DASH: A New Language for Declarative Behavioural Requirements with Control State Hierarchy

Jose Serna, Nancy A. Day, and Sabria Farheen
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
Email: {jserna,nday,sfarheen}@uwaterloo.ca

***Abstract*—We present DASH, a new language for describing formal behavioural models of requirements. DASH combines the ability to write abstract, declarative transitions (as in Z or Alloy) with a labelled control state hierarchy (as in the Statecharts family of languages). The key contribution of DASH is the combination of explicit support for user-level abstractions that create and factor sets of transitions, such as state hierarchy, and the use of full first-order logic to describe the transitions.**

I. Introduction

Languages for the creation of abstract, declarative, formal models of behavioural requirements (*e.g.,* Z [1]) are usually very distinct from languages used for describing behavioural models in model-driven engineering (*e.g.,* Statecharts [2]). Both types of models are operational, in that they specify the operations/steps of the system. But the differences exist because formal requirements models must support the specification of abstract and complex concepts (*e.g.,* uninterpreted and infinite types, uninterpreted functions and relations, and quantified expressions), which have mainly been supported by theorem proving-based analysis. Languages for model-driven engineering (MDE) are usually geared towards code generation, and automated analysis has been done by model checking, which generally requires models to use primitive, finite types.

The Alloy Analyzer [3] has blurred this difference because its automatic analysis is based on finite model finding, where finite scopes for abstract types allow problems to be mapped to propositional logic. The encoding of the temporal logic model checking problem into first-order logic (*e.g.,* [4][5]) and the use of SMT solvers [6] for model checking have continued to attenuate the differences.

However, Alloy provides no explicit support for describing behavioural/dynamic models, *i.e.,* transition relations. Electrum [7] has extended the Alloy language to provide some syntactic support for referring to previous and next state values in a description of a transition relation. TLA+ [8] provides similar support for creating and model checking behavioural models. The nuXmv [4] tool supports multiple model checking algorithms for infinite state systems, but its input language supports limited types of data and operations (*e.g.,* integers, reals). Lacking is a language that provides both the abstractions of first-order logic (FOL) and user-level modelling abstractions, such as labelled control states and events. Furthermore, language support is needed for the systematic organization of transitions within models.

In this paper, we present a new language called DASH[1], which provides explicit user-level modelling primitives to support a formal description of a set of abstract, declarative transitions. By 'abstract', we mean including uninterpreted/user-declared types and functions/relations. By 'declarative', we mean describing a transition as a relationship between previous and next states using all of first-order logic. A transition is not necessarily executable.

DASH is a layer on top of Alloy. It uses Alloy syntax to describe types and elements of the system. User-declared types are possible. User-declared functions and relations can be included as part of the system; these can be axiomatized, defined or left uninterpreted. It also uses Alloy to describe symbolically the conditions and actions of transitions in first-order logic, including quantifiers[2]. Creating models using abstract concepts frees engineers to concentrate on the important details that they want to codify without sacrificing precision for unknown/irrelevant aspects of models.

Beyond Alloy, DASH provides explicit support for describing and **factoring** transitions into groups to permit modular development. We began with the idea of supporting Statecharts' AND- and OR- control state hierarchy. Control states are a popular user-level abstraction in MDE that groups together moments in time that have common future behaviours. We quickly realized that control states are only one method for factoring the set of transitions. As a textual language, DASH can support multiple, nested methods of factoring the set of transitions to aid the systematic representation of models. Events are another common user-level modelling abstraction so DASH supports factoring the transition set by events. Factoring by conditions is another grouping supported in DASH.

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

[2]In this paper, an "Alloy formula" means an "Alloy formula without transitive closure", as it is the one operator in Alloy that is not first-order.

Furthering the idea of grouping the description of related transitions, DASH supports the creation of transitions by patterns, which we call **transition comprehension**. For example, if every state has a transition to an error state, this can be defined in one statement in DASH using a pattern to describe the possible source states.

DASH also supports **layering**, which allows a modeller to describe "add-on" parts of a transition in different places in a model. These descriptions are layered together to create the complete description of the transition. Addons facilitate aspect-oriented modelling [9]. For example, an action common on all transitions (such as a clock tick) can be described once and layered onto all the transitions.

