terms of the original Dash model.

Lastly, many different languages and formalisms have been proposed for the specification of behavioural models. The thesis discusses differences and similarities, between Dash and some of the closely related alternatives, regarding syntax, semantics, and tool support.

In the introduction (see Chapter 1) we identified four general problems that existing modelling formalisms have: 1) lack of expressiveness for specification of complex behaviours, 2) intimidating notations, 3) lack of automated reasoning and analysis, 4) no formal or incongruent semantics. Throughout this thesis we have addressed problems 1,3, and 4, demonstrating how Dash supports the specification of complex behaviours (see Chapters 3 and 6), how analysis in Dash is based on model checking which is an automated form of reasoning (see section 6.3), and that Dash has formal semantics (see Chapter 4). Regarding problem 2, we have presented the syntax of Dash (see Chapter 3) which is based on Alloy, a modelling language that is widely used. Alloy's syntax is considered simple and we hope that our extensions to the Alloy language keeps that simplicity. However, we have not provided a thorough analysis of how the syntax of Dash is less intimidating for the specification of behavioural models, and we consider it one of the main issues to be addressed in future work (see section 8.1).

The contributions of this thesis are

- a description of the syntax of Dash, a new language for formally describing abstract declarative behavioural models,
- a description of the features of Dash for conveniently describing transition systems, and accommodating different modelling paradigms,
- a definition of the semantics of Dash which seamlessly integrate hierarchical control constructs with declarative modelling,
- a tool that translates Dash models to Alloy,
- case studies that show the modelling capabilities of Dash across the spectrum of systems, and the verification of behavioural properties,
- an extension of the notion of significance axioms for hierarchical and concurrent models, and
- guidelines to use Alloy's support of *themes* for better visualisation of instances of a model.

8.1 Future Work

Future efforts can improve Dash by focusing on:

Validation. First and foremost, Dash would greatly benefit from a usability study to validate the different features of the language and its modelling checking capabilities. Future research could examine the different syntactic constructs of Dash to identify which features enhance the modelling experience and which constructs need to be refined. Additionally, some stylistic guidelines and conventions could be developed to help modellers best use novel features such as factoring.

Tool Support. Another way to enhance the user experience is to provide more robust tool support. In Dash, it is easy to distribute the declaration of a transition in multiple parts of a model, by using transition templates, transition comprehension, addons, *etc.*,. So tool support could be developed, for example, to let users select a transition and present them with its complete definition, as it would finally appear in the generated Alloy model.

Expressiveness. A Dash model specifies the behaviour of a single instance of a state. The language could be extended in a way to allow the description of the behaviour of an unspecified number of similar states, and then, at analysis time, a bound would be set in a similar manner as scopes are provided for Alloy atoms. Such feature would permit Dash to easily model the behaviour of distributed systems, where the behaviour of different components have considerable similarities. For example, a telephone network where there are multiple phone number nodes each of which with complex but similar behaviours, all interacting as part of a bigger system which is the network.

Performance. In terms of improving the analysis time of Dash models, future work can focus on two areas: providing more efficient expansions of temporal properties and optimising the generic translation of Dash models.

In the first case, as discussed in section 5.11, temporal properties are expanded to distinguish between small and big steps, so behaviour is properly checked on stable snapshots. The current generic expansion consists on transforming the next temporal operators (EX, AX) to be expressed in terms of an until temporal operator (EU, AU). Some tests showed that the until operator seems to be computationally expensive, and that other property-specific transformations could be used to expedite the analysis. For example, a property of the FGS expressed at the Dash level as

$$AG(isAPEngaged \Rightarrow AX\ modesOn)$$

is expanded and expressed in Alloy as

$$AG(stable \wedge isAPEngaged \Rightarrow AX(\neg stable\ AU\ (stable \wedge modesOn)))$$

However, the following alternative that uses the contrapositive and exploits the fact that environmental input remains the same throughout a big step, takes significantly less time to analyse

$$AG(AX(stable \wedge \neg modesOn) \Rightarrow (\neg isAPEngaged))$$

Another way to improve the performance of the analysis of Dash models is to further optimise the generic translation. We found that the order of declarations in an Alloy signature has an impact on the analysis time of a model. For example, by changing the order of two declarations of one of the sample models of Dash [1], it can take up to 130 times longer for the Alloy Analyzer to find an instance of the model. Further investigation could provide some heuristics to organise the declarations and structure of a model to improve the performance.

[1]The model is the EHealth (Environment) available on the Dash website at `http://129.97.7.33:8080/dash`

References

[1] OMG object constraint specification (OCL) specification. http://www.omg.org/spec/OCL/2.4/PDF, 2014. [Online; accessed 18-November-2018].

[2] OMG unified modeling language. http://www.omg.org/spec/UML/2.5/PDF/, 2015. [Online; accessed 18-November-2018].

[3] Antlr (another tool for language recognition), 2018. [Online; accessed 18-November-2018].

[4] Formally specifying UIs, 2018. [Online; accessed 18-November-2018].

[5] Stateflow. https://www.mathworks.com/products/stateflow.html, 2018. [Online; accessed 04-December-2018].

[6] Tutorial for alloy analyzer 4.0, 2018. [Online; accessed 18-November-2018].

[7] Xtext. https://eclipse.org/Xtext/, 2018. [Online; accessed 18-November-2018].

[8] Ali Abbassi, Amin Bandali, Nancy A. Day, and Jose Serna. A comparison of the declarative modelling languages B, Dash, and TLA+. In *2018 IEEE 8th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pages 11–20, Aug 2018.

[9] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[10] Egidio Astesiano, Michel Bidoit, Hélene Kirchner, Bernd Krieg-Brückner, Peter D Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[12] Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[13] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. volume 8559, pages 334–342, 2014.

[14] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic Model Checking of Declarative Relational Models. In *International Conference on Software Engineering*, pages 312–320, May 2006.

[15] Edmund M. Clarke, Orna Grunberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[16] Darren Cofer and Steven P Miller. Formal methods case studies for DO-333. Technical Report NASA/CR2014-218244, NASA Langley Research Center, 2014.

[17] Jin Song Dong, Ping Hao, Sheng Chao Qin, Jun Sun, and Wang Yi. Timed patterns: TCOZ to timed automata. In *International Conference on Formal Engineering Methods*, pages 483–498. Springer, 2004.

[18] Tzilla Elrad, Omar Aldawud, and Atef Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In *Generative Programming and Component Engineering*, pages 189–201, 2002.

[19] Shahram Esmaeilsabzali. *Prescriptive Semantics for Big-Step Modelling Languages.* PhD thesis, 2011.

[20] Shahram Esmaeilsabzali, Nancy A. Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering Journal*, 15(2):235–265, 2010.

[21] Sabria Farheen. *Improvements to Transitive-Closure-based Model Checking in Alloy.* MMath thesis, 2018.

[22] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. DynAlloy: Upgrading Alloy with actions. In *International Conference on Software Engineering*, pages 442–451, 2005.

[23] Marcelo F. Frias, Carlos G. López Pombo, Gabriel A. Baum, Nazareno M. Aguirre, and Thomas S. E. Maibaum. Reasoning about static and dynamic properties in Alloy. 14(4):478–526, 2005.

[24] Juan Pablo Gruer, Vincent Hilaire, Abder Koukam, and P Rovarini. Heterogeneous formal specification based on object-Z and statecharts: semantics and verification. *Journal of Systems and Software*, 70(1-2):95–105, 2004.

[25] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[26] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[27] Ledang Hung and J SOUQUIERES. Contribution for modelling uml state-charts in b [c]. *proc. Of IFM'2002*, pages 109–127, 2002.

[28] i Logix Inc. Statemate 4.0 analyzer user and reference manual, 1991.

[29] Daniel Jackson. Alloy: a lightweight object modelling notation. 11(2):256–290, 2002.

[30] Daniel Jackson. *Software Abstractions.* MIT Press, 2nd edition, 2012.