DASH accomplishes all of the following:

- Adding hierarchical labelled control states to the Alloy Language for describing transitions.
- Adding user-declared types and operations, and first-order logic formulae in the conditions and actions of Statecharts.
- Creating user-level syntax for the control state hierarchy that we previously proposed as a datatype to be used in transitions described in SMT-LIB [10].
- Offering new factoring, patterning, and layering abstractions to describe and organize systematically the transitions of a model.

The key contribution of DASH is the combination of explicit support for user-level abstractions that create and factor sets of transitions, and the use of full first-order logic to describe the transitions. From a user-level perspective, factoring transitions by states, events, or conditions are higher-level abstractions (than a declarative transition by itself) that group together Kripke structure transitions to describe systematically the system's behaviour.

In this paper, we report on the language definition for DASH. Creating several examples[3] previously written in languages such as Alloy, Z, and Statecharts has helped us to create a grammar for DASH and a parser using Xtext [11]. Besides the parser, Xtext facilitates creating robust tool support for a language that includes linking, type checking, compiling and editing support. This infrastructure allows us to rapidly test and improve our language. The rest of the paper is organized as follows. Section II describes our new language via an example. In Section III, we provide an overview of our planned semantics for DASH. In Section IV, we compare the features of DASH with existing formal languages for describing abstract behavioural models. In Section V, we describe the future work to create tool support for DASH models.

II. DASH

DASH is a language for describing a set of transitions; the transitions are combined to form the transition relation of a behavioural model. A DASH model is a set of **states** within an Alloy module. Figure 1 shows a fragment of a DASH model for a heating system example based on Day [12].

Each state consists of declarations of system elements and a set of transitions. Declarations of variables are in Alloy syntax as on lines 6-10 in the example. There can be declarations of types and global constants outside of a state (*e.g.,* the type `ValvePosition` on line 1).

Modellers describe **transitions** using the following template:

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

These keywords were chosen to match the way a transition is described in English. Lines 35-44 and 51-55 are examples of transitions in the heating system model.

Every part of a transition definition is optional. If the model uses labelled control states, then the source and destination state labels are in the `from` and `goto` parts. If the model uses events, then the triggering events are listed in the `on` part and generated events are listed in the `send` part.

The condition (`when` part) and action (`do` part) can contain any formula in Alloy. A formula can be labelled and defined separately from where it is used (as on line 15). As in Z and Electrum, DASH uses the primed version of variables to refer to their values in the next state (*e.g.,* line 18).

A set of transitions in one DASH state described in the above form alone is roughly equivalent to writing a number of Z schemas.

State blocks can be nested to form a **control state hierarchy**. The set of top-level DASH states is a set of AND-states. Within a state, there can be AND-, OR- and basic substates. AND-states have the keyword `conc` when they are introduced, as on line 30, and OR-states and basic states require no keyword (line 33). The default substate is designated using the `default` keyword as on line 31. These labelled control state blocks can include declarations (line 6), transitions (line 51), and substates (line 46). While DASH models assume global communication, the state hierarchy is used as a scoping mechanism for creating partitioned namespaces. All names of elements declared within a given state must be unique. However, names can be reused at other levels of the state hierarchy. To refer to elements outside of the current scope, fully qualified names must be used.

States can be used to **factor**/group transitions, *i.e.,* transitions specified within a state have to do with that state. If the `from` field is omitted from a transition (*e.g.,* `t1` on line 35) then the source state is the parent state of the transition. Whereas if the `goto` is omitted (*e.g.,* `t2` on line 40) the parent state is the destination of the transition. Furthermore, both the `from` and the `goto` parts of a transition can be omitted to define a looping transition. We find this shortcut particularly useful when a state has no nested states.

[3]A repository of DASH examples and the full grammar can be found in https://cs.uwaterloo.ca/~nday/2017-modre

```
1  abstract sig ValvePosition {}
2  abstract sig Room {}
3  ...
4
5  conc state HeatingSystem {
6    valvePosition: Room -> ValvePosition
7    desiredTemp: Room -> Int
8    actualTemp: Room -> Int
9    occupied: Room
10   requestHeat: Room
11
12   event activate {}
13   event deactivate {}
14
15   action adjValve [
16     all r:occupied |
17       r.actualTemp < r.desiredTemp =>
18       r.valvePosition' = OpenPosition
19   ] {}
20
21   condition roomsNeedHeat [
22     some requestHeat
23   ] {}
24
25   init {
26     all r: Room |
27       r.valvePosition = ClosedPosition
28   }
29
30   conc state Controller {
31     default state Off { }
32     state Error { }
33     state On {
34       default state Idle{
35         trans t1 {
36           when roomsNeedHeat
37           goto HeaterActive
38           send activate
39         }
40         trans t2 {
41           from HeaterActive
42           when (not roomsNeedHeat)
43           send deactivate
44         }
45       }
46       state HeaterActive{
47         default state ActivatingHeater{}
48         state HeaterRunning{}
49       }
50
51       trans t3 {
52         on heatSwitchOff
53         goto Off send deactivate
54       }
55       trans t4 {on furnaceFault goto Error}
56     }
57   }
58
59   conc state Bedroom {
60     default state NoHeatRequestested {}
61     state HeatRequested {
62       default state  IdleHeating{}
63       state WaitForCool{
64         trans t5 {on waitedForCool do adjValve}
65       }
66     }
67   }
68 }
```

Fig. 1. Heating System in DASH

Labelled control states are only one means of factoring the set of transitions. Transitions can be factored by events as in:

```
event deactivate {
  trans off1 {from Activating goto Off}
  trans off2 {from Running goto Off}
}
```

This primitive both declares events and factors the transitions by events, grouping together transitions with a common triggering event. In the example, the event `deactivate` triggers transitions `off1` or `off2`.

Similarly, transitions can be factored by conditions, resembling a switch/case statement, as in:

```
condition C1 [<formula>] {
  <transitions>
}
```

The condition is any formula in Alloy. It is included in the **when** part of every transition declared within a **condition** block. The **condition** keyword also assigns a label to a formula (as on line 21) and the label can be used in other expressions.

The state, event, and condition factoring keywords not only help modellers with the systematic organization of transitions by user-level abstractions, but they can also help with the completeness and consistency of models. For example, tool support for DASH can check if all conditions have been considered when factoring by condition.

Even after grouping related transitions, many transitions have similarities. DASH supports the use of patterns to create a set of transitions with only one statement to make models simpler to read and less tedious to write. We call such a pattern a **transition comprehension**. The use case for this can be illustrated by an example of a telephone number model abstracted from Vakili [13]. The model consists of a phone number and the transitions between its various states, such as, calling, talking, idle, busy, etc. The transitions are triggered by specific events that dictate the phone number's behaviour. One of the characteristics of this model is that the phone number, regardless of its current control state, returns to the `idle` state whenever the event `hang_up` occurs. This captures the behaviour that the owner of the phone number can choose to hang up her phone at any time. One way to model this behaviour without transition comprehension, is to write a transition originating from every control state, such as:

```
trans calling_to_idle {
  from calling on hang_up goto idle
}
trans talking_to_idle {
  from talking on hang_up goto idle
}
trans busy_to_idle {
  from busy on hang_up goto idle
}
...
```

With the transition comprehension feature, DASH allows us to write just one expression to define this set of transitions:

```
trans to_idle {
  from * on hang_up goto idle
}
```

Here, the * symbol represents all states within the current scope. Instead of the wildcard, modellers can provide a list of particular state names. Patterns are also possible for the `goto` definition. Alternatively, state hierarchy can be used to model the same behaviour by encapsulating the affected source states in a superstate, and defining a transition from that superstate. However, transition comprehension gives modellers flexibility to describe similar behaviour from different levels of the state hierarchy.

Another DASH feature to facilitate convenient and systematic modelling of transitions is **layering**. Using the `addon` keyword, we can describe conditions and/or actions to add to a set of transitions, specified by a pattern. For example:

```
1  addon (do incErrorCounter)
2    to (from * goto Error)
3  addon (do incErrorCounter) to t19
```

Declaring the initial state and invariants of a system is also possible in DASH. The keywords `init` and `invariant` are used respectively, followed by a block of Alloy formulae. Figure 1 on line 25 defines the condition on the initial state of the heating system. Additionally, there are well-formedness constraints and quality checks for DASH models that we omit due to lack of space.