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

[32] Bilal Kanso and Safouan Taha. Temporal constraint support for ocl. In *Software Language Engineering*, pages 83–103, 2013.

[33] Yun Lu, Joanne M. Atlee, Nancy A. Day, and Jianwei Niu. Mapping template semantics to SMV. pages 320–325, 2004.

[34] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. pages 373–383, 2016.

[35] Brendan Mahony and Jin Song Dong. Blending object-Z and timed CSP: an introduction to TCOZ. In *International Conference on Software Engineering*, pages 95–104. IEEE, 1998.

[36] K. L. McMillan. The SMV system. http://www.kenmcmil.com/language.ps, November 06 1992.

[37] Alexandre Mota, Adalberto Farias, André Didier, and Jim Woodcock. Rapid prototyping of a semantically well founded circus model checker. In *International Conference on Software Engineering and Formal Methods*, pages 235–249. Springer, 2014.

[38] Nimal Nissanke. *Formal Specification: Techniques and Applications*. 1999.

[39] Jianwei Niu, Joanne M Atlee, and Nancy A Day. Template semantics for model-based notations. *IEEE Transactions on Software Engineering*, 29(10):866–882, 2003.

[40] Liam O'Reilly, Markus Roggenbach, and Yoshinao Isobe. Csp-casl-prover: a generic tool for process and data refinement. *Electronic Notes in Theoretical Computer Science*, 250(2):69–84, 2009.

[41] Gianna Reggio and Lorenzo Repetto. CASL-CHART: a combination of statecharts and of the algebraic specification language CASL. In *International Conference on Algebraic Methodology and Software Technology*, pages 243–257. Springer, 2000.

[42] Germán Regis, César Cornejo, Simón Gutiérrez Brida, Mariano Politano, Fernando Raverta, Pablo Ponzio, Nazareno Aguirre, Juan Pablo Galeotti, and Marcelo Frias. DynAlloy analyzer: A tool for the specification and analysis of Alloy models with dynamic behaviour. pages 969–973, New York, NY, USA, 2017. ACM.

[43] Markus Roggenbach. CSP-CASLâĂŤa new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

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

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

[46] Jose Serna, Nancy A. Day, and Shahram Esmaeilsabzali. Dash: Declarative modelling with control state hierarchy (preliminary version). Technical Report CS-2018-04, University of Waterloo, David R. Cheriton School of Computer Science, 2018.

[47] Jose Serna, Nancy A Day, and Sabria Farheen. Dash: A new language for declarative behavioural requirements with control state hierarchy. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pages 64–68. IEEE, 2017.

[48] John Michael Spivey. *The Z Notation: A reference manual.* International Series in Computer Science (2nd ed.). Prentice Hall, 1992.

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

[50] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. pages 632–647, 2007.

[51] Michael von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 128–148, 1994.

[52] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In *Proceedings of the 5th Irish Conference on Formal Methods*, pages 93–115. BCS Learning & Development Ltd., 2001.

[53] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):19, 2009.

[54] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. pages 54–66, 1999.

[55] Pamela Zave. Using lightweight modeling to understand chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.

[56] Qianchuan Zhao and Bruce H Krogh. Formal verification of statecharts using finite-state model checkers. *IEEE transactions on control systems technology*, 14(5):943–950, 2006.

[57] Paul Ziemann and Martin Gogolla. Ocl extended with temporal logic. In *Perspectives of System Informatics: 5th International Andrei Ershov Memorial Conference*, pages 351–357, 2003.

APPENDICES

Appendix A

Dash Grammar

module ::= /moduleDecl/ import* paragraph*

moduleDecl ::= **module** qualName /**[** name,$^+$ **]**/

import ::= **open** qualName /**[**qualName,$^+$**]**/ /**as** name/

paragraph ::= sigDecl | factDecl | funDecl | predDecl
| assertDecl | cmdDecl | stateDecl | escapeBlock