While our original goal with DASH was to add the common MDE modelling abstraction of state hierarchy to declarative specifications, we believe that we have gone further to enhance the modeller's experience by supporting transition comprehension, layering, and factoring by more than just control state. As in Alloy, there are multiple ways to use these features together to construct equivalent models, which makes DASH flexible for different modelling styles. A flexible modelling language that supports the systematic and convenient description of transitions will hopefully help in obtaining a high quality behavioural model.

III. Semantics

The first step in determining the meaning of a DASH model is to reduce the model to a finite set of transitions and a control state hierarchy. This automatic syntactic expansion step involves 1) flattening the effect of factoring (*i.e.,* combining the factors into the meaning of the transitions); 2) expanding the transition comprehension patterns, and; 3) merging the add-ons with all relevant transitions.

Next, we combine the set of transitions together to produce a relation in first-order logic between two configurations of the model, plus a predicate describing the initial configurations. Together, an initial predicate and next configuration relation form a symbolic Kripke Structure [13], which has a possibly infinite set of reachable configurations.

With a control state hierarchy, the semantics of a transition system cannot be reduced to either a conjunction or a disjunction of the relations for each individual transition. Coordination of transitions is required to describe the meaning of concurrent transitions. For example, to assert that an event has not occurred requires global knowledge across all transitions that are taken in a step.

A significant issue in the semantics is the differing approaches in Statecharts languages and declarative requirements languages with respect to changes in elements in a step: in Statecharts, a variable does not change its value unless explicitly stated; in Z, Alloy, and SMV [14], a variable must be explicitly constrained to retain its value otherwise it can change non-deterministically. When transitions can be taken concurrently, the Statecharts approach is needed otherwise the model would frequently have inconsistencies. However, it is useful for a modeller to state explicitly what variables do not change in a transition. We plan a hybrid approach: a statement of the non-changing variables in a transition will be considered local to a transition and not global across concurrent transitions. However, a top-level directive will indicate whether the user must explicitly state the non-changing variables in a transition or whether these should be inferred from the actions of the transition. In addition, DASH will have the Statecharts semantics that if no transition is taken in a step, the elements do not change (implicit looping transition).

In stating the semantics of DASH, we plan to follow an approach that describes constraints on the set of transitions that can be taken in a step [12][15]. By thinking of the transitions as first-class entities, there is a clear separation of concerns in stating the semantics. The preconditions (events and conditions) on the transitions enforce which transitions are enabled. The state hierarchy constrains which enabled transitions can be taken together. If a model is nondeterministic, there can be multiple sets of transitions that can be taken in a step. The postconditions and generated set of events are enforced for the set of transitions chosen. Eventually, we plan to support the range of semantics possible for languages with control state hierarchies as described in Esmaeilsabzali *et al.* [16]. This will enable DASH to not only support synchronous communication, but also to support asynchronous message passing, which is required to model distributed systems.

IV. Related Work

In this section, we compare DASH to both formal, abstract specification languages and some relevant notations used to create models for model checking analysis.

There are a number of languages for formally describing abstract behavioural requirements: Z [1], VDM [17], B [18], ASMs [19], TLA+ [8], Electrum [7], DynAlloy [20], and SAL [21] [22]. These languages are based on logic and set theory so they support abstract declarative models. These languages provide some syntax (*e.g.,* unprimed and primed system variables) or packaging mechanisms (*e.g.,* schemas in Z) for describing transitions. Labelled control state hierarchy can be represented encoded in variables (*e.g.,* [23]). However, none of these languages explicitly support the representation of control state hierarchy or other methods of factoring, which are included in DASH.

The use of labelled control state hierarchy comes from the Statecharts family of languages, which includes StateMachines in UML [24]. These languages usually have a fixed condition

and action language that does not allow for declarative specification of user-defined datatypes and operations. OCL [25] is a formal language for expressing invariants, pre and post conditions, which can be added onto parts of a UML model (described in a context), somewhat similar to DASH's add-on construct. In contrast, DASH allows the use of FOL formulae directly in transition conditions and actions and will have a fully formal semantics. In addition, DASH offers the flexibility of factoring, layering, and transition comprehension to describe a model.

Model checking tools usually have fairly primitive input languages with no support for abstract datatypes and operations. In SMV [14], transitions can be described using case/switch statements, and labelled control state hierarchy and its semantics can be encoded, but is not supported natively. Abstract datatypes and operations can be encoded into these languages as in Chang and Jackson [26]. nuXmv [4] includes support for infinite datatypes such as integers and reals, but not user-defined types.