sigDecl ::= /sigQual/ **sig** name,$^+$ /sigExt/
{varDecl,$^+$**}** /block/

sigQual::= **abstract** | **lone** | **one** | **some** | **private**

sigExt ::= **extends** sigRef | **in** sigRef (+ sigRef)*

sigRef ::= **univ** | **seq**/**Int** | **Int** | qualName

stateDecl ::= /**default**/ /**conc**/ **state** name
{stateItem***}**

stateItem ::= stateVar | invariantDecl | initDecl
| addOnDecl | factoringDecl | enterDecl | exitDecl

invariantDecl ::= **invariant** /name/ block

initDecl ::= **init** /name/ block

enterDecl ::= **enter** block

exitDecl ::= **exit** block

addOnDecl ::= **addon** /name/ ((**do** expr | **send** qualName,$^{+}$)) **to** transPattern

transPattern ::= * | qualName,$^{+}$

| (/**from** (* | qualName,$^{+}$)/ /**goto** (* | qualName,$^{+}$)/)

factoringDecl ::= eventDecl | actionDecl | conditionDecl

| transDecl | stateDecl | transTemplate

eventDecl ::= /**env**/ **event** name **{**factoringDecl***}**

actionDecl ::= **action** name **[**expr**]{}**

conditionDecl ::= **cond** name **[**expr**]{**factoringDecl***}**

transDecl ::= **trans** /name/ **{**transBody | transInstance**}**

transInstance :: = qualname**[**expr,$^{+}$**]**

transTemplate :: = **def trans** name paraDecls **{**transBody**}**

transBody ::= /**from** (* | qualName,$^{+}$)/

/**on** qualName,$^{+}$/ /**when** expr*/

/**goto** (* | qualName,$^{+}$)/ /**do** expr*/

/**send** qualName,$^{+}$/

mult ::= **lone** | **some** | **one**

decl ::= /**private**/ /**disj**/ name,$^{+}$ **:** /**disj**/ expr

stateVar ::= **env** decl

factDecl ::= **fact** /name/ block

predDecl ::= **pred** /qualName **.**/ name /paraDecls/ block

funDecl ::= **fun** /qualName **.**/ name /paraDecls/ **:** expr block

paraDecls ::= **(**decl,* **)** | **[** decl,* **]**

assertDecl ::= **assert** /name/ block

cmdDecl ::= /name **:**/ /**run** | **check**/ /qualName | block/ /scope/

scope ::= **for** typeScope,$^{+}$ /**expect** (**0**/**1**)/

| **for** number /**but** typeScope,$^{+}$/ /**expect** (**0**|**1**)/

typeScope ::= /**exactly**/ number /qualName | **int** | **seq**/

expr ::= const | primitiveType | qualName | @name

 | **this** | unOp expr | expr binOp expr

 | expr arrowOp expr | expr **[**expr,***]**

 | expr /**!**| **not**/ compareOp expr

 | expr (=> | **implies**) expr **else** expr | **let** letDecl,$^+$ blockOrBar

 | quant decl,$^+$ blockOrBar | **{**decl,$^+$ blockOrBar**}**

 | **(**expr**)** | block | expr'

primitiveType::= **State** | **Event** | **Boolean**

const ::= /**-**/ number | **none** | **univ** | **iden** | **events** | **conf** | **taken**

unOp ::= **!** | **not** | **no** | mult | **set** | # |**~** | ***** | **^** | **ax** | **ex** | **af** | **ef** | **ag** | **eg**

binOp ::= || | **or** | **&&** | **and** | <=> | **iff** | => | **implies**

 | **&** | + | **-** | ++ | <**:** | **:**> | **.** | **eu** | **au**

arrowOp ::= [mult | **set**/ **->** [mult | **set**/

compareOp ::= **in** | = | < | > | =< | >=

letDecl ::= name = expr

block ::= **{**expr***}**

blockOrBar ::= block | bar expr

escapeBlock ::= **{escape}** *string* **{/escape}**

bar ::= |

quant ::= **all** | **no** | **sum** | mult

qualName ::= /**this**// (name/)* name

name ::= ID

Appendix B

Musical Chairs Model in Dash

```
/*****************************************************************************
 * Title: MusicalChairsWithEvents.dsh
 * Authors: Jose Serna
 * Created: 2018-04-25
 * Last modified: 2018-10-29
 *
 * Notes: The purpose is to present an equivalent model of musical active_chairs
 *        but using events
 *
 ****************************************************************************/

open util/integer

sig Chair, Player {}

conc state Game {
    // Game variables
    active_players: set Player
    active_chairs: set Chair
    occupied: Chair set -> set Player

    env event MusicStarts {}
    env event MusicStops {}

    default state Start {
        trans Walk {
            on MusicStarts
            when #active_players > 1
            goto Walking
            do occupied' = none -> none
        }

        trans DeclareWinner {
            when one active_players
            goto End
        }
    }
```

```

     state Walking {
         trans Sit {
            on MusicStops
            goto Sitting
            do  {
                occupied' in active_chairs -> active_players
                active_chairs' = active_chairs
                active_players' = active_players
                // forcing occupied to be total and
                // each chair mapped to only one player
                all c : active_chairs' | one c .(occupied')
                // each " occupying " player is sitting on one chair
                all p : Chair.(occupied') | one occupied'. p
            }
        }
    }

     state Sitting {
         trans EliminateLoser {
             goto Start
             do {
                active_players' = Chair.occupied
                #active_chairs' = (#active_chairs).minus[1]
             }
         }
     }

     state End {}

    init {
        #active_players > 1
        #active_players = (#active_chairs).plus[1]
        // force all Chair and Player to be included
        active_players = Player
        active_chairs = Chair
        occupied = none -> none
     }
 }

{escape}
// allows to run predicate that generates a significant instance
run significance for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot , 2 EventLabel expect 1
{/escape}

/********************************** PROPERTIES **********************************/

/************************************ SAFETY ************************************/
assert ctl_safety {
    // number of active_players is always 1 greater than number of active_chairs
    ag (#Game/active_players = (#Game/active_chairs).plus[1])

}
check ctl_safety for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot , 2 EventLabel expect 0

```

```
{escape}
assert safety {
    all s: Snapshot | #s.Game_active_players = (#s.Game_active_chairs).plus[1]
}
check safety for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot, 2 EventLabel expect 0
{/escape}


/********************************* EXISTENTIAL ********************************/
one sig Alice extends Player {}
pred ctl_existential {
    // Alice wins in some instance
    ef (Game/End in conf and Game/active_players = Alice)
}
run ctl_existential for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot, 2 EventLabel expect 1

{escape}
pred existential {
    some s: Snapshot | Game_End in s.conf and s.Game_active_players = Alice
}
run existential for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot, 2 EventLabel expect 1
{/escape}

/********************************* FINITE LIVENESS ****************************/
assert ctl_finiteLiveness {
    // ctl_mc[af [{ s : Snapshot | Game_Sitting in s.conf }]]
    af (Game/Sitting in conf)
}
check ctl_finiteLiveness for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot, 2 EventLabel expect 0

{escape}
/******************************* INFINITE LIVENESS ****************************/
assert ctl_infiniteLiveness {
    // number of active_players eventually always reaches and remains at 1
    ctl_mc[af[ag[{ s : Snapshot | #s.Game_active_players = 1}]]]
}
check ctl_infiniteLiveness for exactly 3 Player , exactly 2 Chair,
    exactly 8 Snapshot, 2 EventLabel expect 0
{/escape}
```