Alloy supports set theory primitives and other syntactic sugar for first-order logic descriptions, but no explicit support for transitions. Electrum [7] extends Alloy with primed variables to refer to the next values of variables. DynAlloy [20] and [27] extend Alloy to allow the definition of actions to model state changes relating pre and post-conditions on transitions. TLA+ [8] can describe transition systems but has no support for labelled control states. SMT-LIB [28] is intended as a machine-readable version of FOL with theories for datatypes with decision procedures. Previously we created a way to represent labelled control state hierarchy as a datatype in SMT-LIB [10], but no user-level language to describe and group these transitions.

V. Conclusion

We have presented DASH, a new language for describing abstract, declarative transitions of behavioural models. Unlike existing modelling languages, DASH supports both descriptions of transition behaviour in full first-order logic, and common user-level, behavioural MDE abstractions such as labelled, hierarchical control states and events. DASH is a textual language that provides convenient mechanisms for users to create and organize their models: 1) factoring/grouping the set of transitions by control state, event, and condition; 2) transition comprehension for describing a set of transitions using a pattern; and 3) layering together parts of transitions described in different places in a model.

In the future, we plan to:

1) Extend DASH with new features for modularity, such as parameterized states and quantification over states.
2) Build tool support, which initially will be a translator to Alloy, but in the future may also include translations to SMT solvers.
3) Explore model checking of DASH models.

References

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

[2] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987.

[3] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM TOSEM*, vol. 11, no. 2, pp. 256–290, 2002.

[4] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *CAV*, vol. 8559 of *LNCS*, pp. 334–342, Springer, 2014.

[5] A. Vakili and N. A. Day, "Reducing CTL-Live model checking to first-order logic validity checking," in *FMCAD*, pp. 215–218, IEEE, 2014.

[6] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, *Satisfiability Modulo Theories*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, ch. 26, pp. 825–885. 2009.

[7] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *FSE*, pp. 373–383, ACM, 2016.

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

[9] T. Elrad, O. Aldawud, and A. Bader, "Aspect-oriented modeling: Bridging the gap between implementation and design," in *Generative Programming and Component Engineering*, pp. 189–201, Springer, 2002.

[10] N. A. Day and A. Vakili, "Representing hierarchical state machine models in SMT-LIB," in *MISE @ ICSE*, pp. 67–73, ACM, 2016.

[11] "Xtext." https://eclipse.org/Xtext/. [Online; accessed 05-June-2017].

[12] N. A. Day, *A Framework for Multi-Notation, Model-Oriented Requirements Analysis*. PhD thesis, Univ. of British Columbia, Dept. of Comp. Sci., 1998.

[13] A. Vakili, *Temporal Logic Model Checking as Automated Theorem Proving*. PhD thesis, Univ. of Waterloo, Cheriton School of Comp. Sci., 2016.

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

[15] S. Esmaeilsabzali and N. A. Day, "Prescriptive semantics for big-step modelling languages," in *FASE*, vol. 6013 of *LNCS*, pp. 158–172, Springer, 2010.

[16] S. Esmaeilsabzali, N. A. Day, J. M. Atlee, and J. Niu, "Deconstructing the semantics of big-step modelling languages," *REJ*, vol. 15, no. 2, pp. 235–265, 2010.

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

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

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

[20] M. F. Frias, J. P. Galeotti, C. G. López Pombo, and N. M. Aguirre, "DynAlloy: Upgrading Alloy with actions," in *ICSE*, pp. 442–451, ACM, 2005.

[21] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, *et al.*, "An overview of SAL," in *Proceedings of the 5th NASA Langley Formal Methods Workshop*, 2000.

[22] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *CAV*, pp. 496–500, Springer, 2004.

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

[24] "OMG unified modeling language." http://www.omg.org/spec/UML/2.5/PDF/, 2015. [Online; accessed 05-June-2017].

[25] "OMG object constraint specification (OCL) specification." http://www.omg.org/spec/OCL/2.4/PDF. [Online; accessed 07-June-2017].

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

[27] J. P. Near and D. Jackson, "An imperative extension to Alloy," in *ABZ*, pp. 118–131, Springer, 2010.

[28] C. Barrett, P. Fontaine, and C. Tinelli, *The SMT-LIB Standard: Version 2.5*, 2015.