Appendix C

Bit Counter Model in Dash

```
/*****************************************************************************
 * Title: BitCounter.dsh
 * Authors: Jose Serna
 * Created: 2018-04-14
 * Last modified: 2018-10-29
 *
 * Notes: Two bit counter model taken from Shahram's PhD thesis
 *
 ****************************************************************************/

conc state Counter {
    event Tk0 {}

    conc state Bit1 {
        event Tk1 {}

        default state Bit11 {}
        state Bit12 {}

        trans T1 {
            from Bit1/Bit11
            on Tk0
            goto Bit12
        }

        trans T2 {
            from Bit12
            on Tk0
            goto Bit11
            send Tk1
        }
    }

    conc state Bit2 {
        event Done {}

        default state Bit21 {}
        state Bit22 {}

        trans T3 {
            from Bit21
            on Bit1/Tk1
            goto Bit22
        }

        trans T4 {
            from Bit22
            on Bit1/Tk1
            goto Bit21
            send Done
        }
    }
}

{escape}
// The final status of bits when done counting
{/escape}
assert ctl_final_bitStatus {
    ag (
        Counter/Bit2/Done in events =>
        {Counter/Bit1/Bit11 + Counter/Bit2/Bit21} in conf
    )
}
check ctl_final_bitStatus for 7 Snapshot, exactly 2 EventLabel expect 0

{escape}
assert final_bitStatus {
    all s: Snapshot| s.stable = True and Counter_Bit2_Done in s.events =>
        {Counter_Bit1_Bit11 + Counter_Bit2_Bit21} in s.conf
}
check final_bitStatus for 7 Snapshot, exactly 2 EventLabel expect 0
{/escape}

{escape}
// The bitcounter has a significant scope of 10 Snapshots
run significance for 7 Snapshot, exactly 2 EventLabel expect 1

// Model is responsive
assert ctl_model_responsive {
    ctl_mc[
        ag[
            imp_ctl[
                {s: Snapshot| Counter_Tk0 in s.events},
                af[{s: Snapshot |
                    s.stable = True and
                    (Counter_Bit1_T1 in s.taken or
                    Counter_Bit1_T2 in s.taken or
                    Counter_Bit2_T3 in s.taken or
                    Counter_Bit2_T4 in s.taken)
                }]
            ]
        ]
    ]
}
```

```
check ctl_model_responsive for 7 Snapshot, exactly 2 EventLabel expect 0


assert model_responsive {
    all s: Snapshot | s.stable = True and Counter_Tk0 in s.events =>
        some s': s.*nextStep | s.stable = True and
                    (Counter_Bit1_T1 in s'.taken or
                    Counter_Bit1_T2 in s'.taken or
                    Counter_Bit2_T3 in s'.taken or
                    Counter_Bit2_T4 in s'.taken)
}
check model_responsive for 7 Snapshot, exactly 2 EventLabel expect 0
{/escape}
```

Appendix D

Steps Alloy Module

```
/******************************************************************************
 * Title: steps.als
 * Authors: Jose Serna - jserna@uwaterloo.ca
 *
 * Notes: This module helps defining step relations for transition systems.
 *        Several axioms are included to get a significant model.
 *
 *****************************************************************************/

module steps[S]

open ctl[S]

    one sig Step {
        initial: some S,
        next_step: S -> S,
        equality:  S -> S
    }

    // A snapshot is a set of control states, a variable evaluation, and a set
    // of events.
    abstract sig BaseSnapshot {
        /** Label control states */
        conf: set StateLabel,
        /** Semantics consistency */
        taken: set TransitionLabel
    }

    fact {
        all s: S | s in BaseSnapshot
        Step.next_step = nextState
        Step.initial = initialState
    }

    // These functions must be defined by the calling code
    /** Define the elements that represent the initial state of the system */
    fun initial: S { Step.initial }
    /** Define the next state relation */
    fun nextStep: S -> S { Step.next_step }
    /** Define the criteria to consider two elements as equal */
    fun equals: S->S { Step.equality }

/****************************** EVENT SPACE **********************************/
    abstract sig EventLabel {}
    abstract sig EnvironmentEvent, InternalEvent extends EventLabel {}

/****************************** STATE SPACE **********************************/
    abstract sig StateLabel {}

/***************************** TRANSITIONS ***********************************/
    abstract sig TransitionLabel {}

/************************* Significance Axioms *******************************/

    pred reachabilityAxiom {
        all s : S | s in S .((Step.initial) <: * (Step.next_step) )
    }

    pred equalityAxiom {
        all s, s': S |  s->s' in Step.equality => s = s'
    }

    // The system is always in some state
    assert check_some_conf {
        ctl_mc[ag[{s: S | some s.conf}]]
    }
    check check_some_conf for 10 expect 0
```